UNIVERSITY OF CALIFORNIA
Los Angeles

Extracting Latent Semantic Information from Multi-domain Contents by Decalaritive Language and Efficient Algorithm

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Mingda Li

2020

ABSTRACT OF THE DISSERTATION

Extracting Latent Semantic Information from Multi-domain Contents by Decalaritive Language and Efficient Algorithm

by

Mingda Li
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Junghoo Cho, Co-chair
Professor Carlo Zaniolo, Co-chair

With large amounts of data continuously generated by intelligence devices, efficiently analyzing huge data collections to unearth valuable insights has become one of the most elusive challenges for both academy and industry. The key elements to establishing a scalable analyzing framework should involve a well-proposed algorithm to integrate all available information, an optimal training strategy and a declaritive developing interface. In this dissertation, we focus on the comprehensive enhancement through the scalable analyzing framework by (i) domain-specific framework designs to enrich the captured information, (ii) a sample-efficient training method adaptive to a wide variety of multi-class classifiers with extreme large output-class size, and (iii) a cross-language interface for succinct expressions of recursions in advanced analytics.

Our contributions in this thesis are thus threefold: First, we aim to utilize variants of recurrent neural network (RNN) to incorporate some enlightening sequential information overlooked by the previous works in two different domains including Spoken Language Understanding (SLU) and Internet Embedding (IE). In SLU, we address the problem caused by solely relying on the first best interpretation (hypothesis) of an audio command through a series of new architectures comprising bidirectional LSTM and pooling layers to jointly utilize the other hypotheses' texts or embedding vectors, which are neglected but with valuable information missed by the first best hypothesis. In IE, we propose the DIP, an extension of RNN, to build up the internet coordinate

system with the IP address sequences, which are assigned hierarchically and encode structural information of the network but are also unnoticed in previous distance-based internet embedding algorithms. Both DIP and the integration of all hypotheses bring significant performance improvements for the corresponding downstream tasks. Then, we investigate the training algorithm for multi-class classifiers with a large output-class size, which is common in deep neural networks and typically implemented as a softmax final layer with one output neuron per each class. To avoid expensive computing the intractable normalizing constant of softmax for each training data point, we analyze and enhance the well-known negative sampling to the amplified negative sampling algorithm, which gains much higher performance with lower training cost. Finally, for the ubiquitous recursive queries in advanced data analytics, on top of BigDatalog and Apache Spark, we design a succinct and expressive analytics tool encapsulating the functionality and classical algorithms of Datalog, a quintessential logic programming language. We provide the Logical Library (LLib), a Spark MLlib-like high-level API supporting a wide range of Datalog algorithms and the Logical DataFrame (LFrame), an extension to Spark DataFrame supporting both relational and logical operations. The LLib and LFrame enable smooth collaborations between logical applications and other Spark libraries and cross-language logical programming in Scala, Java, or Python.

The dissertation of Mingda Li is approved.

Yingnian Wu

Yizhou Sun

Carlo Zaniolo, Committee Co-chair

Junghoo Cho, Committee Co-chair

University of California, Los Angeles

2020

To my mother, father and girl friend.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my advisor Professor John Cho and Professor Carlo Zaniolo.

VITA

2011-2015	B.S. Computer Science and Technology Harbin Institute of Technology Harbin, China
2015–2016	Research Assistant ScAi Laboratory University of California, Los Angeles Los Angeles, California
2016–2020	Teaching Assistant Computer Science Department University of California, Los Angeles Los Angeles, California
2016	Research Intern Teradata Los Angeles, US
2017	Research Intern NEC Lab Princeton, US
2018	SDE Intern Amazon AWS Redshift Palo Alto, US
2019	Applied Scientist Intern Amazon Alexa AI Boston, US

PUBLICATIONS

Mingda Li, Weitong Ruan, Xinyue Liu, Luca Soldaini, Wael Hamza, Chengwei Su. "Improving Spoken Language Understanding By Exploiting ASR N-best Hypotheses." ArXiv 2020. To be submitted to *SLT 2021*

Mingda Li*, Zijun Xue*, Junghoo Cho. "Amplified Negative Sampling: Sample-Efficient Training for a Large-Class Classifier." In review by the *KDD 2020*

Jin Wang, Chunbin Lin, Mingda Li, Carlo Zaniolo. "Boosting Approximate Dictionary-based Entity Extraction with Synonyms." Accepted by *Information Sciences Journal 2020*

Mingda Li, Weitong Ruan, Xinyue Liu, Luca Soldaini, Wael Hamza, Chengwei Su. "Multi-task Learning of Spoken Language Understanding by Integrating N-Best Hypotheses with Hierarchical Attention." To be submitted to *COLING 2020*

Mingda Li, Jin Wang, Youfu Li, Carlo Zaniolo. "LLib and LFrame: Logical Libraries and DataFrames for More Expressive Logical Programming." To be submitted to *ICLP 2020*

Jin Wang, Jiacheng Wu, Mingda Li, Jiaqi Gu, Ariyam Das, Carlo Zaniolo. "Formal Semantics and High Performance in Declarative Machine Learning using Datalog." In review by *VLDB Journal 2020*

Mingda Li, Cristian Lumezanu, Bo Zong, Haifeng Chen. "Learning-based spoofing detection." In review by the *EuroS&P WTMC 2020*

Jin Wang, Chunbin Lin, Mingda Li, Carlo Zaniolo. "An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms." Accepted by *EDBT 2019*

Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda li, Jin Wang. “Monotonic Properties of Completed Aggregates in Recursive Queries.” *ArXiv 2019*

Ariyam Das, Youfu Li, Jin Wang, Mingda Li, Carlo Zaniolo. “BigData Applications from Graph Analytics to Machine Learning by Aggregates in Recursion.” Accepted by *ICLP 2019*

Mingda Li, Cristian Lumezanu, Bo Zong, Haifeng Chen. “Deep Learning IP Network Representations” **BEST PAPER AWARD** of *ACM SIGCOMM Big-DAMA 2018*

Mingda Li, Cristian Lumezanu, Bo Zong, Haifeng Chen. “Learning IP Network Representations.” Accepted by *ACM SIGCOMM CCR 2018*

Zijun Xue, Ruirui Li, Mingda Li. “Recent Progress in Converesational AI.” *KDD Conversational AI workshop 2018*

Youfu Li, Mingda Li, Ling Ding, Matteo Interlandi . “RIOS: Runtime Integrated Optimizer for Spark.” Accepted by *SOCC 2018*

Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, Miryung Kim. “Automated Debugging in Data-Intensive Scalable Computing.” Accepted by *SOCC 2017*

In the first part, we aim to utilize the variants of recurrent neural network (RNN) to incorporate some enlightening sequential information overlooked by the previous works in different domains and tasks. We find, in Spoken Language Understanding, only the best of automatic speech recognition (ASR) interpretations (hypotheses) for an input audio signal is utilized to understand the intent, while the rest hypotheses containing fragmented important messages are ignored. We investigate a series of methods to jointly utilize top n interpretations by integrating the hypothesized text or hypothesis embedding vectors with BiLSTM and achieve significant accuracy improvements for intent or domain classification. Similarly, while embedding the Internet structure, only the distances among IP address are utilized to build up the network coordinate system but the information contained by the IP address is unnoticed. We propose the DIP, a deep learning based framework for IP network representations, which normalizes each IP address to seperate sequences by the volume of contained routable information within IP bits and exploits a variant of RNN for a low-dimensional representation.

In the second part, we investigate the training algorithm for multi-class classifiers with a large output-class size, which is common in deep neural networks and typicaly implemented as a softmax layer in the final layer containing one output neuron per each word. It is prohibitively to calculate the intractable normalizing constant of softmax for each training data point. We analyze the well-known negative sampling and propose the amplified negative sampling algorithm, which gains higher performance with lower training cost.

CHAPTER 1

Introduction

1.1 Motivations

In the big data era, we have witnessed the rising demand to efficiently and conveniently extracting insights from large-scale data sets for decision making in different domains. The demand has driven researchers to propose various neural network-based algorithms revolutionizing many fields, ranging from image processing (He et al., 2016), natural language processing (Devlin et al., 2018) to speech recognition (Amodei et al., 2016), etc. In addition, the big data analyzing and machine learning platforms in open sorce and commercial markets like PyTorch (Paszke et al., 2019), Tensorflow (Abadi et al., 2016) and Apache Spark (Zaharia et al.) are continuously built up, to provide maximum flexibility and speed while implementing the analyzing pipeline with exisiting or user-defined algorithms.

succinct large-scale data processing.

1.2 Thesis Outline

1.3 Contributions

CHAPTER 2

Extracing Latent Information from IP Network

In this chapter, we firstly present DIP, a deep learning based framework to learn structural properties of the Internet, such as node clustering or distance between nodes, from the IP addresses. Existing embedding-based approaches use linear algorithms on a single source of data, such as latency or hop count information, to approximate the position of a node in the Internet. In contrast, DIP computes low-dimensional representations of nodes that preserve structural properties and non-linear relationships across multiple, heterogeneous sources of structural information, such as IP, routing, and distance information. Using a large real-world data set, we show that DIP learns representations that preserve the real-world clustering of the associated nodes and predicts distance between them more than 30% better than a mean-based approach. Furthermore, DIP accurately imputes hop count distance to unknown hosts (*i.e.*, not used in training) given only their IP addresses and routable prefixes. Our framework is extensible to new data sources and applicable to a wide range of problems in network monitoring and security.

Then, we consider an important topic, spoofing defense, based on the full knowledge of the internet structural properties. Map-based IP spoofing defenses associate source IPs to immutable structural properties of the Internet, such as paths, hop counts, or neighbors to a target, and filter out packets whose header information does not match the maps. Although accurate, existing methods lack sufficient coverage. Network maps on AS border routers do not detect spoofed packets that traverse unprotected networks. Host maps at the edge protect only against spoofed packets with source IPs known by the host. We propose to learn IP maps by constructing a structural model (or embedding) of the Internet from a limited number of measurements. We study the feasibility of learned IP maps by combining hop count filtering, a known map-based spoofing defense mechanism that maps IPs to hop counts to a target, and DIP, a deep learning based learning algorithm

that computes vector representations of IPs that preserve hop count distance between them. Using a large data set of hop counts between Internet hosts, we show that learned maps can detect packets spoofed with almost *any* IP address and traversing *any* path using an embedding generated from only a few thousand IP addresses and hop counts between them. In addition, our embeddings are general: an Internet model trained for a set of hosts can be used by any other host to generate new IP maps with little loss in accuracy.

2.1 Deep Learning IP Network Representations

2.1.1 Introduction

The ability to map, analyze, and understand the structure of the Internet helps network management and operations by revealing opportunities for improvement or potential design flaws. For example, accurately predicting the closest server is critical in peer selection and load balancing (Miao et al., 2017). Knowing how remote IPs are clustered can help diagnose anomalous events such as spoofing attacks (Jin et al., 2003). A holistic view of the network and its structure is essential towards achieving the vision of self-driving networks (Feamster and Rexford, 2017).

Most previous attempts to uncover the Internet structure relying on active probing from multiple vantage points using tools such as *traceroute* and *ping* (Levchenko et al., 2017; Spring et al., 2003). Such techniques provide fine-grained introspection (*i.e.*, can measure specific properties in specific parts of the network, such as the latency of a path) but pose a significant cost in terms of network overhead.

In contrast, embedding-based approaches use fewer, strategic measurements (Dabek et al., 2004; Eriksson et al., 2008) or passive observations on network traffic (Eriksson et al., 2007) to learn vector representations for the network end-hosts in a low-dimensional space. The representations approximate the positions of hosts in the Internet and are used to recover structural network properties, such as distance between nodes or clustering of nodes. However, the complexity of the Internet and the sparse input data make it difficult to compute accurate representations. Oftentimes, embedding approaches rely on additional data sources, which cannot be easily used in the embed-

ding process, to refine and tune the final embeddings. For example, several embedding methods build representations based on distance-based metrics, such as latency or hop count, and then refine (or even replace) the final representations using additional probes or static information such as AS membership or routing information (Dabek et al., 2004; Eriksson et al., 2009).

The emergence of deep learning as a powerful tool to extract hidden features in data calls for revisiting the problem of learning network representations through embedding. In particular, deep learning techniques provide two key benefits. First, they allow multiple heterogeneous sources of information as input, thereby identifying more accurately the relationships between multiple sources of data that jointly contribute to a specific structural property (Karpathy and Fei-Fei, 2015; Wang et al., 2018; Mikolov et al., 2013b). Second, deep neuron networks are extensible and can easily incorporate additional sources of information by attaching more neurons, network layers, or network branches (Wang et al., 2018). One can start with a model trained on the original components and re-train it using only the newly added parts or data sources (Erhan et al., 2010). This makes it easier for network operators to deploy, apply, or update neural network based models.

We propose DIP, a deep learning based framework to learn the structure of the Internet. DIP is a ten-layer neural network[1] that computes a low-dimensional vector representation for any node[2] in the Internet *given only its IP address and routable prefix*. DIP preserves both local and global structure: clustered nodes have similar representations and the distance between two representations approximates the hop count between the associated nodes.

We train our neural network using three heterogeneous data sets: hop count distances between Internet nodes, the 32 bits IP address and inter-domain routable prefix information for each node. A key insight to train DIP is to *first compute representations based on the IP and routing information*, thereby recovering structural information hidden in the IP values, and refine them using a distance-based optimization. As the size of the routable prefix varies by IP, we first normalize the IP and prefix data by representing an IPv4 address on 64, rather than, 32 bits. To capture structural

[1]To avoid confusion and unless explicitly stated otherwise, we use *network* or *Internet* to refer to the physical IP network and *neural network* to refer to the neural network we design to learn the structural properties of the physical network

[2]We use *node* or *(end-)host* to denote any computer connected to the Internet and assigned an IP address.

information encoded in the IP address value, we feed the eight bytes of the normalized IP sequentially at each of the first eight layers of the neural network. We then use the last two layers to get the hop count matrix and optimize the embedding distance prediction. With a trained DIP, we can estimate distance between *any* two Internet hosts as long as we have their IPs, even if they are not part of the training data.

Results on large real-world data sets of hop counts between thousands of IP addresses and 95 geographically distributed servers show that we can predict hop count distance between known hosts (*i.e.*, whose IP address were used in the training) with an absolute error of around 2 hops and over 30% better than a mean-based method. We infer the distance between unknown IPs (*i.e.*, not appearing in training data) with a small loss in accuracy compared to known IPs. In addition, the representations learned by DIP preserve the real-world clustering of the associated hosts. The accuracy of our model increases when we increase the training data set.

While our results are preliminary, they offer us a glimpse of the power of deep learning in recovering structural properties of the Internet from sparse data. DIP is the first framework that can estimate accurately the distance to any Internet host given only its IP address and routable prefix without any distance data.

2.1.2 Background and Related Work

What is structure? Many properties can make up the structure of the Internet: connectivity between IPs, routers, or networks; distance-based metrics such as hop count or latency; similarity-based metrics such as the set of one's neighbors in the connectivity graph; path-based properties such as the sequence of routers on a path. Here, we focus on two specific properties that define both the local and the global structure of a network: clustering of end-hosts and hop count based distance between end-hosts.

Network coordinate systems learn vector representations for participating nodes such that the position of the node in the embedding approximates its position in the Internet. Most coordinate systems build embeddings using a single source of structural data: latency measurements among nodes or to predetermined landmark servers (Dabek et al., 2004; Ng and Zhang, 2002; Costa et al.,

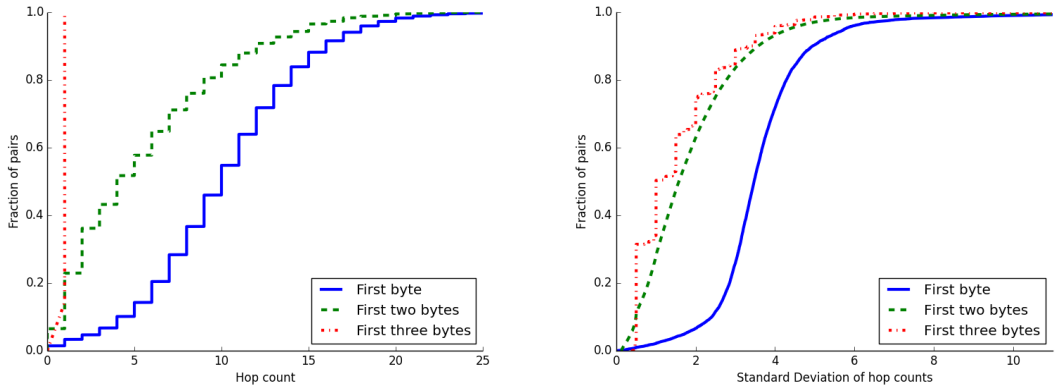


Figure 2.1: Cumulative distribution of (left) hop counts between pairs of host-server IPs that share the first, first two, or first three bytes, and (right) standard deviation of hop count distribution among groups of IPs sharing the first, first two, or first three bytes. The more similar two IPs are, the closer they are and the more similar their distances to the same third IP are.

2004; Pyxida; Zhao et al., 2011) or hop count information from passive traffic observations (Eriksson et al., 2008, 2009). Latency and hop count data is often sparse and cannot always be accurately embedded in metric spaces. To overcome these issues, several approaches use out-of-band information, such as location (Dabek et al., 2004) or routing (Eriksson et al., 2009) data, or perform active measurements (Eriksson et al., 2008) to impute the missing data and detect clusters or distances. Unlike them, we propose to train our embedding jointly using distances, routing information and host IP values, thereby learning hidden structural features encoded in a node's IPv4 address. With a trained model, we are able to embed and find the hop count to any IP, without the participation of its host.

Deep neural networks consist of multiple layers of interconnected neurons (LeCun et al., 2015). A neuron aggregates multiple input values using local weights and biases, applies an activation function, and produces one or more numerical values as output. Given a training task, one can define an objective function to evaluate the output of the entire neural network, *e.g.* , prediction error. Using gradient-based back-propagation algorithms to optimize the objective function (Kingma and Ba, 2014), neural networks automatically tune the weights and biases of each neuron to achieve a better performance.

2.1.3 Learning Network Representations

DIP learns an embedding model that accurately reflects the structure of the Internet, *i.e.*, preserves node clustering and distances between nodes. The goal of learning is to minimize the prediction error for the distance between any two nodes. The learned model is defined by the structure of the neural network and the final values for the weights and biases of each neuron. Next, we describe the data used in learning and how we construct the neural network.

2.1.3.1 Data sources

IP addresses and routing information. The IP address of a host provides a coarse indication of the location of the host in the Internet. To make routing scalable and fast, IP addresses are assigned hierarchically and divided into a network (or routable) part and a host (or local) part. The routable part, usually expressed by an integer representing the number of bits (also called prefix), tells routers how to route the packet through the core of the Internet towards the destination network. Intuitively, IPs with the same routable prefix share a path towards them through the Internet core and are more likely to be close to each other.

Hop counts. The hop count between two hosts represents the number of routers on the default path between hosts. We use hop count, rather than latency, to measure the distance between two hosts, as it can be easily extracted from the TTL value of a network packet (Jin et al., 2003), without active measurements. In Section 2.1.5, we discuss how to extend the model using latency measurements. Our hop count matrix is asymmetric and very sparse; it does not contain hop counts between all IPs.

2.1.3.2 IP transformation

The key idea of our work is to use both local (IPs and routing information) and global (hop counts) structural information to guide the embedding of network nodes. By utilizing deep learning for embedding, we can identify and use hidden features encoded in the IP address of a given node. We perform several transformations on the input, guided by observations on real network data.

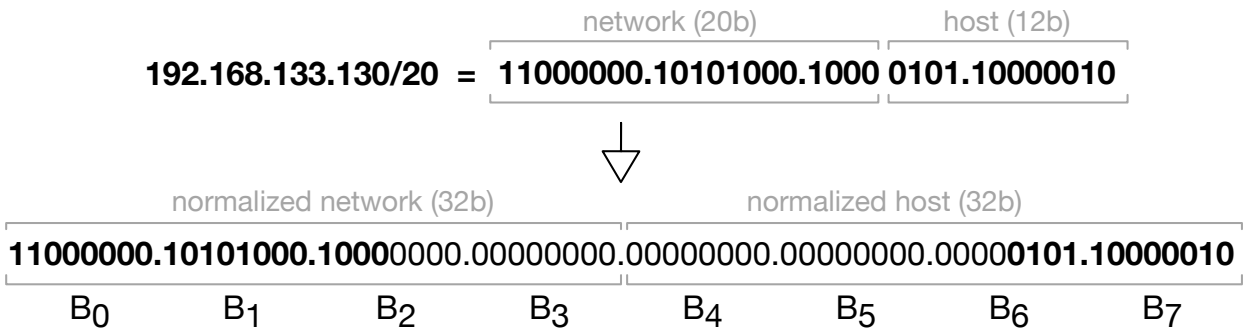


Figure 2.2: Generating a normalized IP address for 192.168.133.130/20.

IP normalization. Because the routable information is tied to an IP address, we combine the IP and prefix values when feeding them to the neural network. To keep the size of the input constant and independent on the prefix size, we generate a *normalized IP address* for each regular IP. The process of normalization is depicted in Figure 2.2. We divide each IP into the network and the host parts. We pad the end of the network part and the beginning of the host part with zero to obtain two four-byte values. We concatenate the values and get the eight bytes normalized IP. Further, for easier processing, we represent each byte of the input in one-hot vector format (256 dimensions), *e.g.* , a one and the rest are 0s, where the 1's position is the value of the byte (0 to 255).

Sequential feeding. IP addresses are assigned hierarchically and encode structural information of the network. To better understand how the hierarchical assignment affects node clustering, we perform two experiments on a data set of hop counts between 95 geographically distributed servers and ten million IP addresses of end hosts. Section 2.1.4.1 describes the data in more detail.

First, we group all pairs of host-server IPs according to whether they share (within the pair) the first byte, first two bytes, or first three bytes. We show the all-to-all hop counts between pairs in each of the three groups in Figure 2.1(left). The more similar two IP addresses are, the closer they are in terms of number of hops. Second, we group separately hosts and servers according to whether they share the first one, two, or three bytes and generate the hop count distribution for each pair of host-server groups that share the same prefix. We present the standard deviation for each pair in Figure 2.1(right). The smaller the standard deviation is, the more similar the distances are. This means that the more similar two IPs are, the more likely they have the same hop count to another node.

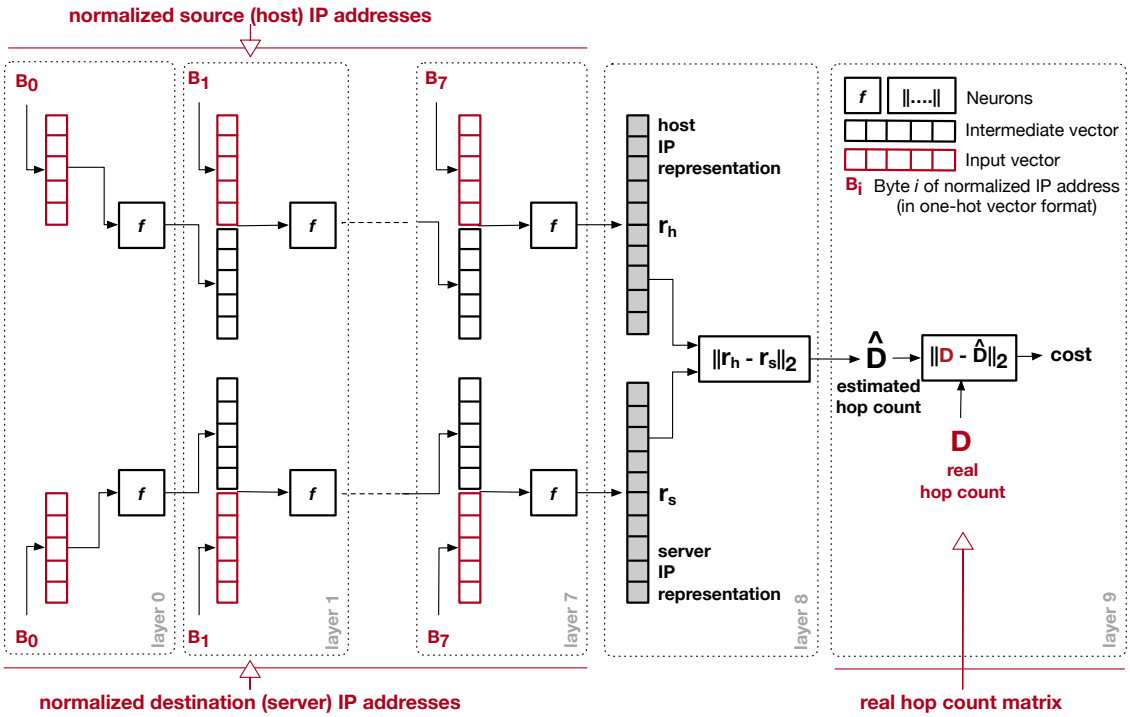


Figure 2.3: The neural network used for training our embedding model. The first eight layers receive the normalized IP addresses as input and compute the IP representations. The ninth layer estimates the hop count between two IP addresses and the tenth layer measures the model error. Elements in red are input. For simplicity we depict the input as one-dimensional vectors (one normalized IP); in reality, all inputs are matrices.

As shown in Figure 2.1, an IP address can help learn node representations that capture the network structure. The more bytes of an IP address we know, the better we can constrain the representation we assign to it. In addition, the more significant bytes of an IP address have a higher influence on the position of the associated host relative to other hosts. Therefore, the key is to capture the sequential correlation among the bytes of an IP address.

2.1.3.3 Network construction

Driven by the insight gained in the previous section, we develop DIP, a deep neural network that computes vector representations of network hosts based on their IP addresses and the hop counts to other hosts. The design of DIP, depicted in figure 2.3, is similar to that of a recurrent neural network (Mikolov et al., 2010), where new data is processed in the context provided by previous data (*e.g.* , like processing natural language). We explain the details below. Even though

the figure and our explanation refer to the input as one-hot vectors (*e.g.* , a normalized IP is represented as a vector of size 8x256=2,048), in reality the inputs are matrices (*i.e.*, the number of IP addresses times 2,048). Because our hop count data (see Section 2.1.4.1) is between separate end-hosts (sources) and servers (destinations) and because distances in the Internet are not always symmetric, we choose to feed the source and destination IPs separately in the neural network.

Intermediate IP representation. As mentioned earlier, to get the most out of the format and value of an IP address towards building a representative embedding for its host, we should treat each byte separately. The more significant bytes can provide a context for how to interpret the less significant bytes. Thus, we choose to input each byte of the normalized IP (a 256-dimension one-hot vector $B_{256\times1}^{i\in\{0,...,7\}}$) separately at each layer of the network. The input of layer i is the concatenation of byte i with the output of the previous layer (except for the first layer). This

$$Input = \begin{cases} i = 0 & B_{256\times1}^{i=0} \\ i \in \{1,...,7\} & concat(f_{d\times1}^{i-1}, B_{256\times1}^{i}) \end{cases} \tag{2.1}$$

where d is the dimension of the final IP representation and *concat* represents the vector concatenation operation.

At each layer, the activation function f is given by:

$$f^i = \begin{cases} i = 0 & softsign(w_{d\times256}^{i=0} \times B_{256\times1}^{i=0} + b_{d\times1}^{i=0}) \\ i \in \{1,...,7\} & softsign(w_{d\times(256+d)}^{i\in\{1,...,7\}} \times concat(B_{256\times1}^{i\in\{1,...,7\}}, f_{d\times1}^{i-1}) + b_{d\times1}^{i\in\{1,...,7\}}) \end{cases} \tag{2.2}$$

where $w_{d\times(256+d)}^{i\in\{0,...,7\}}$ are weights and $b_{d\times1}^{i\in\{0,...,7\}}$) are biases; the *softsign* function is $f(x) = \frac{1}{1+|x|}$. Initially, we assign random values to all weights and zeros to all biases. We employ *softsign* as the activation function for the ease of training, as *softsign* is more robust to saturation compared to other popular activation functions, such as *sigmoid* and *tanh*.

Intermediate distance estimation. We use the first eight layers of the neural network to process each of the eight bytes of the input normalized IP address. The output of the eighth layer is the intermediate vector representation for each IP address in the input data. We then use the last two layers to estimate how good the representation is. First we compute the estimated hop counts given by the current representation using an Euclidean distance. Given two matrices $H_{h\times d}$ and $S_{s\times d}$ storing the intermediate representations for the h hosts and s servers separately, the estimated distance matrix is:

$$Dist_{h\times s} = Euclidean(H_{h\times d}, S_{s\times d}) \tag{2.3}$$

Error reduction. Finally, we compare the estimated hop counts with the real hop counts matrix $D_{h\times s}$ to compute the cost as the mean difference of hop-counts. As the real hop count matrix is sparse, we compare only the valid entries:

$$Cost = \frac{\sum_{i=1}^{h}\sum_{j=1}^{s} W^{(i,j)}(||r_{d\times 1}^{H_{i\in\{1,..,h\}}} - r_{d\times 1}^{S_{j\in\{1,..,s\}}}|| - D^{(i,j)})}{count\,of\,non-zero\,D^{(i,j)}} \tag{2.4}$$

$D^{(i,j)}$ represents the value of the element at i^{th} row and j^{th} column in matrix D. $r_{d\times 1}^{H_{i\in\{1,..,h\}}}$ and $r_{d\times 1}^{S_{j\in\{1,..,s\}}}$ are rows in the matrices $H_{h\times d}$ and $S_{s\times d}$, and correspond to the representation of a host or server in the embedding space. W is a binary (0-1) matrix whose elements are defined as:

$$W^{(i,j)} = \begin{cases} 0 & D^{(i,j)} == 0 \\ 1 & D^{(i,j)} \neq 0 \end{cases} \tag{2.5}$$

To minimize the cost, we utilize the Adam algorithm, a gradient descent based back-propagation method (Kingma and Ba, 2014), which is able to automatically tune the learning rate during the training process.

2.1.4 Evaluation

2.1.4.1 Data and methodology

We use a large data set of network hop counts from the Ark project (Ark). The data contains hop count information from 95 geographically distributed servers to ten million IP addresses that cover all routable prefixes in the Internet. We use data collected by Ark during Jun 2015. For each IP in the data, we look up the routable prefix and normalize it using the steps in Section 2.1.3.2. Due to the cost of monitoring a large number of IPs, not all servers have hop counts for all ten million IPs. Our hop count matrix is incomplete and contains valid entries for only 29% of the pairs. We extract IP prefix information from Routeviews data (RouteViews) and use a default value of 24 for missing prefixes.

We build a prototype for DIP using TensorFlow. We train the neural network using several smaller data sets obtained by randomly sampling 1,000, 10,000, and 100,000 IPs from the original data and keeping only the hop counts to them. Sampling increases the sparsity of the data: less than 15% of the entries in the smaller data sets are valid. We also vary the number of servers and the dimensionality of the embedding space. Intuitively, having fewer IPs or servers may not provide sufficient constraints to learn accurate representations and lead to an underfit model. Increasing the number of dimensions can reveal more hidden features, invisible at lower dimensions, but may lead to overfitting. Each training session has 2,500 iterations, *i.e.*, passes through the neural network to update the weights and biases. We use a GPU server with four 3.5GHz quad-core Intel Xeon processors and 128GB of RAM. We generate testing sets by randomly sampling the original data and preserving the previously trained parameters for embedding arbitrary IP via its address (*e.g.* , the weights and biases of each byte/layer).

2.1.4.2 Embedding accuracy

Clustering. First, we assess how well DIP preserves the clustering of hosts in the original IP space. For this, we group all IPs first according to their routable prefix and then at random. For each cluster we compute an embedding similarity metric, defined as the ratio between the

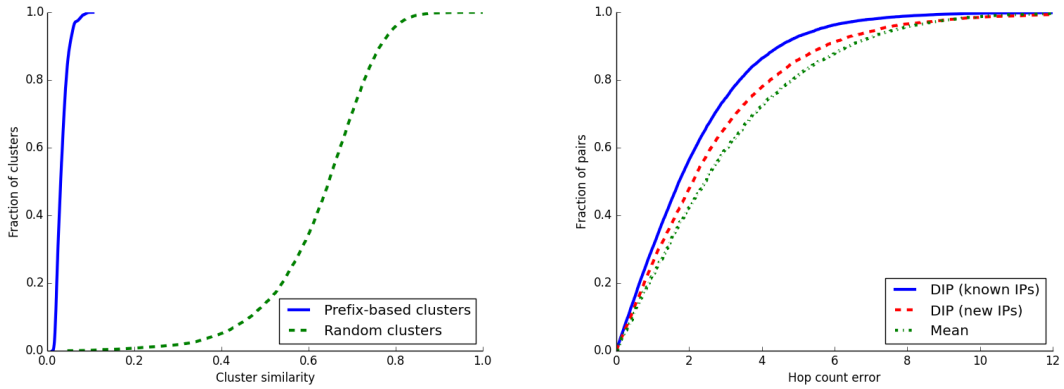


Figure 2.4: (left) Cumulative distribution of cluster similarity, computed using IP vector representations, for prefix-based and end-host random clusters; (right) Cumulative distributions of absolute distance estimation errors for DIP and *mean*. DIP representations preserve real-world prefix-based clustering and predict distances accurately.

average distance between all pairs of IP representations in the cluster and the maximum distance across all clusters. The lower the similarity value, the closer to each other the IPs of a cluster are in the embedding space. Figure 2.4(left) shows the similarity distribution for prefix-based and random clusters. Each IP representations is a 140-dimensional vector and computed after training the network using 10,000 IP addresses and 95 servers. Our embedding preserves the clustering of the original IP space well.

Distance prediction. To assess the quality of distance prediction, we first look at previous embedding mechanisms. Network coordinate approaches (Dabek et al., 2004; Ng and Zhang, 2002) are not directly comparable as they embed latencies between strategically chosen pairs of nodes, while we rely on hop count information from passively observed traffic. Eriksson *et al.* (Eriksson et al., 2009) propose a matrix factorization based algorithm to predict hop count information but first build baseline representations of the monitoring servers using an all-to-all hop count matrix. We lack complete hop count information among servers and build our embedding directly from incomplete server-to-host distances. Therefore, we compare against a *mean estimation approach*, where we predict a host-to-server distance as the mean of all valid distances to the same server.

We look at how well our embedding estimates the hop count value between a host and a server. We first consider only the IP addresses used in the training process (*i.e.*, *known IPs*). For this, we train a model using 90% of all host-server pairs and use the remaining 10% for testing. Fig-

	DIP		Mean	
	known IPs	**new IPs**	**known IPs**	**new IPs**
Number of IPs				
1,000	2.16 (1.86)	2.89 (2.38)	2.99 (2.44)	3.27 (2.51)
10,000	2.15 (1.79)	2.68 (2.34)	3.00 (2.40)	3.04 (2.43)
100,000	2.06 (1.76)	2.29 (2.00)	2.98 (2.40)	2.97 (2.40)
Number of servers				
12	2.79 (2.25)	3.03 (2.47)	3.21 (2.54)	3.28 (2.62)
24	2.39(2.14)	2.60 (2.25)	2.90 (2.36)	2.97 (2.42)
48	2.34 (2.05)	2.75 (2.27)	2.99 (2.39)	3.02 (2.40)
95	2.15 (1.79)	2.68 (2.34)	3.00 (2.40)	3.04 (2.43)
Embedding dimension				
110	2.28 (1.93)	2.80 (2.39)	3.00 (2.42)	3.04 (2.43)
140	2.15 (1.79)	2.68 (2.34)	3.00 (2.42)	3.04 (2.43)
170	2.19 (1.86)	2.72 (2.33)	3.00 (2.42)	3.04 (2.43)

Table 2.1: Absolute mean error (standard deviation between brackets) of distance prediction of DIP and *mean*, for both known and new IPs, when varying the number of IPs, the number of servers and the embedding dimension. The default values are 10,000 IPs, 95 servers, and 140 dimensions.

ure 2.4(right) compares the absolute error between estimated and real hop count for DIP and *mean* for the 10,000 IPs data set on 140 dimensions. DIP predicts distances with a mean absolute error of around two hops (23% mean relative error) and reduces the error of the *mean* estimation by almost 30%. Table 2.1 presents the average absolute error and standard deviation for hop count estimation for embeddings trained with different number of hosts, servers, or dimensions. As expected, increasing the the number of IPs, servers, or the dimensionality reduces the absolute error. We also trained models with parameters outside the ranges presented in the table but found no improvement.

New IPs. An important feature of DIP is its ability to impute hop count values to arbitrary nodes based on their IP address. *New IPs* are IP addresses not used in the training process and that DIP has never seen before. Figure 2.4 (right) and Table 2.1 show that DIP approximates distance to new IPs with high accuracy. The distance prediction error is only around half a hop more than that for known IPs. To the best of our knowledge, DIP is the first framework to predict hop counts to arbitrary hosts based only on the value of their IP address and routable prefix and without any other domain knowledge.

2.1.5 Discussion

We discuss several future applications and directions of using deep learning to understand and capture the Internet structure.

Extensions. An important benefit of using neural networks for learning the structure of the Internet is that they can be extended easily for other data sources. Similarly to previous embedding approaches (Dabek et al., 2004; Ng and Zhang, 2002), we could use latency measurements instead of, or in addition to, hop counts. This would require simply changing the cost estimation part of the neural network (last two layers). While gathering latency measurements is expensive as it introduces traffic into the network, the ability of our approach to work with sparse data can limit the cost necessary to obtain the measurements.

Furthermore, AS membership information could help find more accurate representations as many ASes cover limited areas in the network and provide a coarse indication of locality (Eriksson et al., 2009). To add AS membership information, we could either extend the shape of our input vectors (by adding two bytes for AS number) or adapt the cost estimation layers to use AS data in estimating error, similarly to Eriksson *et al.* (Eriksson et al., 2009).

Applications. Building a model that accurately predicts structural properties of the Internet has several applications. Knowing the distance to remote IPs can help selecting a load balancing server or an overlay peer more efficiently and without having to perform expensive measurements. Understanding how nodes are clustered can make the transmission of video or large files faster by using close-by CDN nodes. DIP can be a passive defense mechanism against IP spoofing attacks, where malicious users change the source IP of attack packets to avoid identification and subvert authentication. By comparing the predicted distance according to the spoofed source IP to the real distance (extracted from a packet's TTL field), one could verify whether the packet is spoofed or not (Jin et al., 2003).

Limitations and future work. Our current approach uses structural information embedded in the value of IP addresses, routing data, and distances between nodes, but does not consider the actual physical links between nodes on the Internet (*i.e.*, the Internet physical topology). Adding topology would further constrain the embedding, since it is well known that the Inter-

net is not a metric space and latency or hop count distances cannot always be embedded in metric spaces (Dabek et al., 2004). We plan to extend our framework using graph embedding algorithms to take advantage of physical topology information (Wang et al., 2016a).

While our preliminary experiments focused on accuracy, the performance of building an embedding model is equally important. Training a model with 100,000 addresses and 95 servers on our 16-core GPU server takes a few hours, indicating that we may need to train models incrementally when resources are constrained (Bruzzone and Prieto, 1999). For example, in a live deployment, we envision reconstructing our model every few days to capture the changes in topology triggered by the dynamic Internet. We are currently studying ways to incrementally add or update models without rebuilding from scratch.

Because our data is sparse, not even the best embedding may be able to recover all structural properties. While we show that our results are reasonably accurate, even when we have less than 15% of all distances available, getting more data is clearly helpful (Eriksson et al., 2009). We plan to use active monitoring techniques (*e.g.* , *traceroute*) to collect more information for the training phase. Knowing the IPs and connectivity of routers in the network would make the training data set richer and constrain the representation of end-hosts further.

2.1.6 Conclusions

We used deep learning to learn vector representations for nodes in the Internet based on their IP address, routing information, and a sparse hop count distance matrix. Deep learning helps uncover hidden features in the input data and recover structural properties of the Internet, such as node clusters or distances between nodes. Our experiments on a large real-world data set show that our embeddings can recover most distances, even to arbitrary hosts, with two hops absolute error, even when the training data is sparse.

2.2 Learning IP Maps for Network Spoofing Detection

2.2.1 Introduction

Spoofing the source IP address of network packets is a mechanism frequently used in denial-of-service attacks (Chen et al., 2008; spoofing-ps). IP spoofing can both hide the true source of the attack, thereby subverting IP-based firewalls and authentication mechanisms, and force legitimate hosts to redirect malicious traffic, thereby amplifying the effects of the attack (github). In 2018 alone, IP spoofing was the vehicle for several high-profile, terabyte-size DDoS attacks (cloudflare; github; arbor).

There are two types of defenses against spoofing attacks. Active methods encrypt connections (Frankel and Krishnan, 2011), probe suspicious IPs (Feng et al., 2005), or mark legitimate packets with unique identifiers (Bremler-Barr and Levy, 2005; Yaar et al., 2006). They come at the expense of bandwidth, latency, or computation overhead (Shue et al., 2005). Passive (or map-based) defenses construct explicit offline maps between IPs and immutable structural network properties, such as paths (Park and Lee, 2001; Duan et al., 2008), neighboring networks (Ferguson and Senie, 2000; Killalea, 2000; Baker and Savola, 2004), or hop counts (Wang et al., 2007) and filter packets whose header information does not match the maps.

Passive map-based defenses have little detection overhead and are less intrusive than active defenses, but pose a significant construction cost. Measuring the network properties associated with each IP is a long and tedious process that requires intrusively probing other hosts, querying routing tables, or passively waiting to receive sufficient traffic (Jin et al.). In addition, as network properties are different for every vantage point, maps computed for a location cannot be easily transferred to a different location and must be recomputed.

We propose to *learn a structural model (or embedding) of the Internet and use the model to predict, rather than explicitly compute, IP maps*. Learning instead of measuring network properties could significantly reduce the cost of achieving complete host IP maps while making them more general. First, training a network embedding requires only a small set of ground truth information, such as distances between IPs (*e.g.* , hop counts, latencies) (Dabek et al., 2004; Lumezanu et al.,

2007; Eriksson et al., 2009). This reduces the amount of data necessary *a priori* for constructing a good map, thereby reducing the construction cost. Second, network embeddings are general and can be used to predict associations between *any* IP and its local network properties, even when the IP was not part of the training.

We study the feasibility of using network embeddings to learn IP maps for spoofing detection. Towards this goal, we present and build a learning-based spoofing detector that combines DIP, our previous deep learning based network embedding algorithm (Li et al., 2018a,b) and hop count filtering, and a popular host-based spoofing detection mechanism (Jin et al.). Hop count filtering compares the hop count information of packets arriving at a target server to the known hop count value from the packets' IP source to the target. It considers a packet spoofed if the two values do not match. Rather than build an explicit map of known IP-to-hop-count associations, we use DIP to learn a network embedding that preserves hop count distance between IPs. Using the embedding, we predict with high accuracy the hop count between any two IPs and identify when a packet is spoofed.

We analyze the coverage, accuracy, and cost of our learning-based detector. We show that learning, instead of explicitly computing IP maps, dramatically increases the detection coverage over spoofed sources and targets. Our learning-based detector needs hop count information from only around 1,000 IPs to several targets to build a model that can help detect spoofing from most of the Internet. Although, not as accurate as the original hop count filtering in detecting packets spoofed with previously known IPs (*i.e.*, to which the hop count is known), ours is the first map-based detector to identify spoofing with unknown IPs (*i.e.*, for which the hop count to the target is not known). It can also be used to detect spoofing to new targets, not part of the training process, with only a small loss in accuracy. In addition, learning an embedding is fast: even with almost three million hop counts from 100,000 IPs, it takes less than an hour to learn an embedding that can predict maps for any IP address.

This work brings two contributions. First, we introduce a framework to help *any* Internet server detect packets spoofed with *any* IP address without any additional measurements to that IP. This represents a major shift from previous map-based spoofing detection mechanisms, who are either restricted to specific targets or to detect packets spoofed with known IPs. Although our method

does not yet achieve by itself the accuracy necessary for a real-world deployment, it nevertheless shows that learning, rather than measuring, network properties could be an important piece in the spoofing detection puzzle.

Second, our work explores the impact of deep learning in the network and security operations decision making. As AI moves beyond just being the word *du jour* and increasingly becomes an integral part of the network (dls), it is important to understand the trade-offs between its cost and benefits (Sommer and Paxson, 2010). It is precisely such a trade-off that we analyze in our work: while learning IP maps for spoofing detection can ease the task of administrators and help build complete maps much faster, it may decrease the detection accuracy. Understanding this balance can lead to better and more protected networks.

2.2.2 Towards learned maps for spoofing detection

Map-based spoofing defense mechanisms associate IP addresses with immutable network properties that are difficult to modify by attackers. Packets that carry information (*i.e.*, in headers) not matching the map are dropped. Unlike active spoofing defense methods that encrypt connections (Frankel and Krishnan, 2011) or probe suspicious IPs (Feng et al., 2005), map-based approaches are less intrusive and do not introduce additional traffic. To be effective, map-based detection must provide *coverage*: given a random attacker, protect any target against attack packets spoofed with any IP address.

Network maps. Network-based maps reside on AS border routers and pair IPs to the incoming network interface (Baker and Savola, 2004; Duan et al., 2008) or with keys inserted in the packet by an upstream trusted party (Bremler-Barr and Levy, 2005). Such maps have the potential to provide complete coverage, if installed pervasively by *all* ASes. However, their deployment has been slow. Recent measurements performed by CAIDA (spoofing-state) show that around 25% of all ASes allow spoofed traffic to traverse them. Until all networks deploy network-based detection, the Internet is susceptible to spoofing. A likely cause for the incomplete coverage is the misaligned economic incentives (Lone et al., 2017): the cost to deploy network maps is high and supported by each AS, while the benefit is incremental and spread to the entire Internet.

Host maps. Host-based detection aligns the incentives by pushing the onus of detection towards the edge of the network. Hosts[3], rather than routers, build and maintain IP maps and benefit immediately from them. Most notably, several solutions build maps between IPs and hop count values to the target, relying on hop count data as a measure of the network topology (Wang et al., 2007; Jin et al.). Hop counts can be easily derived from the IP TTL field, whose value depends on the network infrastructure (*i.e.*, every router decrements the value before forwarding the packet) and cannot be easily forged by an attacker.

By pushing map computation to the edge, host-based maps decentralize the spoofing detection process and may see decreased coverage. First, their *detection is limited* to packets spoofed with IPs to which the target knows the hop count. They fail in detecting packets spoofed with IPs unknown[4] to the target. As attackers rely on surprise and obfuscation, they often use random spoofed addresses that can be from anywhere, even from non-routable ranges (cloudflare). Increasing coverage by building complete maps is difficult. Passively collecting hop counts from incoming traffic is not likely to provide a large coverage (Jin et al.), while actively probing all IPs is expensive, intrusive, and time-consuming (Ark). Second, host maps are *location specific*. A map associating IPs with hop counts to a target is only useful for that target and cannot be transferred to other hosts, even if they are part of the same network.

Learned host maps. We propose to *learn* IP-to-hop-count mappings, instead of extracting them from incoming packets. A natural solution is to first learn representations of IPs in a vector space and then estimate the distance (*i.e.*, hop count) between IPs as the distance between their representations. Network embedding methods can easily compute IP representations in Euclidean spaces from a limited set of latency or hop count information (Dabek et al., 2004; Ng and Zhang, 2002; Costa et al., 2004; Pyxida; Eriksson et al., 2009). However, they are limited to the hosts part of the training set and cannot generate representations or hop counts for other hosts. This means that they cannot be used to estimate hop counts for, and thereby detect packets spoofed with, IP

[3]Throughout the paper, we interchangeably refer to the server that deploys host-based maps as host, end-host, target, destination, or victim.

[4]We consider an IP address *unknown* to a target if the target does not know the hop count to it. Existing host-based maps cannot detect packets spoofed with an unknown IP address.

addresses unknown to a target.

The emergence of learning frameworks that preserve structural network properties such as distances or clustering among nodes offers an alternative to build explicit host-based IP maps from passive or active measurements (Li et al., 2018a; Wang et al., 2016b). Deep learning based embeddings use several data sets, *i.e.*, distances, routing information and host IP values, to learn IP representations. Because they learn hidden structural features encoded in a node's IP address, they could generate representations for any node, given only its IP address (Li et al., 2018a). In addition, deep learning algorithms are easily extensible and adaptable when new or updated data is available.

Figure 2.5 presents a visual representation of the benefits of learning-based spoofing detection. Unlike network maps, which detect only spoofed packets traversing protected ASes (left diagram), or explicit host maps, which detect only packets spoofed with IP sources known by the host (middle diagram), learned host maps can detect packets spoofed with unknown IPs (right diagram). The host uses a learned structural model of the Internet to estimate a specific network property, *e.g.* hop count information, associated with the source IP of an incoming packet and decide whether the packet is spoofed or not. In the next section, we describe a learning-based spoofing detector combining hop count filtering with the DIP network embedding framework.

2.2.3 Learning-based spoofing detection

To illustrate the benefits of a learning-based spoofing detector, we propose a prototype detection framework based on the hop count filtering method first introduced by Wang *et al.* (Jin et al.). Our prototype consists of two main components: an offline IP map learning module (to learn IP representations and estimate hop counts between IPs) and a spoofing detector (to detect spoofed traffic). We describe both modules next and study their effectiveness in the next section.

2.2.3.1 IP map learning

In a previous work, we introduced DIP, a deep learning framework that uses hop count information between network host to learn an embedding of the Internet (Li et al., 2018a). Using a

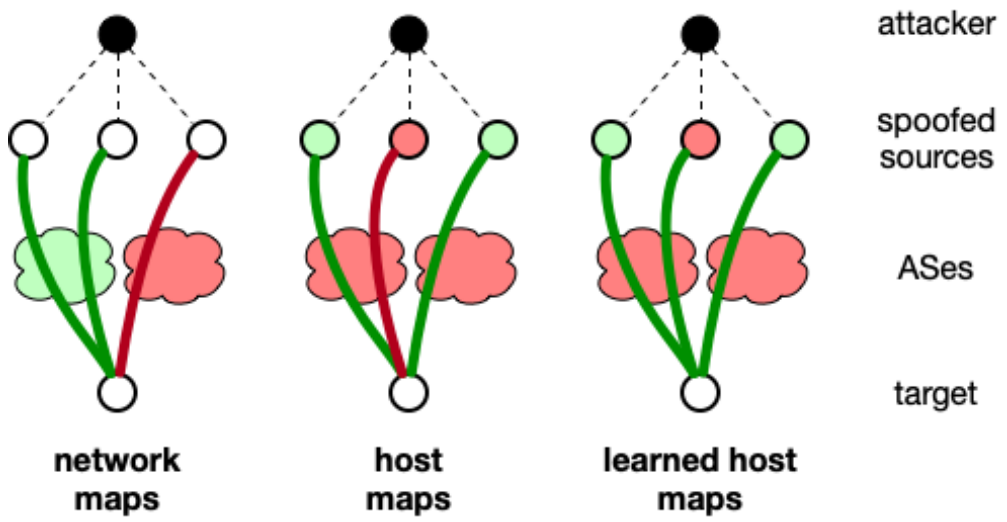


Figure 2.5: Map-based spoofing detection. Network maps detect only spoofed packets traversing protected ASes (colored in green). Host maps detect only packets spoofed with IPs present in the map (also colored in green). Learned host maps have the ability to detect all packets because they learn missing map entries. The paths in the diagrams indicate the apparent source of the packet (the spoofed source). In reality, all packets originate from the attacker.

small data set of IP addresses, their prefix information, and hop counts among them, DIP computes a vector representation of each IP in a high-dimensional Euclidean space. The algorithm ensures that representations reflect hop counts between hosts: the Euclidean distance between two representations estimates the hop count between the associated IPs. We briefly describe DIP below and refer the reader to the work of Li et al. (2018a,b) for more details.

The training data for DIP includes the IP addresses and hop counts among them. The IP addresses are hierarchical sequences which contain two parts, i.e. network (routable, routing prefix) and host (local) part. For example, for a 32-bit IPv4 address 192.167.2.17/24, the number 24 means the first 24 bits form the routable prefix, which can indicate the identification of the subnet. The rest 8 bits will be the local information for this IP address within the subnet. The hop count between two IPs represents the total number of intermediate devices (e.g. routers) that a data package needs to pass. We use it as the distance between two IPs and the training objective is to estimate the hop count between two IPs. The utilized hop count matrix is actually very sparse and asymmetric. During training, we will not pre-filling any value to the matrix to avoid introducing noises. We will measure the loss only when the hop count information exists.

The DIP is designed to exploit the above mentioned information efficiently with a variant of recurrent neural networks to capture the structure encoded in the value of an IP address (*i.e.*, network part and host part provide an implicit hierarchy of IPs). According to the definition of the network and host part, we can find it is actually more significant to use the network part expressing the structural position. However, for IP addresses, their network parts can have different sizes. The network part's size can be any value from 0 to 32 (IPv4). Similarly, the number of bits contained in the host part varies. Hence, before feeding in the IP addresses, we will normalize them by: 1) Divide the 32 bits into network and host parts; 2) For the n bits in network parts, we pad in $32-n$ 0s at the end; 3) For the $32-n$ bits in the host parts, we pad in n 0s at the beginning. Then, we can get two four-byte normalized values.

With the two four-byte values, we can convert their each byte's numerical value to a one-hot vector (256 dimensions). The reason to use one-hot vector is we think there is no relationship like natural ordering or neighboring between different values of byte. For example, there are three hosts with the second byte equalling 156, 192, 15. We cannot easily say that the first two hosts are nearer to each other compared to the third because the difference between 156 and 192 is smaller. So we would like to regard the byte value as categorical data and use one-hot vector to represent.

By two experiments result shown in Figure 1 of Li et al. (2018a), we find the more bytes for an IP address, the better we can constrain the representation of the IP. This shows the sequential information contained by the IP addresses. So, in each layer of DIP, we feed the one-hot vector of the eight bytes from the normalized IP separately and use the structure similar to Recurrent Neural Network to capture the sequential information. Suppose the 8 bytes are $B_i, i \in 0,...,7$, where each B is actually converted to a one-hot vector of 256 dimensions. The input for the first layer of DIP will be B_0. Via the formula:

$$h_0 = activate(W_0 \times B_0 + bias_0) \tag{2.6}$$

we can get the hidden output of the first layer. The activate function can be either $softsign$, $tanh$ or $sigmoid$. In our experiments, we use $softsign$ for an easy and efficient training.

Then, for the following layers, the hidden output of the last layer will participate into the input

of the current layer.

$$h_i = activate(W_i \times [B_i, h_{i-1}] + bias_i), i \in 1, .., 7 \tag{2.7}$$

The $[B_i, h_{i-1}]$ means to concatenate the current input and the last hidden state. Then, by the h_7, we can add another feedforward layer to convert the h_7 to h_8 and estimate the hop count by:

$$hop - count = Euclidean(h_{8,source}, h_{8,destination}), \tag{2.8}$$

which utilizes the source and destination IPs' hidden representation for estimation. During learning, the trainable parameters include the W_i, $bias_i$ where $i \in 0, ..., 7$ and the final layer's parameters to convert h_7 to h_8. Since we think the first 4 bytes, which is the normalized network part of the IP address, are more significant for structural information. We will assign a higher initial value for the W_i when $i \in 0, ..., 3$ and a smaller value for the W_i when $i \in 4, .., 7$. This will enhance the influence of the first four layers and weaken the influence of the second four layers. The learning will be supervised: at each iteration of the algorithm, the representations are refined towards minimizing the error between estimated hop counts (computed from the Euclidean distance between two representations) and real hop counts. All the previous mentioned formulas and steps are simplified version for the processes described in Li et al. (2018a). If you need more detailed information, please check the Li et al. (2018a).

The advantage of DIP compared to traditional network embedding approaches is that it incorporates structural information encoded in the value of an IP address towards learning their position in a vector space. Learning from the value of an IP address is critical for our purpose. The learned model can be used to generate representations for *any IP*, even if it was not part of the training set or we do not have distance information to it. This, in turn, increases the coverage of our spoofing detection. We are not limited to the information from IP maps anymore; missing associations between IPs and hop counts can be easily estimated from the learned model. In addition, embedding models are general: they can be transferred to and immediately used by other hosts, without additional retraining, thereby reducing the overall cost of achieving global detection.

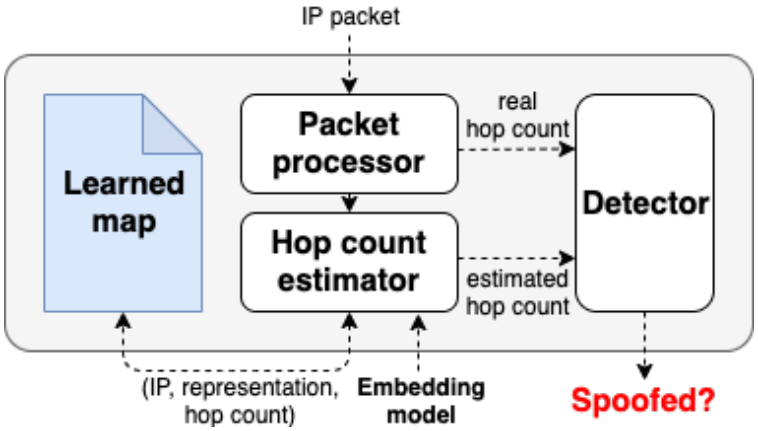


Figure 2.6: The learning-based spoofing detector uses an embedding of the Internet to estimate hop count information to any IP address and detect when the IP is used as the spoofed source of an attack packet.

Notwithstanding its advantages, DIP suffers from the same limitation as previous network embedding methods (Dabek et al., 2004; Ng and Zhang, 2002; Costa et al., 2004; Pyxida; Eriksson et al., 2009): Internet hop counts do not form a metric space (*e.g.* , they violate the triangle inequality (Lumezanu et al., 2007)) and cannot be embedded accurately into an Euclidean space. DIP is able to predict distances between Internet nodes with a mean absolute error of 2.06 hops, after being trained on distances between around 15,000 pairs of nodes. The error increase is small (from 2.06 hops to 2.29 hops) when we consider distances to IPs not used in the training, and therefore unknown to the targets. Detecting spoofed packets based on imprecise hop count information may lead to incorrect decisions. To minimize the number of missed spoofed packets, we must allow a margin of error when comparing the estimated and real hop counts. In Section 2.2.4, we show that spoofing detection still works when the hop count estimations are not exact and discuss how to select the detection threshold.

2.2.3.2 Spoofing detector

The spoofing detector is deployed on a potential target and consists of the learned map, packet processor, hop count extractor, and detector (Figure 2.6). The packet processor extracts the source IP and TTL values from an incoming packet. It then computes the hop count information from the TTL, using the algorithm described by Wang *et al.* (Wang et al., 2007). The hop count estimator

Method	Description
network maps	Complete coverage if all ASes deploy network maps; currently 25% of ASes are unprotected (spoofing-state); cost of building maps is high (Lone et al., 2017).
host maps	Coverage limited to the IPs with known hop count to the host: potentially covering packets spoofed with 95% of all IPs (Jin et al.), if map contains *all* IP addresses; maps cannot be transferred and need to be explicitly computed by each host; cost of building maps is high (Jin et al.; Ark).
learned host maps	Limited by the accuracy of embedding; covering packets spoofed with around 70% of all IPs, even when the IP is unknown to the host; embedding models can be transferred and do not need to retrained; cost of building maps is low.

Table 2.2: Map-based spoofing detection methods.

uses an existing embedding model (*i.e.*, built offline using IP and hop count data from legitimate packets) to compute a representation for the packet's source IP and estimate the hop count to the target. The detector compares the real and estimated hop counts to the target. If the difference between them is greater that a pre-specified threshold, we consider the packet spoofed and raise an alert. To save future computation, all estimated hop counts and IP representations are saved into a cached learned map. When packets from an IP present in the map arrive at the target, we use information from the map without recomputing the hop count value.

2.2.4 Evaluation

We evaluate the feasibility of learning-based spoofing detection as follows. First, we analyze what coverage we can achieve compared to detectors based on exact maps, such as hop count filtering. Second, we simulate spoofing and measure how well we detect spoofed packets under various scenarios. Finally, we look at the cost of building a detector. Table 2.2 contains a summary of our findings.

Data. We use a large data set of network hop counts collected by the Ark project (Ark) during June 2015. The data contains hop count information from 96 geographically distributed servers to ten million IP addresses that cover all routable prefixes in the Internet. Because monitoring a

large number of IPs is costly, not all servers have hop counts for all ten million IPs. Our hop count matrix is incomplete and contains entries for only 29% of the pairs. Furthermore, due to privacy issues, the last eight bits of each IP were zeroed out after collection. Although this makes the set of possible spoofed sources more general, it also helps us learn more robust models, as there are likely more servers monitoring the same /24 prefix than the same single IP address, *i.e.*, the number of hop counts per IP is higher.

It is possible that the hop counts collected by Ark are to routers on the path, rather than the end host itself. This can happen when routers, and not the target, reply to the monitoring packets. As our detection algorithm is intended for the edge of the network, training with router IPs and hop counts may bias the results. It is difficult to determine precisely which response comes from a router. To minimize the potential bias, we use a simple heuristic based on the fact that many router IPs are not part of advertised BGP prefixes. Using this heuristic, we filter out around 1.2% of the IPs in our data.

Methodology. We build a prototype spoofing detector using TensorFlow and Python, reimplementing the learning in DIP (Li et al., 2018a) and hop count filtering (Jin et al.). We train the neural network offline using several smaller data sets obtained by randomly sampling 1,000, 10,000, and 100,000 IPs from the original data and keeping only the hop counts to them. Sampling increases the sparsity of the data: less than 15% of the entries in the smaller data sets are valid. We train our spoofing detector for 1,000 iterations on a server with four 3.5GHz quad-core Intel Xeon processors and 128GB of RAM.

Assumptions. We assume a single attacker, randomly placed in the Internet, with no knowledge about network structure around the spoofing target or the spoofed IP. This means that the attacker cannot use information about hop count to the target to avoid detection. We also assume that the hop count information is static and does not change. In Section 2.2.5, we discuss what happens when we relax these assumptions.

Goals and non-goals. Our main goal is to evaluate the feasibility of learned maps for detecting spoofed network packets and to inform future decisions of security operators or researchers. We use an existing embedding algorithm, DIP (Li et al., 2018a) to learn hop count information between

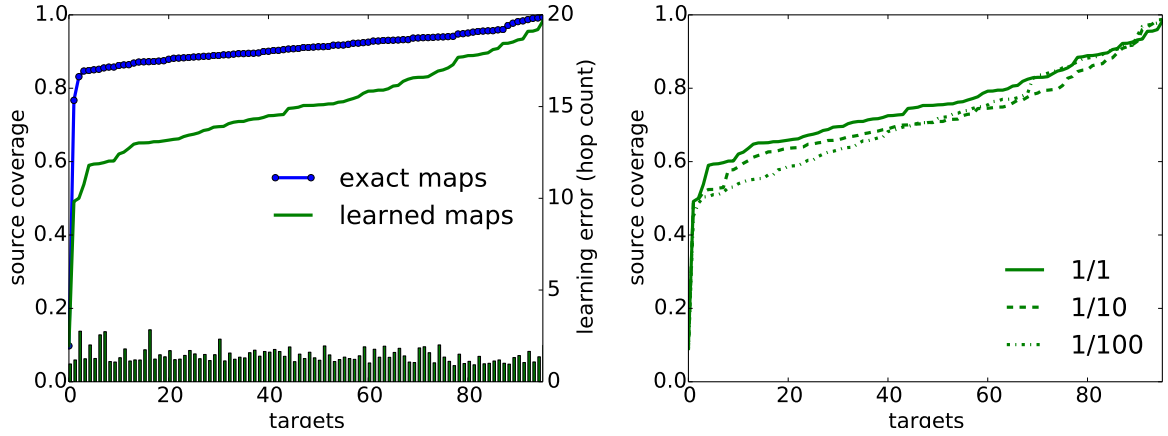


Figure 2.7: Coverage for (left) exact and learned maps for 1,000 source IP addresses, and (right) learned maps for the same sources used in training (labeled "1/1"), ten times as many sources ("1/10"), and a hundred times as many sources ("1/100"). The bars represent the error of a learning-based spoofing detector, for each target, in hop counts. Learning-based spoofing detectors adapt well to new sources with little loss of coverage.

IPs, and study its interaction with hop count filtering (Jin et al.). A non-goal is to evaluate the DIP embedding accuracy. For a detailed study, we refer the reader to the DIP paper (Li et al., 2018a), which analyzes the embedding accuracy when varying various learning parameters and in contrast to other distance prediction mechanisms.

2.2.4.1 Coverage

The limited range of hop count values (*i.e.*, 0 to 255) means that no hop-count-based spoofing detector can achieve perfect detection. To understand the limits of detection, we define the *coverage* of each target as the expected fraction of source IPs that can be unambiguously identified when spoofed by a random attacker. In other words, the coverage represents the probability that a packet spoofed with a random IP by a random attacker is detected by the target.

Exact maps. Before studying learning-based maps, we analyze the boundaries of what exact hop count based maps can achieve. Hop count filtering (Jin et al.) identifies packets as spoofed when the packet hop count does not match the known hop count associated with the source address. If the attacker spoofs a packet with an IP address that has the same hop count to the target as the

attacker IP, hop count filtering cannot detect the attack. To compute the coverage of exact maps, we repeatedly select a source IP at random as the attacker for each target, calculate the fraction of IPs that share the same hop count to the target as the attacker and subtract the value from 1. Figure 2.7(left) shows the distribution of coverage for all targets in our data set. We consider 1,000 IPs; results for other data sets are similar. Focus only on the line labeled *exact maps* for now. Most targets have a coverage of at least 0.8.

An important observation is that exact maps are not necessarily limited to detecting traffic spoofed with IPs that have communicated with the target in the past. They are efficient in detecting attacks spoofed with any IP, as long as the target knows the hop count to the IP. Oftentimes, IPs that are part of the same prefix, especially for smaller prefixes, share the same hop count to a target. Our data set already contains at most one IP per /24 prefix to ensure it does not underestimate the performance of exact maps.

Learned maps. As described in Section 2.2.3, learning maps introduces errors in estimating hop count values. Any learning-based spoofing detector needs to account for those errors in identifying spoofing. First, for each target, we compute the average error between real and estimated hop counts from all sources to the target. We use hop counts from 1,000 IPs to the 96 servers to learn the embedding. Then, similarly to above, we select an IP at random from the 1,000 IPs in the training set as the attacker and compute the fraction of IPs whose hop count is within the estimation error of the hop count from attacker to target. We subtract this value from 1 to obtain the coverage for the learned map of each target. This captures the probability that learned maps detect packets spoofed with a random IP from a random attacker. Note that here we select an IP *from the training set* as the attacker to compute source coverage for known IPs and compare with exact maps. Below, we re-do the analysis for unknown IPs, that are not from the training set. Figure 2.7(left) shows that the coverage of learned maps is lower than that of exact maps when considering the same set of source IPs. On the average, a target could detect spoofed packets only 70% of the time, compared to 90% of the time with exact maps.

The power of learned maps lies in estimating hop counts to IPs that are *not* in the training set. We increase the number of source IPs to which we estimate hop counts to 10,000 and 100,000 (10, respectively 100 times more than the training set) and show the coverage in Figure 2.7(right). Even

when we increase the number of IPs 10- or 100-fold (lines labeled "1/10", respectively "1/100") compared to the training set, the coverage does not decrease much. In comparison, the coverage of exact maps for any unknown IPs *is 0*. Thus, learned maps offer an immense benefit when protecting against spoofing carried with IPs not known to the target.

Summary. IP-to-hop-count maps learned via deep learning embedding are less accurate than exact maps extracted from incoming traffic, when evaluated on IPs to which the host knows the hop count. However, learned maps dramatically outperform exact maps on unknown IPs.

2.2.4.2 Accuracy

We define the sensitivity of the detector as the fraction of detected spoofed packets out of all spoofed packets, and the specificity as the fraction of correctly identified legitimate packets out of all legitimate packets. Sensitivity (also known as recall or true positive rate) represents the probability of detecting a spoofed packet and specificity (also known as selectivity or true negative rate) captures the probability of not raising a false alert. Unlike coverage, which gives a measure of how well the detector can do for a randomly spoofed IP at any target, sensitivity and specificity measure the accuracy of the detector in a realistic scenario, given both legitimate and spoofed packets.

We set up the experiment as follows. We use the 10,000 IP data set for both training and detection. We simulate packets arriving from each of the 10,000 IPs at random and introduce spoofing with an average rate of 0.01 (one spoofed packet for every 100 packets). For each spoofed packet, we generate a fake hop count value at random from a normal distribution with a mean of 15 and standard deviation of 5. We chose these values as they match the distribution of hop counts in our data set. We measure sensitivity and specificity as averages across all targets in a specific experiment.

Varying threshold. Figure 2.8(left) shows the specificity and sensitivity as we vary the detection threshold. For a threshold of two hops, which is the average training estimation error, we obtain an overall sensitivity of around 0.75, which is consistent with the expected coverage from Figure 2.7(left). However, a high sensitivity also results in a low specificity: while we detect most

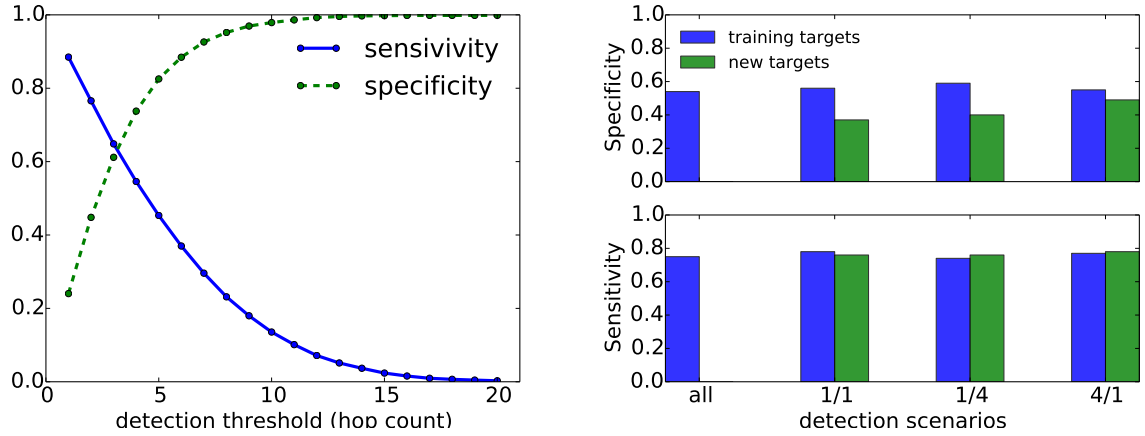


Figure 2.8: Detector accuracy under various scenarios. (left) We run detection on the same targets used in training and vary the detection threshold; increasing the threshold reduces sensitivity and improves specificity; (right) We perform detection using both training targets and new targets not used in the training process; we set the detection threshold to two hops (the average estimation error of the model); as we vary the ratio between training and detection targets, the sensitivity for the new targets is comparable to that of the training targets, while the specificity is lower; "all" indicates that we perform training and detection on all targets, therefore there are no new targets.

spoofed packets, we do so at the expense of tagging the majority of legitimate packets as spoofed. Modifying the detection threshold adjusts the trade-off: we find that the best balance between sensitivity and specificity is when the threshold is just above three hops.

Summary. Learning maps can trade-off high accuracy in detecting spoofed traffic with a high rate of false positives. This indicates that learning maps could be an important part of a bigger framework for spoofing detection, which could process the false alarms faster or dynamically adjust the detection threshold to control their rate.

2.2.4.3 Model transfer

To understand whether our learned model is general and can be transferred to new hosts, we split the data set into two disjoint sets, such that the targets in each set are disjoint. We train a model using only the targets in the first set and then run detection on both sets. Our goal is to understand how the detector performance for a target changes when we use a model learned for other targets. We consider three different splits: the number of targets in each set is the same ("1/1" split), the

number of training targets is four times larger ("4/1" split) and, the training targets are four times fewer ("1/4" split). We set the detection threshold at two hops and compute the average sensitivity and specificity for spoofing detection on each set across all splits.

Figure 2.8(right) presents the results. The sensitivity of targets not part of the training process is comparable to that of those targets used in training, showing that our models are transferable to new targets with little loss in accuracy. This reduces the cost of global deployment as new targets do not need to gather training data and build their own models. An interesting observation from Figure 2.8 is that the specificity of new targets is lower than for training targets: raising false alerts is more likely on hosts not part of the training.

Summary. Maps learned at a single location are transferable to other servers in the Internet with little loss in accuracy. This is because maps are learned based on structural network properties that are the same from every vantage point and do not depend on specific locations.

2.2.4.4 Performance

Training a representative embedding does not require significant resources. Learning a model with hop counts from 100,000 IPs to 96 servers takes about one hour on a server with four 3.5GHz quad-core Intel Xeon processors and 128GB of RAM (Table 2.3). Each model takes less than 5MB of memory and thus is easily transferable over the network. Detection is fast as well. Using the same machine used for training, and given an IP packet, we are able to decide whether the packet is spoofed or not in under a millisecond. While running the detector at line speed on a network gateway instead of a powerful GPU machine may reduce these numbers, recent deep learning platforms, such as Net2Vec (Gonzalez et al., 2017), that are able to capture and process packets at 60Gbps can make deployment easier and faster.

2.2.5 Discussion: Limitations and opportunities

Accuracy trade-off. Our detector trades off accuracy for generality. Low specificity may be unacceptable for many operators, who would still need to sift through alerts to identify legitimate packets incorrectly identified as spoofed. In practice, we envision that our approach works in

Number of source IPs	1,000	10,000	100,000
Time to train (min)	1	6	58
Size of model (MB)	4.5	4.5	4.5

Table 2.3: Performance of training for data sets containing hop count information from 96 servers to 1,000, 10,000, and 100,000 IPs. Each data set is sparse, with only about 15% of all entries available.

conjunction with network-based maps and host maps built from passive measurements to offer a comprehensive and cost-efficient spoofing detection solution.

Non-metric Internet. Paths between Internet hosts are dictated by routing policies and AS peering agreements, and are not always shortest in terms of hop counts. Estimating the hop count between two hosts as the Euclidean distance between their embeddings does not capture the intricacies of Internet routing and may introduce errors. We are working on reducing these errors in two ways. First, we plan to introduce AS membership as an additional data set in the deep learning framework. Knowing to which AS a host belongs to can help constrain its possible positions in the embedding space. Second, we plan to add IP addresses and hop counts to routers on the path between a source and a destination. While a perfect metric embedding of Internet hop counts is impossible (Lumezanu et al., 2007), we can reduce the learning error and better predict anomalies.

Furthermore, we are exploring non-metric embeddings. It is possible to learn structural embeddings of the Internet that estimate other network properties helpful in spoofing detection. For example, graph embeddings preserve local and remote network structure features such as first- and second-order neighbors (Wang et al., 2016b). We are investigating how to use such mechanisms to devise spoofing detectors that avoid the shortcomings of hop based spoofing detection.

Dynamic Internet. The dynamic nature of the Internet with frequent misconfigurations, outages, or policy changes means that hop counts or IP addresses may also change (Cunha et al., 2011; Padmanabhan et al., 2016). If this happens, we may need to retrain the embedding model to reflect the updated values. A practical deployment would passively listen to incoming legitimate traffic, or selectively probe IPs from learned maps, and update existing maps to reflect new hop count values. We are currently working on how to re-train our model to learn a more accurate hop

count estimation in the presence of new hop count data.

Asymmetric Internet. Internet routing may not always be symmetric (John et al., 2010). Due to ISP routing practices, the direct and reverse route between two nodes may be different, resulting in potentially different hop count measurements between the same pair of nodes. Because we compute hop count information from TTL values decremented exclusively by routers on the *direct* path between source IPs and targets, our detection is not impaired by the routing asymmetry.

Bootstrapping. Learning a representative structural model of the Internet requires hop count measurements from many vantage points. Not all enterprises have access to many geographically distributed monitors to collect hop count data for the initial model training. There are two possibilities to generate learned maps in such scenarios. First, one could use third-party measurement data, such as that provided by CAIDA (Ark), to train a descriptive model and generate learned maps. Second, an enterprise could deploy a model already trained in another location. As we showed in Section 2.2.4.3, our models are transferable with little loss in accuracy.

Attack types. We assumed a spoofing attack carried by a random attacker without any knowledge of the network topology. In reality, some attackers may be able to obtain information about the network that could help subvert the defense. For example, if an attacker controls multiple bots, it can send the attack from the bots that have a more popular hop count value to the target. This maximizes the likelihood of passing by our filter as there are more IP addresses with which to spoof the source. Furthermore, if the attacker learns the hop count between the spoofed IP and the target, it can also spoof the TTL field of the attack packets, inserting a value that corresponds to the learned hop count. In this case, no hop count filtering can detect the attack. To learn the hop count between two arbitrary hosts, one can use DIP (Li et al., 2018a), the same framework we employ, or the algorithm described by Barford *et al.* (Eriksson et al., 2009). However, both mechanisms require coordination and control of multiple geographically distributed servers, available only to more complex attackers.

CHAPTER 3

Extracing Latent Information from Abandoned Speech Interpretations

In a modern spoken language understanding (SLU) system, the natural language understanding (NLU) module takes interpretations of a speech from the automatic speech recognition (ASR) module as the input. The NLU module usually uses the first best interpretation of a given speech in downstream tasks such as domain and intent classification. However, the ASR module might misrecognize some speeches and the first best interpretation could be erroneous and noisy. Solely relying on the first best interpretation could make the performance of downstream tasks non-optimal especially when the ASR model does not perform well. To address this issue, we introduce a series of simple yet efficient models for improving the understanding of semantics of the input speeches by collectively exploiting the *n*-best speech interpretations from the ASR module.

3.1 Introduction

Currently, voice-controlled smart devices are widely used in multiple areas to fulfill various tasks, e.g. playing music, acquiring weather information and booking tickets. The SLU system employs several modules to enable the understanding of the semantics of the input speeches. When there is an incoming speech, the ASR module picks it up and attempts to transcribe the speech. An ASR model could generate multiple interpretations for most speeches, which can be ranked by their associated confidence scores. Among the *n*-best hypotheses, the top-1 hypothesis is usually transformed to the NLU module for downstream tasks such as domain classification, intent classification and named entity recognition (slot tagging). Multi-domain NLU modules are usually designed hierarchically (Tur and De Mori, 2011). For one incoming utterance, NLU modules will

firstly classify the utterance as one of many possible domains and the further analysis on intent classification and slot tagging will be domain-specific.

In spite of impressive development on the current SLU pipeline, the interpretation of speech could still contain errors. Sometimes the top-1 recognition hypothesis of ASR module is ungrammatical or implausible and far from the ground-truth transcription (Peng et al., 2013; Jyothi et al., 2012). Among those cases, we find one interpretation exact matching with or more similar to transcription can be included in the remaining hypotheses ($2^{nd} - n^{th}$).

To illustrate the value of the $2^{nd} - n^{th}$ hypotheses, we count the frequency of exact matching and more similar (smaller edit distance compared to the 1^{st} hypothesis) to transcription for different positions of the n-best hypotheses list. Table 3.1 exhibits the results. For the explored dataset, we only collect the top 5 interpretations for each utterance ($n = 5$). Notably, when the correct recognition exists among the 5 best hypotheses, 50% of the time (sum of the first row's percentages) it occurs among the $2^{nd} - 5^{th}$ positions. Moreover, as shown by the second row in Table 3.1, compared to the top recognition hypothesis, the other hypotheses can sometimes be more similar to the transcription.

Table 3.1: Spoken recognition quality distribution of the n best hypotheses.

n Best Rank Position	2^{nd}	3^{rd}	4^{th}	5^{th}
Match	19%	14%	10%	7%
Prob (better than 1^{st} best)	22%	17%	16%	15%

Over the past few years, we have observed the success of reranking the n-best hypotheses (Peng et al., 2013; Charniak and Johnson, 2005; Morbini et al., 2012; Dikici et al., 2012; Sak et al., 2011b, 2010, 2011a; Collins et al., 2005; Chan and Woodland, 2004) before feeding the best interpretation to the NLU module. These approaches propose the reranking framework by involving morphological, lexical or syntactic features (Sak et al., 2011a; Collins et al., 2005; Chan and Woodland, 2004), speech recognition features like confidence score (Peng et al., 2013; Morbini et al., 2012), and other features like number of tokens, rank position (Peng et al., 2013). They are effective to select the best from the hypotheses list and reduce the word error rate (WER) (Oba et al., 2007) of speech recognition.

Table 3.2: Motivating example: comparison of ASR *n*-Best hypotheses with the corresponding transcription.

Transcription	1^{st} best	2^{nd} best	3^{rd} best
play muse	play news	**play muse**	play mus
track on bose	check on bowls	check on bose	**track on bose**
harry porter	how **porter**	how patter	**harry** power

Those reranking models could benefit the first two cases in Table 3.2 when there is an utterance matching with transcription. However, in other cases like the third row, it is hard to integrate the fragmented information in multiple hypotheses.

This paper proposes various methods integrating *n*-best hypotheses to tackle the problem. To the best of our knowledge, this is the first study that attempts to collectively exploit the *n*-best speech interpretations in the SLU system. This paper serves as the basis of our *n*-best-hypotheses-based SLU system, focusing on the methods of integration for the hypotheses. Since further improvements of the integration framework require considerable setup and descriptions, where jointly optimized tasks (e.g. transcription reconstruction) trained with multiple ways (multitask (Caruana, 1997), multistage learning (Gong et al., 2013)) and more features (confidence score, rank position, etc.) are involved, we leave those to a subsequent article.

This paper is organized as follows. Section 3.2 introduces the Baseline, Oracle and Direct models. Section 3.3 describes proposed ways to integrate *n*-best hypotheses during training. The experimental setup and results are described in Section 3.4. Section 3.5 contains conclusions and future work.

3.2 Baseline, Oracle and Direct Models

3.2.1 Baseline and Oracle

The preliminary architecture is shown in Fig. 3.1. For a given transcribed utterance, it is firstly encoded with Byte Pair Encoding (BPE) (Sennrich et al., 2015), a compression algorithm splitting words to fundamental subword units (*pairs of bytes* or *BP*s) and reducing the embedded vocabulary

size. Then we use a BiLSTM (Schuster and Paliwal, 1997) encoder and the output state of the BiLSTM is regarded as a vector representation for this utterance. Finally, a fully connected Feed-forward Neural Network (FNN) followed by a softmax layer, labeled as a multilayer perceptron (MLP) module, is used to perform the domain/intent classification task based on the vector.

Figure 3.1: Baseline pipeline for domain or intent classification.

For convenience, we simplify the whole process in Fig.3.1 as a mapping *BM* (Baseline Mapping) from the input utterance S to an estimated tag's probability $p(\tilde{t})$, where $p(\tilde{t}) \leftarrow BM(S)$. The *Baseline* is trained on transcription and evaluated on ASR 1^{st} best hypothesis ($S =$ ASR 1^{st} best). The *Oracle* is trained on transcription and evaluated on transcription ($S =$ Transcription). We name it Oracle simply because we assume that hypotheses are noisy versions of transcription.

3.2.2 Direct Models

Besides the Baseline and Oracle, where only ASR 1-best[1] hypothesis is considered, we also perform experiments to utilize ASR n-best hypotheses during evaluation. The models evaluating with n-bests and a BM (pre-trained on transcription) are called *Direct Models* (in Fig. 3.2):

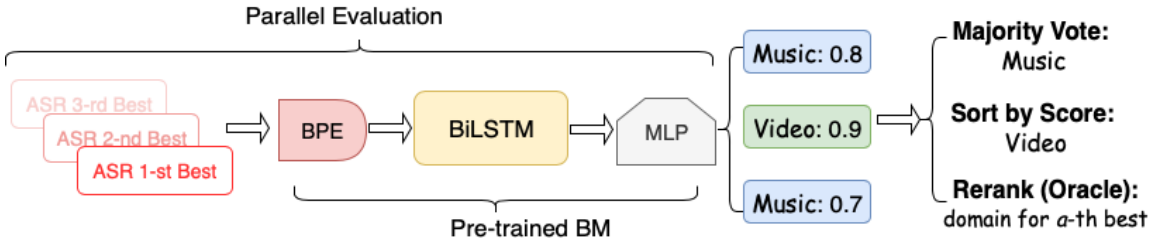


Figure 3.2: Direct models evaluation pipeline.

- *Majority Vote.* We apply the BM model on each hypothesis independently and combine the predictions by picking the majority predicted label, i.e. Music.

[1]We use ASR *n-best hypotheses* or n-bests to denote the top n interpretations of a speech, and the *1,5-best* standing for the top 1 or 5 hypotheses.

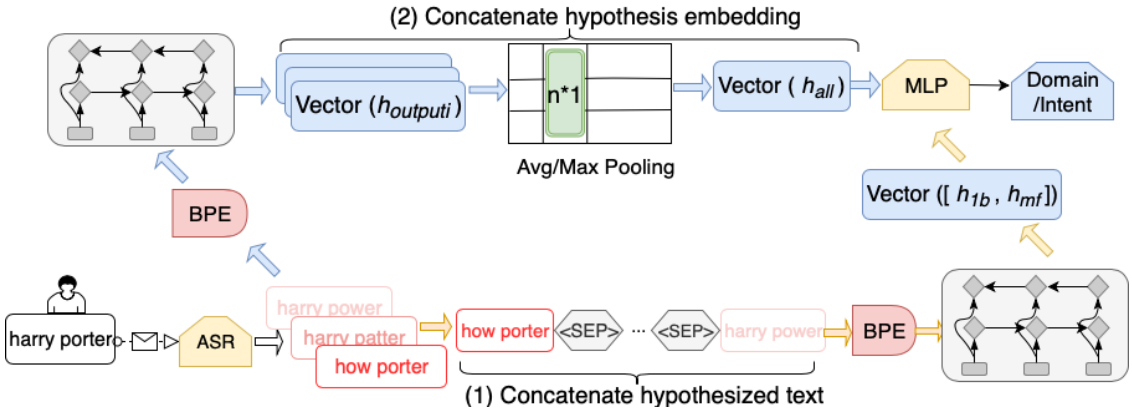


Figure 3.3: Integration of n-best hypotheses with two possible ways: 1) concatenate hypothesized text and 2) concatenate hypothesis embedding.

- *Sort by Score.* After parallel evaluation on all hypotheses, sort the prediction by the corresponding confidence score and choose the one with the highest score, i.e. Video.
- *Rerank (Oracle).* Since the current rerank models (e.g., (Peng et al., 2013; Charniak and Johnson, 2005; Morbini et al., 2012)) attempt to select the hypothesis most similar to transcription, we propose the Rerank (Oracle), which picks the hypothesis with the smallest edit distance to transcription (assume it is the a-th best) during evaluation and uses its corresponding prediction.

3.3 Integration of N-BEST Hypotheses

All the above mentioned models apply the BM trained on one interpretation (transcription). Their abilities to take advantage of multiple interpretations are actually not trained. As a further step, we propose multiple ways to integrate the n-best hypotheses during training. The explored methods can be divided into two groups as shown in Fig. 3.3. Let $H_1, H_2, ..., H_n$ denote all the hypotheses from ASR and $bp_{H_k,i} \in BPs$ denotes the i-th pair of bytes (BP) in the k^{th} best hypothesis. The model parameters associated with the two possible ways both contain: embedding e_{bp} for pairs of bytes, BiLSTM parameters θ and MLP parameters W, b.

3.3.1 Hypothesized Text Concatenation

The basic integration method (*Combined Sentence*) concatenates the n-best hypothesized text. We separate hypotheses with a special delimiter (<SEP>). We assume BPE totally produces m BPs (delimiters are not split during encoding). Suppose the n^{th} hypothesis has j pairs. The entire model can be formulated as:

$$(h_1, ..., h_m) \leftarrow BiLSTM_\theta(bp_{H_1,1}, ..., bp_{<sep>}, ..., bp_{H_n,j}) \tag{3.1}$$

$$p(\tilde{t}) = \sigma(W[h_{1b}, h_{mf}] + b) \tag{3.2}$$

In Eqn. 3.1, the connected hypotheses and separators are encoded via BiLSTM to a sequence of hidden state vectors. Each hidden state vector, e.g. h_1, is the concatenation of forward h_{1f} and backward h_{1b} states. The concatenation of the last state of the forward and backward LSTM forms the output vector of BiLSTM (concatenation denoted as $[,]$). Then, in Eqn. 3.2, the MLP module defines the probability of a specific tag (domain or intent) $\tilde{t}$ as the normalized activation (σ) output after linear transformation of the output vector.

3.3.2 Hypothesis Embedding Concatenation

The concatenation of hypothesized text leverages the n-best list by transferring information among hypotheses in an embedding framework, BiLSTM. However, since all the layers have access to both the preceding and subsequent information, the embedding among n-bests will influence each other, which confuses the embedding and makes the whole framework sensitive to the noise in hypotheses.

As the second group of integration approaches, we develop models, *PoolingAvg/Max*, on the concatenation of hypothesis embedding, which isolate the embedding process among hypotheses and summarize the features by a pooling layer. For each hypothesis (e.g., i^{th} best in Eqn. 3.3 with j pairs of bytes), we could get a sequence of hidden states from BiLSTM and obtain its final output state by concatenating the first and last hidden state (h_{output_i} in Eqn. 3.4). Then, we stack all the output states vertically as shown in Eqn. 3.5. Note that in the real data, we will not always have a

fixed size of hypotheses list. For a list with r ($< n$) interpretations, we get the embedding for each of them and pad with the embedding of the first best hypothesis until a fixed size n. When $r \geq n$, we only stack the top n embeddings. We employ h_{output_1} for padding to enhance the influence of the top 1 hypothesis, which is more reliable. Finally, one unified representation could be achieved via *Pooling* (Max/Avg pooling with n by 1 sliding window and stride 1) on the concatenation and one score could be produced per possible tag for the given task.

$$(h_{H_i,1}, ..., h_{H_i,j}) \leftarrow BiLSTM_\theta(bp_{H_i,1}, ..., bp_{H_i,j}) \tag{3.3}$$

$$h_{output_i} = [h_{H_i,1b},\ h_{H_i,jf}] \tag{3.4}$$

$$h_{outputs} = \left\{ \begin{array}{ll} \left\{ \begin{array}{c} h_{output_1} \\ \dots \\ h_{output_r} \end{array} \right\} & r - bests \\ \left\{ \begin{array}{c} h_{output_1} \\ \dots \end{array} \right\} & Padding\ with\ h_{output_1} \end{array} \right\} \tag{3.5}$$

$$h_{all} = Pooling(h_{outputs}) \tag{3.6}$$

$$p(\tilde{t}) = \sigma(W h_{all} + b) \tag{3.7}$$

3.4 Experiment

3.4.1 Dataset

We conduct our experiments on $\sim$ 8.7M annotated anonymised user utterances. They are annotated and derived from requests across 23 domains.

3.4.2 Performance on Entire Test Set

Table 3.3 shows the relative error reduction (RErr)[2] of Baseline, Oracle and our proposed models on the entire test set ($\sim$ 300K utterances) for multi-class domain classification. We can see among all the direct methods, predicting based on the hypothesis most similar to the transcription (Rerank (Oracle)) is the best.

Table 3.3: Micro and Macro F1 score for multi-class domain classification.

Category	Model	RErr(%)
Baseline		**0.00**
Integration	PoolingAvg	**14.29**
	PoolingMax	13.20
	Combined Sentence	11.67
Direct	Sort by Score	1.85
	Majority Vote	1.64
	Rerank (Oracle)	**3.71**
Oracle		27.04

As for the other models attempting to integrate the n-bests during training, PoolingAvg gets the highest relative improvement, 14.29%. It as well turns out that all the integration methods outperform direct models drastically. This shows that having access to n-best hypotheses during training is crucial for the quality of the predicted semantics.

3.4.3 Performance Comparison among Various Subsets

To further detect the reason for improvements, we split the test set into two parts based on whether ASR first best agrees with transcription (ignore difference with wake words of hypotheses) and evaluate separately. Comparing Table 3.4 and Table 3.5, obviously the benefits of using multiple hypotheses are mainly gained when ASR 1^{st} best disagrees with the transcription. When ASR 1^{st} best agrees with transcription, the proposed integration models can also keep the performance. Under that condition, we can still improve a little (3.56%) because, by introducing multiple

[2]The RErr for a model m is calculated by comparing the relative difference between $100\% - MicroF1_m$ and $100\% - MicroF1_{\text{Baseline}}$.

Table 3.4: Performance comparison for the subset ($\sim$ 19%) where ASR first best disagrees with transcription.

Category	Model	RErr(%)
Baseline		**0.00**
Integration	PoolingAvg	24.67
	PoolingMax	26.23
	Combined Sentence	19.23
Direct	Sort by Score	9.95
	Majority Vote	7.59
	Rerank (Oracle)	7.25
Oracle		53.02

Table 3.5: Performance comparison for the subset ($\sim$ 81%) where ASR first best agrees with transcription.

Category	Model	RErr(%)
Baseline		**0.00**
Integration	PoolingAvg	3.56
	PoolingMax	-0.38
	Combined Sentence	4.50
Direct	Sort by Score	-8.269
	Majority Vote	-3.19
	Rerank (Oracle)	0.00
Oracle		0.00

ASR hypotheses, we could have more information and when the transcription/ASR 1^{st} best does not appear in the training set's transcriptions, its n-bests list may have similar hypotheses included in the training set's n-bests. Then, our integration model trained on n-best hypotheses as well has clue to predict. The series of comparisons reveal that our approaches integrating the hypotheses are robust to the ASR errors and whenever the ASR model makes mistakes, we can outperform more significantly.

3.4.4 Improvements on Different Domains and Different Numbers of Hypotheses

Among all the 23 domains, we choose 8 popular domains for further comparisons between the Baseline and the best model of Table 3.3, PoolingAvg. Fig. 3.4 exhibits the results. PoolingAvg

consistently improves the accuracy for all 8 domains.

In the previous experiments, the number of utilized hypotheses for each utterance during evaluation is five, which means we use the top 5 interpretations when the size of ASR recognition list is not smaller than 5 and use all the interpretations otherwise. Changing the number of hypotheses while evaluation, Fig. 3.5 shows a monotonic increase with the access to more hypotheses for the PoolingAvg and PoolingMax (Sort by Score is shown because it is the best achievable direct model while the Rerank (Oracle) is not realistic). The growth becomes gentle after four hypotheses are leveraged.

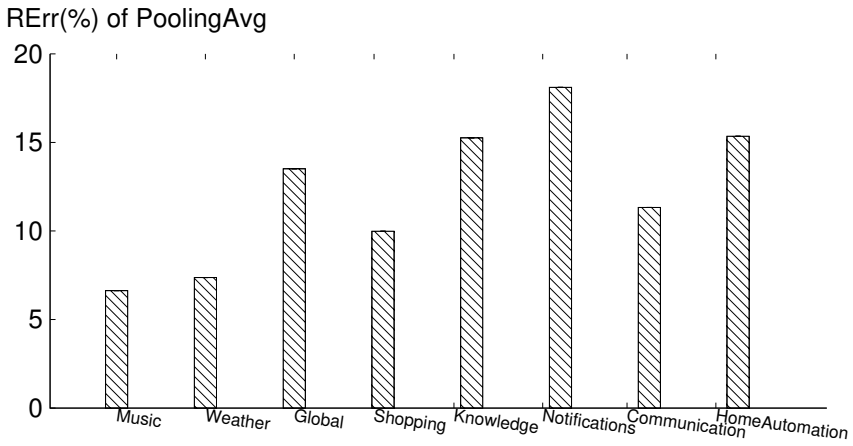


Figure 3.4: Improvements on important domains.

3.4.5 Intent Classification

Table 3.6: Intent classification for three important domains.

Domain	Metric	Shopping	Knowledge	Communication
Baseline	RErr (%)	0.0	0.0	0.0
Oracle		47.63	40.28	32.89
PoolingAvg		**25.55**	**25.00**	**11.92**

Since another downstream task, intent classification, is similar to domain classification, we just show the best model in domain classification, PoolingAvg, on domain-specific intent classification

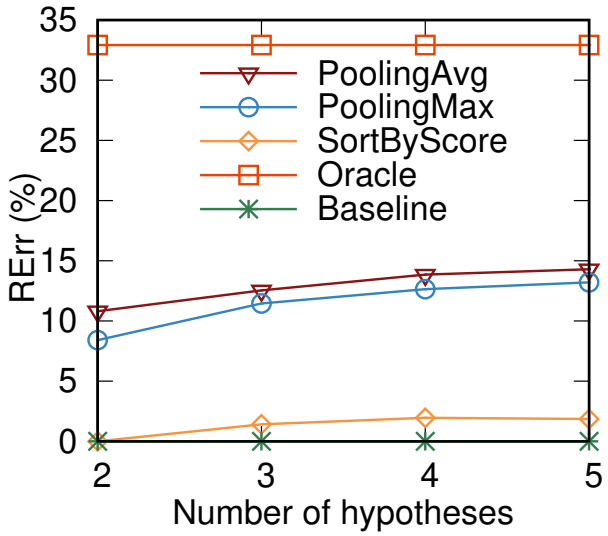


Figure 3.5: The influence of different amount of hypotheses.

for three popular domains due to space limit. As Table 3.6 shows, the margins of using multiple hypotheses with PoolingAvg are significant as well.

3.5 Conclusions and Future Work

This paper improves the SLU system robustness to ASR errors by integrating n-best hypotheses in different ways, e.g. the aggregation of predictions from hypotheses or the concatenation of hypothesis text or embedding. We can achieve significant classification accuracy improvements over production-quality baselines on domain and intent classifications, 14% to 25% relative gains. The improvement is more significant for a subset of testing data where ASR first best is different from transcription. We also observe that with more hypotheses utilized, the performance can be further improved. In the future, we aim to employ additional features (e.g. confidence scores for hypotheses or tokens) to integrate n-bests more efficiently, where we can train a function f to obtain a weight for each hypothesis embedding before pooling. In addition, since more improvements are from the disagree part, which indicates the integration model is more helpful for low-quality hypotheses. We want to discuss, if the quality of hypotheses gets improved by a better ASR recognizing algorithm or some cleaning techniques on hypotheses (our datasets are raw recognition results from ASR without further cleaning), the change of the benefits brought by our integration

model and how to improve the design for this condition. Another direction is using deep learning framework to embed the word lattice (Liu et al., 2014) or confusion network (Hakkani-Tür et al., 2006; Tur et al., 2002), which can provide a compact representation of multiple hypotheses and more information like times, in the SLU system.

CHAPTER 4

Amplified Negative Sampling: A Sample-Efficient Training Algorithm

In this chapter, we propose a new sample-efficient training method, called *amplified negative sampling*, for training multi-class classifiers with a large output-class size. Our method jumps out of the framework of softmax appximation and directly tackle the optimum convergence point of the learning algorithm. Our proposed method is based on our analysis of the well-known negative sampling technique (Mikolov et al., 2013c) and is designed for (1) higher performance in the general task and (2) lower training computational cost. Our analysis provides a better understanding of a few empirical observations that are well known among practitioners. Our experiments on real-world datasets demonstrate that our proposed method leads to sampling cost savings with performance boost compared to the standard technique.

4.1 Introduction

In this work,we provide a novel sample-efficient method for training a multi-class classifier $C : X \rightarrow Y$ (X: input features, Y: output class labels) when the output-class size is large, say, $|Y| = 50,000$. Typically, when a classifier is modeled as a neural network, the final layer is implemented as a softmax layer with *one output neuron per each output class label* $y \in Y$, making it prohibitively expensive to train even for a reasonably large output-class size. To address this computational challenge, a number of techniques have been proposed, such as hierarchical softmax (Morin and Bengio, 2005),negative sampling (Mikolov et al., 2013a), adaptive softmax (Bengio, 2008) and its variants (Rawat et al., 2019; Blanc and Rendle, 2017; Grave et al., 2017; Chen et al., 2015; Bai et al., 2017). Due to its simplicity and efficiency, negative sampling is one of the most popular

techniques used in practice (Mikolov et al., 2013c; Wang et al., 2017; Grover and Leskovec, 2016a; Barkan and Koenigstein, 2016). In particular, it is widely utilized in many embedding frameworks, such as word embedding (Mikolov et al., 2013c), graph embedding (Grover and Leskovec, 2016a; Wang et al., 2017), and product-user embedding (Barkan and Koenigstein, 2016).

The key idea behind negative sampling is as follows: Given a training data point (x_i, y_i), the standard training algorithm updates the weights of the output neurons for *all* y's $\in Y$, not just for the training label y_i, making the training cost proportional to the output-class size $|Y|$. Negative sampling avoids this high cost by adjusting the weights for (1) the given training label, $y = y_i$ ("the positive sample") and (2) just a few y's that are randomly sampled from $Y - \{y_i\}$ ("negative samples"). Clearly, taking a few negative samples reduces the training cost by several orders of magnitudes when the output class size is large.

In general, it is reported that using a larger negative sample size leads to better downstream performance. For example, when Mikolov used negative sampling to embed words into high-dimensional vectors in (Mikolov et al., 2013c), he reported between 2-15% increase in downstream task performance when he used the 15 negative samples ($k = 15$) compared to 5 negative samples ($k = 5$). Unfortunately, the training cost of $k = 15$ is three times as large as that of $k = 5$, making its use significantly less appealing in practice. For instance, since training on a larger corpus generally improves the downstream performance as well, it may be the case that using a smaller k on a larger corpus may be just as good as or even better than using a larger k on a smaller corpus. This is the primary topic of this chapter. Is it possible to get the best of both worlds? Can we achieve a higher-quality model trained on a larger k without paying its training cost?

To obtain an answer to this question, we first conduct a rigorous mathematical analysis of the impact of the negative-sampling technique on the accuracy of the learned model. The result of this analysis will show exactly how negative sampling affects the accuracy of the learned model and provide theoretical explanations for a few empirical observations that have been well known among practitioners. It will also shed light on how it can be further improved for higher training efficiency. Based on this insight, we find a surprisingly simple yet effective modification to the negative sampling technique, named *amplified negative sampling*. Compared to the standard negative-sampling technique, our proposed technique can be used to either (1) *improve the predic-*

tion accuracy of the trained model *for the same training cost* or (2) *lower the training cost for the same prediction accuracy*. We demonstrate the effectiveness of our proposed technique through extensive experiments on real-world data sets.

In summary, we make the following contributions in this chapter. We propose *amplified negative sampling*, a simple yet effective modification to the widely-used negative-sampling technique that can improve its accuracy and lower its training cost based on our rigorous mathematical analysis. We compare the effectiveness of our proposed technique with standard negative sampling by conducting an extensive set of experiments on real-world data sets. Our results show that the effectiveness of our technique is in line with our theoretical prediction and can often be *twice as sample efficient* as the standard technique.

The rest of this chapter is organized as follows. In Section 4.3, we formally describe the multi-class classification problem and review the standard negative sampling technique. Then in Section 4.4, we propose amplified negative sampling and give the rigourous matematical analysis of the method. In Section 4.5, we present the results of our experiments. We review related work in Section 4.2 and wrap up the chapter in Section 4.6.

4.2 Related Work

Softmax has been widely used in various models. In large output class classification problem, the intractable normalizing constant of softmax function will slow down the computation efficiency greatly. There are three major types of strategies have been investigated by the research community, including sampled softmax(Bengio, 2008), hierarchical softmax(Morin and Bengio, 2005) and spherical softmax(Vincent et al., 2015). We are not aiming to approximate the softmax function. This work makes two related yet distinct contributions: (1) a rigorous mathematical analysis of negative sampling and (2) an efficient training method based on negative sampling strategy for a large classifier. The (Ruiz et al., 2018) share a similar scope with us which is not targeting on softmax appximation. Since the amplifying factor is distribution agnostic, this method actually can be applied to all the sampling-based softmax approximation method(Blanc and Rendle, 2017; Rawat et al., 2019).

Sampled Softmax This category contains the method which generates a subset of negative samples to avoid the high overhead of full negative sampling. Among this category, one class of methods try to generate samples from the softmax distribution. An adaptive sampling method was proposed by Bengio (Bengio, 2008) inspired by the importance sampling. Another prominent example is the negative sampling (Mikolov et al., 2013a) which uses a simple noisy distribution to generate negative samples. TAPAS (Bai et al., 2017) uses a two pass scheme to generate samples from two different size candidate pools to reduce the sample overhead. Hashing method (Bakhtiary et al., 2015; Vijayanarasimhan et al., 2014) is also applied to either find the closest class or partial computation. (Rawat et al., 2019). Another class of methods instead focus on the sampled loss, including Noisy Contrastive Estimation(NCE) (Gutmann and Hyvärinen, 2010) by assuming the partition as an extra parameter to be computed during the computation and Adversarial Contrastive Estimation(ACE) (Bose et al., 2018) and (Schroff et al., 2015; Mussmann et al., 2017) selects the hardest negative examples.

Hierachical Softmax Hierarchical softmax was introduced in (Goodman) by utilizing the cluster structure to reduce the computation cost of softmax function. Bengio (Morin and Bengio, 2005) extend it into the tree structure. Due to the different inference procedure, the hierarchical softmax need extra steps to update the tree structure and maintain its property. Various method are proposed to stabilize this process such as class similarity, frequency binning and other optimization techniques. Zweig did some experiments to compare various tree structures.

Spherical softmax and kernel method The spherical softmax was proposed in (Vincent et al., 2015; de Brébisson and Vincent, 2015) which use quadratic function to replace the exponential function and enable faster computation of the gradients. However, these method seems not quite stable when the output label size is large. Kernel-based methods are also explored in (Blanc and Rendle, 2017; Rawat et al., 2019). These works introduces quadratic kernel and random Fourier features which show very promising results.

4.3 Framework

In this section, we briefly go over the general problem formulation of multi-class classifier learning and negative sampling to introduce key notation used in this paper.

4.3.1 Preliminaries

We are given a dataset $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, where $x_i \in X$ is an *input feature* and $y_i \in Y$ is an *output class label*. We assume a discrete space of feature values x_i and output labels y_i, such that x_i and y_i take an integer value between $1 \leq x_i \leq m$ and $1 \leq y_i \leq m$. The multi-class classifier learning problem is to find a classifier $C : X \rightarrow Y$ that returns the correct label y_i given the input feature x_i: $y_i = C(x_i)$. Due to noise in the dataset and the uncertainty in predicting the correct label, this problem is often formulated as finding a conditional probability distribution $P(y|x)$ from the dataset D, which is interpreted as the probability that the correct output label is y given the input feature x.

Note that this formulation encompasses not just the multi-class classifier learning problem, but also most of the "data embedding" problems, such as word embedding (Mikolov et al., 2013a,c), graph embedding (Grover and Leskovec, 2016b), and item embedding (Barkan and Koenigstein, 2016). For example, the well-known skip-gram model for word2vec (Mikolov et al., 2013c) falls under this formulation by defining a *context word* as an input feature x_i and any *target word* that appears near the context word as an output label y_i.

Learning the conditional probability function $f(x,y) = P(y|x)$ from the dataset D is done by assuming a parameterized hypothesis space $f_\theta(x,y) = P_\theta(y|x)$, where the hypothesis space $f_\theta : (x,y) \rightarrow [0,1]$ is a space of differentiable functions parameterized by $\theta \in R^d$. Among all possible parameters $\theta \in R^d$, an *optimal parameter* θ^* is chosen to minimize the loss function $L(f_\theta, D)$, where $L(f_\theta, D)$ captures the "loss of f_θ" or the difference between f_θ and D. Multiple definitions of the loss function $L(f_\theta, D)$ are used in practice, including $L1$, $L2$, and *cross entropy*:

$$L_1 : \sum_{(x_i,y_i)\in D} \sum_{y\in Y} |\mathbb{1}(y = y_i) - f_\theta(x_i, y)| \tag{4.1}$$

$$L_2 : \sum_{(x_i,y_i)\in D} \sum_{y\in Y} (\mathbb{1}(y = y_i) - f_\theta(x_i, y))^2 \tag{4.2}$$

$$L_{CE} : - \sum_{(x_i,y_i)\in D} \sum_{y\in Y} [\mathbb{1}(y = y_i) \log f_\theta(x_i, y) + (1 - \mathbb{1}(y = y_i)) \log (1 - f_\theta(x_i, y))] \tag{4.3}$$

Here, $\mathbb{1}(y = y_i)$ is an indicator function that takes the value 1 if $y = y_i$ and 0 otherwise. In order to make our discussion concrete, we primarily assume *L2* as our loss function in the rest of this paper and simply state the result of our analysis for the other loss functions.

The gradient-descent method is often utilized to identify the parameter θ^* that minimizes the loss function $L(f_\theta, D)$. Given the definition of the *L2* loss function, its gradient is:

$$\nabla_\theta L_2(f_\theta, D) = -2 \sum_{(x_i,y_i)\in D} \left[\sum_{y\in Y} (\mathbb{1}(y = y_i) - f_\theta(x_i, y)) \nabla_\theta f_\theta(x_i, y) \right] \tag{4.4}$$

Note that the inner summation of the above equation makes its computation prohibitively expensive: For each training data $(x_i, y_i) \in D$, we take the inner sum over *every* output label $y \in Y$, not just the training label y_i.[1] This makes the computational cost *proportional to the output class size* $|Y|$. We refer to the training method that computes the full gradient of Equation 4.4 as *full-gradient training*.

4.3.2 Negative Sampling

Negative sampling is a technique that tries to reduce the high computational cost of full-gradient training. The idea of negative sampling was originally proposed in 2010 as Noisy Contrastive Estimation (NCE) (Gutmann and Hyvärinen, 2010), which was generalized for natural language

[1] In certain cases, this sum over every $y \in Y$ is implicitly added to the hypothesis space $f_\theta(x, y)$. For example, when implemented as a neural network, the final layer is typically implemented as a softmax layer with one neuron per output label, which has the same effect as summing over ever $y \in Y$.

processing by Mihn in (Mnih and Teh, 2012). It was used as part of the word2vec computation (Mikolov et al., 2013a), which led to a wide adoption for general vector-embedding problems (Grover and Leskovec, 2016a; Barkan and Koenigstein, 2016; Grover and Leskovec, 2016b).

The basic idea of negative sampling can be summarized as follows: Given a training data (x_i, y_i), we refer to y_i as the "*positive sample*" and all other label $y \in Y - \{y_i\}$ as "*negative samples*." Full-gradient training sums up the gradients from (1) the positive sample y_i and (2) *all* negative samples $y(\neq y_i)$. Negative sampling, instead, sums up the gradients from (1) the positive sample y_i and (2) *just a few randomly-selected negative samples* $y \in Y - \{y_i\}$.

More precisely, we use NEG-$k(i)$ to represent the k randomly-chosen negative samples for the ith training data (x_i, y_i). That is, NEG-$k(i)$ is a size-k random subset of $Y - \{y_i\}$. Negative sampling then *approximates* the $L2$ loss function $L_2(f_\theta, D)$ as follows:

$$
\begin{aligned}
& L_2(f_\theta, D) \\
&= \sum_{(x_i,y_i)\in D} \left[\sum_{y\in Y} (\mathbb{1}(y = y_i) - f_\theta(x_i, y))^2 \right] && (4.5) \\
&= \sum_{(x_i,y_i)\in D} \left[(\mathbb{1}(y = y_i) - f_\theta(x_i, y))^2 \Big|_{y=y_i} \right. \\
&\qquad \left. + \sum_{y\in Y-\{y_i\}} (\mathbb{1}(y = y_i) - f_\theta(x_i, y))^2 \right] && (4.6) \\
&\approx \sum_{(x_i,y_i)\in D} \left[(1 - f_\theta(x_i, y_i))^2 + \sum_{y\in \text{NEG-}k(i)} f_\theta(x_i, y)^2 \right] && (4.7)
\end{aligned}
$$

From Equation 4.5 to 4.6, the sum over $y \in Y$ is expanded into the sum of $y = y_i$ and $y \in Y - \{y_i\}$. From Equation 4.6 to 4.7, the sum over all negative samples $y \in Y - \{y_i\}$ is approximated by the sum over NEG-$k(i)$. Clearly, this approximation can decrease the cost of computing the loss function significantly when $|Y|$ is large, which is the key reason for its efficiency. But what is the accuracy of this approximation? Will we still be able to get the same accurate model despite this approximation? If not, what is its exact impact? The next section will show the answers for those issues.

4.3.3 Optimal Model of Negative Sampling and Full-gradient Model

In this section, we investigate the achievable optimal model of the NEG-$k(i)$ and compare it with the model training with all $Y - \{y_i\}$ negative samples (Full-gradient) to see how accurate the NEG-$k(i)$ can approximate. For a better explanation, we use $\#(x)$ to represent the number of times that the input feature value x appears in the training data D and $\#(x,y)$ to represents the number of times that the feature-and-label-value pair (x,y) appears in D. More formally,

$$\#(a) = |\{(x,y) \in D \mid x = a\}|$$
$$\#(a,b) = |\{(x,y) \in D \mid x = a \text{ and } y = b\}|$$

When we select the k negative samples for the ith training data (x_i, y_i), NEG-$k(i)$, we assume that a negative sample $y \in Y - \{y_i\}$ is sampled with replacement with probability p_y. Two popular choices of the sampling probability p_y are (1) the uniform distribution $p_y = c$ for some constant c and (2) according to the frequency of y in D. Our results are stated with the generic symbol p_y without making any explicit assumption on the sampling distribution. With this notation, we now could show the analysis to the negative sampling.

Theorem 1 (Optimal Model of Negative Sampling)**.** *When the hypothesis space f_θ has sufficient capacity,*[2] *the optimal model f_{θ^*} trained with k negative samples is the following with high probability:*

- *For L1 loss:* $f_{\theta^*}(x,y) = \mathbb{1}\left(\frac{\#(x,y)}{\#(x)} > \frac{k \cdot p_y}{k \cdot p_y + 1}\right)$
- *For L2 or cross-entropy loss:* $f_{\theta^*}(x,y) = \frac{\#(x,y)}{k \cdot p_y [\#(x) - \#(x,y)] + \#(x,y)}$

That is, for example, let θ_t^* be the parameter that minimizes the L1 loss function after t training epochs. Then for any $\varepsilon > 0$,

$$\lim_{t \to \infty} P\left(\left| f_{\theta_t^*}(x,y) - \mathbb{1}\left(\frac{\#(x,y)}{\#(x)} > \frac{k \cdot p_y}{k \cdot p_y + 1}\right)\right| < \varepsilon\right) = 1,$$

where $\mathbb{1}(a > b)$ is an indicator function whose value is 1 if $a > b$ and 0 otherwise.[3] *Similar state-*

[2] By having sufficient capacity, we mean that the hypothesis space has an independently adjustable parameter per every discrete (x,y) value pair following the assumption of (Goldberg and Levy, 2014).

[3] More precisely, $\mathbb{1}(a > b)$ may take any value between 0 to 1 when $a = b$.

ments can be made for L2 and cross entropy loss functions.

While for the Full-gradient model training with all the negative samples instead of only k samples, we can get a corollary as follows.

Corollary 1.1 (Full-Gradient Model)**.** *When the hypothesis space f_θ has sufficient capacity, the respective optimal models trained with the full-gradient method are the following:*

- *For L1 loss:* $f_{\theta^*}(x,y) = \mathbb{1}\left(\frac{\#(x,y)}{\#(x)} > \frac{1}{2}\right)$
- *For L2 or cross-entropy loss:* $f_{\theta^*}(x,y) = \frac{\#(x,y)}{\#(x)}$

Note that the theorem and corollary mentioned above are achievable global optimal model but it does not guarantee that the global optimal can be always achieved in practice. The Corollary has ever been mentioned in (Levy and Goldberg, 2014) while in our another work, we provide a rigorous mathematical analysis and proof to both the Theorem 1 and the Corallary 1.1. For this paper, we do not show the analysis as we focus proposing a more efficient sample-based training method based on the Theorem 1 and the Corallary 1.1. In next section, we will show the design of the new sample-efficient training method and the analysis for it.

4.4 Amplified Negative Sampling

Our analysis and some related works like PMI (Levy and Goldberg, 2014) have shown that the optimal model trained with the full-gradient method is equivalent to the maximum likelihood estimator $f_{\theta^*}(x,y) = \frac{\#(x,y)}{\#(x)}$ while the model from negative sampling is not. Assuming that the dataset D is a representative sample from the true underlying distribution $P(y|x)$ (that is, $P(y|x) \approx \frac{\#(x,y)}{\#(x)}$) we can expect that full-gradient training are likely to result in a better model than negative sampling for a classification task.

Indeed, this is the general trend reported in the literature – not just for classification tasks but also for embedding tasks where negative sampling is frequently used. For example, when Mikolov used negative sampling to embed words into high-dimensional vectors in (Mikolov et al., 2013c), he reported noticeable accuracy increase in the word-analogy-task performance when he

used $k = 15$ compared to $k = 5$.[4] Unfortunately, the training cost of using k negative samples is proportional to k, making the use of a higher k value unappealing in practice; since training on a larger corpus generally increases the embedding quality as well, one may prefer using a smaller k on a larger corpus than using a larger k on a smaller corpus, assuming that it is bound by the same computational cost. This is where our study of *amplified negative sampling* started. Can we obtain the high quality model of higher k for the low computational cost of lower k? Can we get the best of the both worlds? We now explain how this can be achieved using amplified negative sampling.

Amplifying Factor. The key idea behind our amplified negative sampling comes from Theorem 1. From its analytic form, we observe that the optimal model $f_{\theta^*}(x,y)$ depends *only on the negative-sample size k*, not on how the samples are obtained. Given this, can we use *one negative sample multiple times during training*, pretending that it is the result from multiple random sampling? More formally, what will happen if we change the loss function $L(f_\theta, D)$ of Equation 4.7 (standard negative sampling) to the following?

$$L_2(f_\theta, D) = \sum_{(x_i,y_i)\in D} \left[(1 - f_\theta(x_i,y_i))^2 + \beta \sum_{y\in \text{NEG-}k(i)} f_\theta(x_i,y)^2 \right] \tag{4.8}$$

Note that Equation 4.13 is different from Equation 4.7 just by a constant factor β of the second term. We refer to β as a *amplifying factor*, since its intended role is to artificially "amplify" the effect of the negative samples NEG-k(i) by making its size look larger than they really are. Surprisingly, the following corollary shows that adding a amplifying factor produces this exact outcome.

Corollary 1.2 (Amplified Negative Sampling)**.** *When the hypothesis space f_θ has sufficient capacity, the respective optimal models trained with k negative samples with amplifying factor β under the three loss functions are the following with high probability:*

- *For L1 loss:* $f_{\theta^*}(x,y) = \mathbb{1}\left(\frac{\#(x,y)}{\#(x)} > \frac{\beta\cdot k\cdot p_y}{\beta\cdot k\cdot p_y + 1}\right)$
- *For L2 loss:* $f_{\theta^*}(x,y) = \frac{\#(x,y)}{\beta\cdot k\cdot p_y[\#(x) - \#(x,y)] + \#(x,y)}$
- *For cross-entropy loss:*
  $f_{\theta^*}(x,y) = \frac{\#(x,y)}{\beta\cdot k\cdot p_y[\#(x) - \#(x,y)] + \#(x,y)}$

[4] For embedding tasks, the ultimate goal is to obtain vector representations of words that lead to high accuracy for downstream tasks. For this reason, obtaining the MLE model may not necessarily be "better." However, in a relatively small range of k, it is generally observed that higher k leads to better downstream task performance as well.

Note that the factor k in Theorem 1 is replaced with βk in Corollary 1.2. That is, the amplifying factor β makes the optimal model trained with NEG-k effectively identical to the one from NEG-βk! Simply by multiplying a constant β to the loss function, we get the optimal model from a much larger sample size.

Proof for Corollary 1.2: We assume a discrete domain of input data D, where each data point $(x,y) \in D$ is an integer value pair of $1 \leq x \leq m$ and $1 \leq y \leq l$. Within the discrete domain, the most general parameterization of the function $f_\theta(x,y)$ is to assign an independent parameter per every pair of values (x,y) within $1 \leq x \leq m$ and $1 \leq y \leq l$. We use the symbol θ_{ab} to represent these parameters, i.e., $f_\theta(a,b) = \theta_{ab}$ for $1 \leq a \leq m$ and $1 \leq b \leq l$, where θ_{ab} can take any value in $[0,1]$.

Given the notation, the L_2 loss function of negative sampling is:

$$L_2(f_\theta, D) = \sum_{(x,y)\in D} \left[(1 - f_\theta(x,y))^2 + \sum_{y' \in \text{NEG-}k(x,y)} f_\theta(x,y')^2 \right] \tag{4.9}$$

$$= \sum_{(x,y)\in D} \left[(1 - \theta_{xy})^2 + \sum_{y' \in \text{NEG-}k(x,y)} \theta_{xy'}^2 \right] \tag{4.10}$$

Now, the $L2$ loss function for amplified negative sampling is

$$L_2(f_\theta, D) = \sum_{(x,y)\in D} \left[(1 - f_\theta(x,y))^2 + \beta \sum_{y' \in \text{NEG-}k(x,y)} f_\theta(x,y')^2 \right] \tag{4.11}$$

Since the only difference from standard negative sampling (4.9) is the coefficient β of the second summation, we can show that

$$\frac{\partial L_2(f_\theta, D)}{\partial \theta_{ab}} \xrightarrow{P} \#(a,b)(-2)(1-\theta_{ab}) + [\#(a) - \#(a,b)]\beta\, k\, p_b\, 2\theta_{ab} \tag{4.12}$$

as $t \to \infty$. By setting $\frac{\partial L_2}{\partial \theta_{ab}} = 0$, we can show that

$$\theta^*_{t,ab} \xrightarrow{P} \frac{\#(a,b)}{\beta\, k\, p_b [\#(a) - \#(a,b)] + \#(a,b)} \qquad \text{as } t \to \infty. \tag{4.13}$$

The proof for other loss functions can be done similarly.

Computational Cost of Amplified Negative Sampling. Note that both the standard negative sampling and our amplified negative sampling are *sampling methods* that are independent of the particular choice of the training method. They simply provide a straightforward recipe for selecting a few negative samples and incorporating them in the computation of the loss function. Therefore, both the standard and the amplified versions use the same training algorithm,[5] making their algorithmic and computational complexity identical. That is, as long as they use the same negative sample set NEG-k, their computational costs are (almost) identical.[6] At the same time, our mathematical analysis indicates that even though the same negative samples are used, the amplified version is likely to produce a significantly more accurate model than the standard version by the factor β. In the later experiment section, we evaluate the validity of this analytical result both in terms of the computational cost and the model accuracy through an extensive set of experiments on real-world datasets.

Amplifying Factor vs Learning Rate. Conceptually, our amplifying factor may look similar to the *learning rate* used for the gradient-descent algorithm; at epoch t, the gradient-descent algorithm updates the parameter θ from the current value θ_t to the new value θ_{t+1} via the the following equation:

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla_\theta L(f_\theta, D), \tag{4.14}$$

where α, the learning rate, controls how quickly and reliably the updates converge. As we can see from Equation 4.13, our amplifying factor β is also multiplied to (a part of) the loss function, so it indeed plays a role very similar to the learning rate. The only difference is that α is multiplied to the *entire* loss function $L(f_\theta, D)$ while β is multiplied to its *negative-sample terms only* $\sum_{y \in \text{NEG-}k(i)} f_\theta(x_i, y)^2$. Interestingly, our analysis and later experiments show that this seemingly minor change leads to an enormous difference in terms of the final trained model.

Optimal Amplifying Factor and Model Quality. The similarity of the learning rate and the amplifying factor raises another interesting question. How big a amplifying factor can we safely use? It is well known that a higher learning rate generally leads to a faster convergence rate initially,

[5] Perhaps, the most popular choice is the stochastic-gradient descent algorithm

[6] The amplified version has the overhead of multiplying β compared to the standard version, but this cost negligible in practice.

but it makes the training process less stable at a later stage. Will using a large amplifying factor lead to similar behavior? Or will it always be better to use a larger amplifying factor? Our analysis indicates that the optimal value of β might be $\beta = \frac{1}{kp_y}$ since this value leads to the MLE model.

The results from our experiments do not provide a single answer to this question. In the experiments conducted with our own code, where we measure the model accuracy in terms of the difference of the trained model from MLE, we observe that using a large amplifying factor always produces a model closer to MLE. It also does not introduce much instability to the training process all the way from 1 through $\beta = \frac{1}{kp_y}$. In the experiments conducted with existing codes for *other downstream tasks*, we observe that using a amplifying factor up to $\beta = 3$ reduces the training time and improves downstream-task performance, but starting from $\beta > 3$, we sometimes observe reduced downstream-task performance. This may be due to the fact that downstream-tasks performance does not necessarily correlate with how well our model estimates the conditional probability $P(y|x)$. Given these two results, we find using a reasonably small amplifying factor, say $\beta = 3$, may be a safe choice in general; it reduces training time significantly and improves downstream-task performance.

In our experiments, we also explored a few other directions, including (1) decaying the amplifying factor over epochs similarly to decaying the learning rate, (2) dynamically setting the amplifying factor based on the "loss value" of the negative sample similarly to the idea of importance sampling and (3) early stopping, where we stop using the amplifying factor after a few epochs. We find that these changes do not introduce a meaningful difference to the results.

4.5 Experiments

The primary goal of this section is to experimentally investigate the following two issues: (1) Does the result of our analysis hold in practice? We want to examine how well our theoretical results match with experiments. We also want to experimentally explore a few questions raised in this paper, including the choice of the optimal amplifying factor and the difference between the learning rate and the amplifying factor. (2) Does amplified negative sampling help other downstream tasks as well? The performance of other downstream tasks may not necessarily depend on

how accurately the trained model captures the conditional probability $P(y|x)$, so we want to experimentally check whether amplified negative sampling has positive effects on other downstream tasks or not.

In the subsection (Model Accuracy and Training Efficiency) 4.5.1, we explore the first issue by measuring the difference between the maximum likelihood estimator (MLE) $\tilde{P}(y|x) = \frac{\#(x,y)}{\#(x)}$ and the models trained with (a) full-gradient training (b) negative sampling and (c) amplified negative sampling. The results of our experiments show that the conclusions of our analysis hold in practice to a surprising degree of accuracy. They also show that the learning rate and the amplifying factor have vastly different effects on the trained model. In the subsection (Experiments on Other Downstream Tasks) 4.5.3, we investigate the second issue by running experiments on three different downstream tasks: word-analogy tasks (Mikolov et al., 2013a), rare-word-similarity tasks (Bojanowski et al., 2017), and graph-node-classification tasks (Grover and Leskovec, 2016a). Here, we observe that amplified negative sampling leads to improved performance on these downstream tasks as well.

In summary, our experimental results strongly indicate that there really is not much downside to using amplified negative sampling; as long as we use a reasonably small amplifying factor, say $\beta = 3$, amplified negative sampling leads to lower training time and higher model accuracy.

4.5.1 Model Accuracy and Training Efficiency

Experimental Settings. In this subsection, we experimentally compare four training algorithms, full-gradient training (*FullGrad*), 5 negative samples (*Neg5*), 5 negative samples with the amplifying factor 3 (*Neg5-Amplify3*), and 15 negative samples (*Neg15*), under three different loss functions, $L1$, $L2$, and cross entropy. For the choice of the hypothesis space $f_\theta(x,y)$ and the training set, we use a setting similar to (Mikolov et al., 2013a). That is, as our hypothesis space we use the skip-gram model of (Mikolov et al., 2013a) with a 100-dimensional hidden layer. As our dataset, we use a subset of Text8 corpus from (Mikolov et al., 2013a) by extracting the first 75,000 words

and applying the same min_count filter of 5 in (Mikolov et al., 2013a).[7] We use the stochastic-gradient descent (SGD) with the batch size of 500 as the training algorithm. All our experiments use the window size 3 and the learning rate 0.025 unless noted otherwise. All other parameter settings are the same as in (Mikolov et al., 2013a). All results reported are the average of three independent runs with identical settings. All codes were implemented using PyTorch v1.0.1.

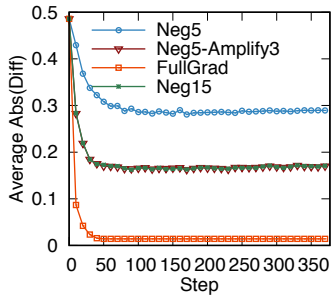


Figure 4.1: Model accuracy

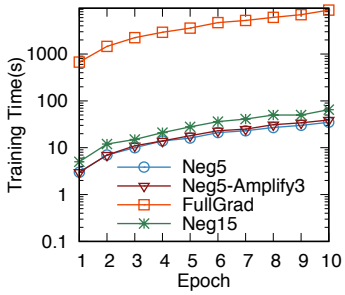


Figure 4.2: Training time.

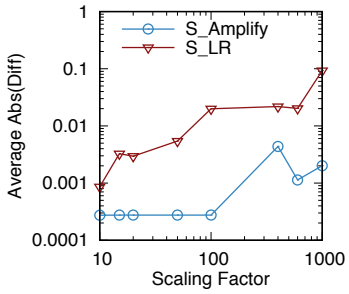


Figure 4.3: Amplifying factor vs learning rate

Model Accuracy and Convergence Rate. In Figure 4.1, we compare the model accuracy and convergence rate when the model is trained with the four algorithms (FullGrad, Neg5, Neg5-Amplify3, Neg15) under the $L2$ loss function.[8] In the graph, the horizontal axis corresponds to the stochastic-gradient-descent training batch steps (with roughly 70 steps corresponding to one training epoch) and the vertical axis corresponds to the average absolute difference between the trained model $f_{\theta^*}(x,y)$ and MLE $\tilde{P}(y|x) = \frac{\#(x,y)}{\#(x)}$, i.e., $\sum_{(x,y)\in D} \frac{1}{|D|} \left| f_{\theta^*}(x,y) - \frac{\#(x,y)}{\#(x)} \right|$.

From the graph, a few things are clear: (1) Full-gradient training converges to MLE. Even at epoch 1 (step 70), the mean absolute difference of *FullGrad* is close to zero, indicating that it converged to MLE. (2) The amplifying factor β effectively "increases" the negative sample size

[7] Using a subset of Text8 here is due to the high training cost of *FullGrad* and our desire to keep the training time at a manageable level. In our next experiments on other downstream tasks, we run report our results from experiments on much larger datasets.

[8] While we performed experiments under all three loss functions, we report the results only from $L2$ here due to space limit. The conclusions from other loss functions are essentially the same.

by the factor β in terms of model accuracy. The mean absolute difference of Neg5-Amplify3 and Neg15 are the same at every training step — they overlap so closely and it is difficult to tell them apart in the graph — indicating that they both converge to the same model at the same rate. This result is what our theoretical analysis predicts: $(k = 15, \beta = 1)$ leads to the same optimal model as $(k = 5, \beta = 3)$. (3) A model trained on larger k approximates MLE better. The mean absolute difference of Neg15 is significantly smaller than that of Neg5.

Training Time and Computational Cost. In Figure 4.2, we compare the training time of the four algorithms. The horizontal axis corresponds to training epochs and the vertical axis corresponds to training time, which roughly captures the computational cost of each algorithm. The vertical axis is logarithmic; since the training time of *FullGrad* is two orders of magnitude larger than others, its result is not visible in the same graph otherwise. From the graph, we again observe what is predicted by our analysis. The training time of Neg5-Amplify3 is practically the same as that of Neg5. That is, Neg5-Amplify3 works almost like Neg5 in terms of its training time and computational cost, but it works almost like Neg15 in terms of its model accuracy and convergence rate! Amplified negative sampling indeed gives the best of both worlds.

Learning Rate vs Amplifying Factor. In Figure 4.3, we compare the effect of using different learning rates and amplifying factors. The graph is from training the model using Neg15 under $L1$ loss. The curve labeled as *S_LR* is obtained by multiplying the default learning rate of 0.025 by a factor between 10 and 1,000. The curve labeled as *S_Amplify* is obtained by including the amplifying factor β between 10 and 1,000. The vertical axis is again in the logarithmic scale due to the high difference between the two curves and represents the model accuracy (the mean absolute difference from MLE) at the given learning rate/amplifying factor. From the graph, we see that changing the learning rate and changing the amplifying factor lead to vastly different results. As we increase the learning rate, the trained model diverges further away from MLE. When we increase the amplifying factor, however, the trained model stays close to MLE all the way through $\beta = 100$. Only after $\beta > 100$, the model starts to diverge and becomes unstable. This result is consistent with our analysis; according to Corollary 1.2, amplifying converges to MLE at $\beta \approx 140$

under the current setting,[9] so its divergence beyond $\beta > 140$ is expected.

4.5.2 Language Model perplexity test

In this section, Amplified Negative Sampling method is tested on the Language Model on Penn Tree Bank(PTB) dataset (Marcus et al., 1993) to demonstrate its performance. We use one of the commonly tested RNN Language model (Zaremba et al., 2014) with the medium regularized settings as in (Blanc and Rendle, 2017). The PTB dataset contains 929K training words, 73k validation and 82k test words. We keep the original settings of the RNN model and set the hidden dimension as 650.

The perplexity result is reported in Figure 4.4 and 4.5 for Neg5 and Neg15 experiment. For each negative sample setting, the amplifying factor is set to 1, 3 and 10 respectively. As the amplifying factor got enlarged, the perplexity got decreased. Every experiment converges in around 20 epochs. As shown in figure 4.4, the amplifying factor can effectively decrease the perplexity compared with other model when only a few number of samples are allowed. Here the result is drawn from (Blanc and Rendle, 2017) to demonstrate the Neg5 result with amplifying factor 10 can compete with (Blanc and Rendle, 2017)'s 10 sample result.

Further experiments are made on the effect of the amplifying factor. As shown in Figure 4.6, when the amplifying factor is increased from 1, the perplexity have a clear decrease until 10 and the perplexity starts to increase when the amplifying factor keeps enlarging. The best model performance will be obtained when the amplifying factor is configured around 10. An explanation based on formula (4.13) is the convergence condition is t(epoch) tends to infinity. However, in experiment, the training process will be stopped as the performance doesn't change anymore or the model got trapped in the saddle point. In this case, the model won't converge to the optimum value and a large amplifying factor will also change the convergence point to a non-optimum point.

[9] Amplified negative sampling converges to MLE when $\beta k p_y = 1$. Given $k = 15$ and the uniform probability $p_y \approx 1/2000$ for this experiment, $\beta k p_y = 1$ at $\beta \approx 140$.

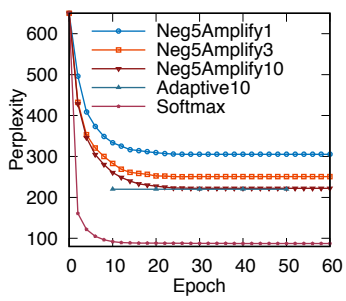


Figure 4.4: PTB: Negative sampling 5.

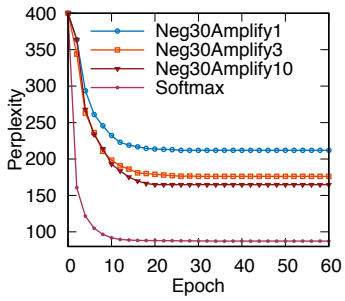


Figure 4.5: PTB: Negative sampling 15.

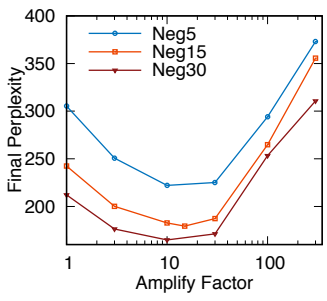


Figure 4.6: Different amplifying factor.

4.5.3 Experiments on Other Downstream Tasks

In the previous set of experiments, we investigated the effect of amplified negative sampling on the trained model in terms of its difference from MLE and a language model. In the next set of experiments, we investigate its effect on three other downstream tasks: (a) word-analogy tasks (b) rare-word-similarity tasks and (c) graph-node-classification tasks. In all our experiments, we compare the results from Neg5, Neg5-Amplify3, and Neg15 at training epoch 3.

Word2vec: Word analogy. This test is conducted with the CBOW model trained on the Text8 corpus (Mikolov et al., 2013c) using the code downloaded from (word2vec, 2019) after we add amplifying code. The performance is evaluated by the 14 word-analogy tasks in (Mikolov et al., 2013c). We use the default parameter settings of the downloaded code.

Fasttext: Rare word similarity. For the word-similarity task, we use the code for Fasttext (Bojanowski et al., 2017) downloaded from (fastText, 2019) using its default settings after the addition of amplifying code. The model is trained on the rare word dataset (RW) (Luong et al., 2013). The effectiveness of a trained model is measured by the Spearman's rank-correlation coefficient (Spearman, 1904) between the human judgment in the dataset and the output from the trained model.

Node2vec: Node classification. For the node classification task, we use the node2vec model

(Grover and Leskovec, 2016a) with the code downloaded from (node2vec, 2019). The dataset used is BlogCatalog (Zafarani and Liu, 2009). The nodes are embedded into a 50-dimensional space and we use 20-80 train-test split. The default parameter setting of the downloaded code is used after the addition of amplifying code.

In Table 4.1 we show the accuracy of Neg5, Neg5-Amplify3, and Neg15 for the first 5 word-analogy semantic tasks of (Mikolov et al., 2013c). From the results, the trend is clear: the performance of Neg5-Amplify3 is higher than Neg5 and is close to Neg15. In Table 4.2, we show the training times of Word2vec, Fasttext, and Node2vec. In all three cases, the training time of Neg5-Amplify is close to Neg5 and is significantly smaller than that of Neg15. In the cases of Fasttext and Word2vec, the difference is by a factor 2. In the case of Node2vec, the difference is much smaller due to the dominance of other training overhead when the absolute training time is small. From our other experiments whose results we could not include here due to space limit, we observe the same general trend: The downstream-task performance of Neg5-Amplify3 is close to Neg15 while its training cost is close to Neg5.

Table 4.1: Word-analogy semantic-task top-1 accuracy.

Task Name	Neg5	Neg5-Amplify3	Neg15
capital-common-countries	38.14	43.02	47.43
capital-world	24.66	29.34	32.14
city-in-state	15.17	17.29	14.93
currency	15.72	22.26	22.21
family	56.86	60.30	64.27

Table 4.2: Training time (seconds).

Model	Neg5	Neg5-Amplify3	Neg15
Fasttext	736	731	1404
Word2vec	17.35	17.37	36.93
Node2vec	3.28	3.28	4.00

4.6 Conclusion

In this paper, we proposed *amplified negative sampling*, a new sample-efficient method for training multi-class classifiers with a large output-class size. Our proposed method was based on our rigorous mathematical analysis. Our extensive set of experiments demonstrated that amplified negative sampling gives us the best of both worlds: It leads to the higher-accuracy model of a larger sample size without paying its high computational cost. Given its simplicity, and experimental effectiveness, we believe our proposed method will be an important extension to the widely-popular technique that results in meaningful improvements in practice.

CHAPTER 5

Expressive Library for Recursive Queries: LLib and LFrame

With the increasingly large volumes of data, there is an urgent need to provide expressive and efficient data analytic tools. Datalog, a declarative logical programming language, has been widely implemented due to its superiority of concise expressing and efficient execution of recursive queries. In this chapter, we focus on extending the expressive power of the language in the data scientists' familiar ways. We propose the LLib, Logical Libraries, to provide a wide range of Datalog algorithms on top of BigDatalog and Apache Spark. LLib encapsulates all the complex logic of algorithms into high-level APIs to simplify the development and provides a unified interface similar to the Spark MLlib. LLib is proposed to be DataFrame-based so that it can flexibly collaborate with other existing Spark operations in MLlib, Spark SQL, etc. Considering the requirements for developing novel recursive applications not contained in LLib, we also show a new LFrame-based Datalog programming interface. The LFrame is an extension to the Spark DataFrame data structure and works in the similar way. Except the normal relational operations like projection, selection, the LFrame could support logical operations like definitions of recursive rules and base relations. To visually show the expressive power of LLib and LFrame, we utilize amounts of running examples during description.

5.1 Introduction

In the era of big data, the demand for flexible analytics of large-scale data has driven the researchers to build various general-purpose user-friendly platforms like Apache Spark (Zaharia et al., 2012), AsterixDB (Alsubaiee et al., 2014), Pig (Olston et al., 2008), Hive (Thusoo et al., 2009), etc. Among these systems, Spark is getting more and more attractive due to its efficient

in-memory computation and abundant APIs (i.e. Spark SQL, GraphX, MLlib and SparkR) for complex analytics to extract the rich information encapsulated in the data.

However, for the iterative applications like identifying the transitive closures or connected components of millions of vertexes, there are no dedicated designs for optimization among recursions in Spark because each iteration is contained in an identical job. For the more advanced recursive analytics, the programming needs deep understanding and extensive knowledge of the platform. To solve these issues, the researchers attempt to implement the Datalog (Consens and Mendelzon, 1990) systems for logical programming.

The Datalog, a well known recursive programming language with superior expressive power, consists of a set of rules and facts. The DeALS project of UCLA (Yang et al.) implements a unified Datalog programming language and provides the parallel evaluation on multicore machines. For the distributed logical computing on clusters, the BigDatalog (Shkapsky et al., 2016) system is further developed on Spark. While considering the SQL programming customs, the Recursive-aggregate-SQL (RaSQL) (Gu et al., 2019) language is proposed as a simple extension of Spark SQL for Datalog.

This torrent of Datalog platforms, however, underscores the need to provide a high-level API to simplify the development or the usage of logical programming as an important step within the data processing pipeline. In those systems, the Datalog applications run independently as one job with input rules and datasets. It is hard for users to take the output of one Datalog program as the input for another Datalog program. Similarly, the collaboration with other Spark APIs like machine learning (MLlib), graph computation (GraphX) or the Spark SQL is inconvenient. Also, as a pre-knowledge, it is necessary for users to learn about Datalog or get used to a SQL extension (RaSQL) when they actually want a one-line library call for a common recursive application like transitive closure or BOM (BOM) as a step of data processing.

In this chapter, we focus on providing an ecosystem for Datalog programming on Spark, which contains LLib, wrapping normal Datalog algorithms with an easily extended interface for contributing new algorithms, and LFrame, the dataframe extension supporting the basic logical operators. The LLib and LFrame are implemented on BigDatalog and Spark. The wide audience of Spark

community should be familiar with the interface of LLib (like Spark MLlib) and LFrame (like Spark DataFrame). With LFrame and LLib, the data scientists could have access to the data manipulated in Datalog applications and easily continuously do other operations like machine learning algorithms of MLlib within one job. This ecosystem makes it possible to ask for help from the existing Datalog algorithms by solely a one-line function calling within a processing pipeline and develop user-defined recursive application as friendly as the Pandas(McKinney et al.)/Spark DataFrame.

Our contributions can be concluded as following:

- Usability. The LLib provides functionality for a wide range of typical Datalog algorithms. It simplifies the development of end-to-end data processing pipelines with the high-level API and avoids the efforts required to learn a new language syntax.
- Interoperability. The LLib is the DataFrame-based API, which takes the DataFrames as the input and also generates DataFrames. This facilitate the collaborations between Datalog applications and Spark MLlib, Spark SQL or GraphX. In addition, we propose the function for the conversion between the DataFrames and LFrames, which enhances the interoperability during data processing.
- Extendability and flexibility. In LLib, there is a template and some utility functions contained to help adding extra Datalog algorithms. We also design a user-defined Datalog function for any possible Datalog applications. The LFrame data structure is associated with various general Datalog operations supporting different recursive algorithms.

The chapter is organized as follows. Section 5.2 reviews the basics about the Datalog language and related platforms including Apache Spark, BigDatalog and RaSQL. Section 5.3 describes the working paradigm of LLib, different categories of algorithms supported by LLib, and utilizes some running examples expressing the user-defined Datalog functions and the collaborations between LLib and other Spark libraries. Section 5.4 presents the conversion between LFrame and DataFrame, the basic unary and N-ary operations supported by LFrame, and some examples developed with LFrame. Section 5.5 draws conclusion and our plans for future work.

5.2 Prelimiaries

5.2.1 Datalog

A Datalog application is comprised of a finite set of rules. Each rule r is formed as H $\leftarrow$ $l_1, l_2, ...l_n$, where H is the head of r, $l_{1..n}$ (the *body*) are *literals* and the $\leftarrow$ means implication. The literals ($l_{1..n}$) are positive or negated atoms. One atom (H or l_i) can be formed as $p(t_1, .., t_k)$, where p is a *predicate* and $(t_1, .., t_k)$ terms can be *variables*, *constants* or *functions*. So, the r is the rule to infer H. However, if r does not have the body $l_1, l_2, ...l_n$, it becomes the *fact*, which corresponds to a tuple in a relation . The comma separating literals means the logical conjunction (AND). To evaluate a Datalog application, we need a *query* mentioning which predicate to evaluate.

Next, we will illustrate a classic example in Datalog, single source shortest path (SSSP), with more terms introduced. The SSSP is to calculates the length of shortest paths from one source vertex to all other vertices in a graph with weighted edges.

Query 1 - Single source shortest path (SSSP).

1 : $database(\{\ warc(A : integer,\ B : integer,\ Cost : integer)\ \}).$

2 : $sp(B,\ mmin < C >) \leftarrow B = \{startvertex\},\ C = 0.$

3 : $sp(B2,\ mmin < C >) \leftarrow sp(B1, C1),\ warc(B1, B2, C2),\ C = C1 + C2.$

4 : $result(B,\ min < C >) \leftarrow sp(B, C).$

5 : $query\ result(T, C).$

As shown in Query 1 line 1, the input relation (*base relation*) is warc with the schema (A:integer, B:integer, Cost:integer). One fact of this relation can be warc(1, 2, 5), which shows the cost from vertex 1 to 2 is 5. The database is a keyword specifying the base relation. In the first rule (line 2), it initializes the shortest distance from the source vertex to itself as 0, where the "{startvetex}" can allow user to input the source vertex ID. The second rule (line 3) recursively produces all the minimum distances for all the possible paths from source node to another node. The monotonic aggregate, mmin is utilized, which allows the aggregation inside the recursion when set containment

is satisfied. The mmin will get new lower value with a larger set of possible paths. And a normal aggregate min is finally exploited (line 4) to obtain the minimum cost path for each source vertex. The fifth line denotes the predicate (*relation*) result(T,C) will be evaluated and become the output of the application.

5.2.2 Related Platforms: Apache Spark, BigDatalog and RaSQL

Apache Spark. Apache Spark is a DISC system with various modules like Spark SQL (Armbrust et al., 2015), Spark Streaming (Zaharia et al., 2013), MLlib (Meng et al., 2016) and GraphX (Gonzalez et al., 2014) to support analytics on structured data, streaming applications, machine learing algorithms and graph computation tasks respectively. All the Spark applications are eventually represented by a series of transformations and actions on Resilient Distributed Datasets (RDDs) (Dean and Ghemawat, 2004), the main abstraction provided by Spark. The operators like groupBy, filter are lazily evaluated until the output actions like count trigger evaluations on RDDs. In this way, before execution, the Spark Optimizer can design a better physical plan by avoiding some duplicate computation or pipelining operations. RDDs in Spark are actually Python or Java objects stored in memory and can be processed in parallel. The RDD is fault-tolerant due to its lineage. The lineage graph of RDDs records the transformations applied to them. When records get lost, it is easy to recover them by rerunning through the lineage graph. The Spark has attracted wide audiences from database community due to its high usability and performance.

BigDatalog. With the popularity of Spark and the benefits for recursive query evaluation and optimization brought by Datalog, the new requirements have been re-emerged to support the Datalog development on Spark. As far as we can find, the BigDatalog (Shkapsky et al., 2016) is the first one implementing the DeALS, a Datalog platform, on Spark. It supports the execution of recursive operations on multi-core machines and clusters. The BigDatalog also proposes some optimizations on physical planning for recursive queries to obtain the orders of magnitude speed-up.

RaSQL. The BigDatalog enables development of Datalog on Spark, but users should get used to the Datalog syntax. With the continuing popularity of SQL, it is beneficial to design a language similar to SQL for Datalog queries. The RaSQL (Gu et al., 2019) proposes a new language follow-

ing and extending SQL standards, and utilizes some novel optimizations on fixpoint operators for the Datalog platform built on Apache Spark.

Developing a Datalog application on BigDatalog or RaSQL requires a query file, a structured data file, and a standard execution program, which takes the query and data files with some arguments for execution. Users are required to learn Datalog or RaSQL syntax and it is not so interoperable between one Datalog application and another Spark MLlib, Spark SQL or Datalog application. In this chapter, we would like to introduce the LLib and LFrame to solve those issues as described above.

5.2.3 Spark MLlib and DataFrame

The MLlib is the machine learning library of Spark. It consists of the normal classification, regression algorithms. Users can flexibly build a pipeline of a sequence of algorithms to process data with the abundant libraries in MLlib. The pipeline could span data cleaning, model initialization, training, prediction, and evaluation.

The DataFrame is a popular facility for data scientists and has been supported by various trendy data analytics platforms and languages like Spark, Pandas (McKinney et al., 2010), R, etc. It is utilized as a data structure and the data stored in it is organized in rows and columns like a table in Excel. This increases the visibility of development for non-expert users. The supported operations for the DataFrame are similar to the relational algebra, but are exposed as pre-defined functions like libraries of Java, Python, etc.

The LLib and LFrame proposed in this chapter provide a similar interface to Spark MLlib and DataFrame separately. The LLib contains the libraries for common Datalog algorithms and the LFrame is an extension to DataFrame encapsulating the normal Datalog operations like rule definition, registration of the base relations.

5.3 LLib

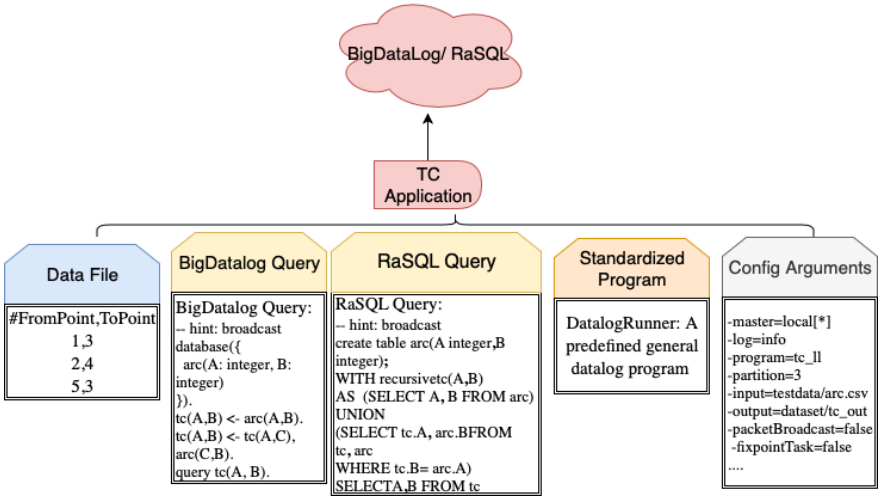
In this section, we discuss LLib, which works in the similar way to MLlib to develop Datalog applications. It does not require users to be familiar with logical programming. LLib contains a wide spectrum of recursive applications including graph algorithms (Transitive Closure, i.e. *TC*), temporal database queries (Interval Coalesce), financial applications (MLM), machine learning algorithms (Logistic Regression), etc. The data analysts could easily take one Datalog algorithm as one step within their complex data processing process, which is more flexible than the previous distributed Datalog programming interfaces.

5.3.1 LLib Working Paradigm

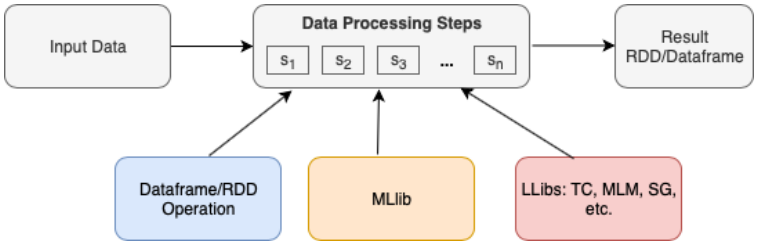
The LLib grants the access to the processed data to users through the Spark DataFrame, which greatly reduces the required boilerplate code and improves the readability. In this part, we would like to firstly provide a high level picture of the difference while working on the two popular Spark-based distributed Datalog platforms (BigDatalog and RaSQL) and our LLib.

As shown in Figure 5.1 (a) about the TC application, the previous Spark-based Datalog program is made up of (1) the source data file(s), which should follow a fixed schema like the names of fields for the corresponding application or the comma delimiter, (2) a query file following the RaSQL or BigDatalog syntax, (3) a standardized program to accept the source data file(s) and query file for recursive computation, and (4) a set of arguments to guide the program execution. This is a general-purpose design for all potential applications that can be expressed by a combination of rules running on several data files, but it causes a troublesome programming considering the following aspects.

- *Learning new language.* To develop with those platforms, users are required to get used to the syntax of Datalog or an extension of SQL while they actually want to use a well-wrapped typical Datalog application as a step of their whole data processing pipeline.
- *Data transfer.* For each application on the previous platforms, the required data need to follow a fixed schema. It costs much for the schema conversion.



(a) (a) The composition of TC application on custom Datalog platforms.



(b) (b) LLib paradigm

Figure 5.1: Working paradigm comparison between custom distributed Spark-Based Datalog platforms and the LLib.

- *Isolation from other applications.* The development mode of the original logical programming works in a way similar to a black box. Users provide data and rules to standardized program and get results. They only have access to the generated file and can hardly involve other preprocessing or subsequent processing.

To overcome those drawbacks, we develop the DataFrame-based LLib as shown in Figure 5.1 (b), which works in a similar way to Spark MLlib. Within a LLib-based application, there can be many steps made up of the permutations of regular DataFrame operations like selection or projection, MLlib's ML algorithms or featurizations and LLib's Datalog algorithms. Later in this section, we describe the underlying architecture and working pipeline of LLib through a small running example that shows how to evaluate the transitive closure of a directed acyclic graph (DAG).

5.3.1.1 Working session and acquiring data

In LLib, the first step is to construct a working environment for the Datalog queries and libraries. We respect the customs of building Spark Session and exploit the similar way as following (with Transitive Closure, i.e. TC as a running example):

Example 1.1 - Transitive closure with LLIb: Working session.

$$session = LLibSession.builder().appName(\text{“}TC\text{”}).master(\text{“}local[*]\text{”}).getOrCreate()$$

$$schema = StructType(List(StructField(\text{“}Point1\text{”}, IntegerType, true),$$

$$StructField(\text{“}Point2\text{”}, IntegerType, true)))$$

$$df = session.read.format(\text{“}csv\text{”}).option(\text{“}header\text{”}, \text{“}false\text{”}).schema(schema).load(\text{“}arc.csv\text{”})$$

The *LLibSession* synthesizes the Spark environment and the special designs for logical programming. Within the same session, users are free to utilize the existing Spark libraries like MLlib or Datalog libraries i.e. LLib. As shown in the third line, the source data can be loaded via the loading function in Spark to a DataFrame *df*.

5.3.1.2 Initializing an executable object of LLib and mapping the schema

Initialization: In LLib, the pipeline of the data processing for a typical application is compressed to an object, which you can initialize and set parameters on to process your data. In this example, we initialize a transitive closure object tc with the TC library, which is pre-defined and included in the LLib. Then, we can set the property by built-in functions.

Example 1.2 - Transitive closure with LLIb: Initializing TC.

$$import\ edu.ucla.cs.wis.bigdatalog.spark.LLib.TC$$

$$val\ tc = new\ TC()$$

$$tc.setDirection(FromCol = "Node1", ToCol = "Node2")$$

Schema Mapping: Among the built-in functions, all the libraries are required to contain one function called setDirection for schema mapping. This is necessary because we want a more general design that can accept DataFrame with various schemas as the input. DataFrame may call an attribute with an arbitrary name but we need to know the corresponding relationship between its attributes and the attribute names mentioned in our library's rules. The Query 2 is a set of Datalog rules for the transitive closure. We have two attributes From and To in the arc table in the computing logic. The two attributes can be called differently like (Node1, Node2) as shown in the df initialized in Example 1.1. The mapping between (From, To) and (Node1, Node2) can be provided by the setDirection function as shown in the third line of Example 1.2 setDirection (FromCol = "Node1", ToCol = "Node2"). If there is a long mapping list, we can easily extend the current design by storing the mapping information within a hash table as the transferred parameter to the function.

Query 2 - Transitive closure.

$$database(arc(From : integer, To : integer)).$$

$$tc(From, To) \leftarrow arc(From, To).$$

$$tc(From, To) \leftarrow tc(From, Tmp), arc(Tmp, To).$$

$$query\ tc(From, To).$$

While processing the data with our library, we need the mapping information. But after pro-

cessing, the output data's format should be consistent. There is one mechanism to rollback the schema. At the beginning of data processing, we store the schema of input data. Then, in the end, we could use the pre-stored schema to recover.

When the amount of required attributes by the library matches the amount of columns in the input DataFrame, we could make full use of the dataset. However, when the dataset contains more attributes than the required, it should be firstly pruned before analysis. As for the condition that the required attributes not included or the dataset contains fewer attributes than the required, the exception will be raised.

5.3.1.3 Execution and persistence

With the executable object and imported data, the execution and persistence can be merely a one-line calling of a pre-defined function (run, genDF or genRDD) in LLib. These three functions can support the basic requirements to operate data, store the result to a variable or a file.

Example 1.3 - Transitive closure with LLIb: Execution and persistence.

$$tc.run(df, output = "File", session)$$

$$val\ dfNew = tc.genDF(df, session)$$

$$val\ rddNew = tc.genRDD(df, session)$$

Execute and persist: As shown in Example 1.3, the function *run* is to run the logical programming and persist the result directly to the target address. In the first line, the tc operates df and store the result to File.

Execute and return a DataFrame/RDD: If the user would like to keep operating the data, they would prefer to get the output data in a form of DataFrame or resilient distributed dataset (RDD) instead of persistence to files. As shown in the second and third lines of the Example 1.3, the dfNew and rddNew will be the outputs and can be further utilized for the next steps of data processing.

These three functions are implemented for each library of LLib. They expect the input data (df) and the environment (session) as the parameters. The session information is also needed because during execution, we want the program running within an environment with capability to support the logical programming.

5.3.1.4 Operate on multiple datasets

The previous showed TC example operates on only one relation, however there are many applications requiring more than one relation, which causes some changes to the pipeline. We illustrate the Multi level Market (MLM) Bonus as a typical Datalog application (MLM, 2008) acquiring more than one datasets. The application is to calculate the bonus for members of a hierarchical structural Multi Level Marketing organization. In the organization, the new members are recruited by and get products from the old members (sponsors). One member's bonus is based on his own sales and a ratio of the sales from the people directly or indirectly recruited by him. The scale of the ratio is a pre-defined rate.

There are two relations in the MLM Bonus, including the *sponsor* and *sales*. The sponsor relation describes the recruiting relationship among the members, while the sales relation records the profits for each member. In the Datalog rules, the base case is to calculate the member's bonus by the sales table. And the recursive rule is to calculate the bonus based on the basic profits and the profits derived from the downline members.

With the help of LLib, users could implement the MLM Bonus by a pre-defined MLM class in LLib ignoring the complex logic. The program can be as easy as the follows.

Example 2 -LLib with more than one input relation: Multi level Market Bonus.

$$val\ MLM = new\ MLM()$$

$$MLM.setDirection(MCol = "M", ProfitCol = "P")$$

$$MLM.setSecDirection(MCol = "M1", M2Col = "M2")$$

$$MLM.run(Array(dfSales, dfSponsor), output = "resMLM", session)$$

Suppose we already have the two relations stored in dfSales and dfSponsor. In the first line, we build an executable object of MLM. Then, we set the schema mappings for two relations in the next two lines. To operate the data and persist to resMLM file, we use the forth line calling the run function. The run function of TC application only expects one relation as the input. While dealing with multiple relations, we aggregate the relations in an array as the input. The schema mapping is implemented by adding a new function for the second relation. However, it is possible to maintain the schema mapping for each relation in a hash function (h_r) and use another hash function with the relation's name as the key and the schema mapping information (the hash function, h_r) as the value.

5.3.2 LLib Categories

The supported common Datalog algorithms and utilities in LLib can be categorized into five groups. We briefly introduce each group and the typical algorithms.

Graph algorithms: The most common cases of recursive computations belong to the graph algorithms. In LLib, the typical graph algorithms are Single Source Shortest Path (SSSP), Transitive Closure (TC), Connected Components (CC), and Count Paths (CP). The SSSP computes the shortest paths from a specific source vertex to all other nodes in a graph with weighted edges. The length of paths are computed iteratively and there is a min aggregation to choose the shortest one. The usage of the SSSP from LLib is as follows.

Example 3 - A graph algorithm (SSSP) supported by LLIb.

$$val\ SSSP = new\ SSSP()$$

$$SSSP.setFromVertex(vertexID = 1)$$

$$SSSP.setDirection(fromCol = "From", toCol = "To", costCol = "Cost")$$

$$SSSP.run(df, output = "resSSSP", session)$$

The program differs from the TC example on the second line, where the source vertex is set. The remaining part of the program is identical to TC, which has been fully explained previously.

CC is to identify the connected components of a graph by attaching and updating each node with a group ID in iterations. The group ID is set each time as the minimum node ID among the nearby nodes. The nodes in a connected component will finally share the same group ID and we just need to compute the number of distinct group IDs. In CP algorithm, we obtain the number of paths from one node to the other nodes in a graph. The reachability is transferred iteratively along the edges of graph. The development with CC and CP library will be similar to the TC library, where only the schema mapping is required before execution.

Machine learning algorithms: Another series of algorithms, which can be expressed by Datalog queries, are machine learning algorithms. Previously, the development for a simple algorithm like Linear Regression or Logistic Regression needs the intensive understanding of the Datalog. With inspirations from Spark MLlib, we provide a more declarative way to develop the machine learning algorithms (including logistic regression, linear regression, SVM, etc.) on the Datalog platform. To illustrate the development with our API, we show a running example exploiting the logistic regression class.

Example 4 - Developing machine learning algorithms with LLib.

$// \ Import \ data.$

$var \ Vschema = StructType(List(StructField(\text{"}Id\text{"}, IntegerType, true),$

$StructField(\text{"}C\text{"}, IntegerType, true), StructField(\text{"}V\text{"}, DoubleType, true),$

$StructField(\text{"}Y\text{"}, IntegerType, true)))$

$var \ df = spark.read.format(\text{"}csv\text{"}).option(\text{"}header\text{"}, \text{"}false\text{"}).schema(Vschema).load(\text{"}dataV\text{"})$

$// \ Training \ on \ the \ input \ relation \ df.$

$import \ edu.ucla.cs.wis.bigdatalog.spark.LLib.DL\_LogisticRegression$

$val \ lr = new \ DL\_LogisticRegression().setMaxIter(10)$

$val \ lrModel = lr.run(df, session)$

The running environment of our program is still the LLibSession. Then, we can load a pre-processed training dataset stored in a verticalized view with *Vschema* (Id, F, V, T), where the Id is the identification of each training instance; the F is the feature's ID; the V is the feature's value and the T is the label. The verticalized format is beneficial for sparse training data and the transactions of Datalog rules. After importing the required training functions for logistic regression, we could build an executable training object, *lr*. The *lr* wraps all the logical rules and some required relations (e.g. parameters with default value 0) of the Datalog implementation for the logistic regression training. While initializing the lr, users can exploit the built-in functions to set the properties like the limit of iterations, the intial value of parameters, etc. During training (*run*), the session information of the Datalog environment is also required.

For comparison, we leverage the Spark MLlib for the above example. The development will become the following, which looks very similar. The differences are: 1)The input data (*dataS*) do not need verticalization and the schema is not the *Vschema*; 2) An assembler is leveraged to specify

the attributes belonging to the features, while in LLib, the verticalized relation is self-explanatory.; 3) The session information is not required during training, while in LLib, the session should be one input parameter for training. Even though the differences exist, the expressing with Spark MLlib and LLib are extraordinarily similar and both user-friendly.

Example 5 - Machine learning algorithms training with Spark MLlib.

```
// Import data.
var schema = StructType(List(StructField("X1",IntegerType,true),
StructField("X2",IntegerType,true),StructField("X3",DoubleType,true),
StructField("label",IntegerType,true)))
var df = spark.read.format("csv").option("header","false").schema(schema).load("dataS")

// Training on the input relation df.
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.VectorAssembler
val assembler = new VectorAssembler().setInputCols(Array("X1","X2","X3"))
.setOutputCol("features")
val lr = new LogisticRegression().setMaxIter(10)
val pipeline = new Pipeline().setStages(Array(assembler,lr))
val lrModel = pipeline.fit(df)
```

Temporal database queries: LLib also supports the transactions on data related to time. For example, in the Interval Coalesce, the goal is to find the smallest set of intervals to cover the input intervals. In Datalog, users are supposed to exploit one rule to find the start points (S) of intervals

which are not inside the other intervals and another rule to recursively extend the right side of intervals, whose left side belonging to S. However, in the LLib, the development will be as simple as follows with *df* (S and E column represents the start point and end point) as the input intervals and the *session* built by LLibSession.

Example 6 - Temporal database query with LLib.

$$val\ Coalesce = new\ Coalesce()$$

$$Coalesce.setDirection(startCol = "S", endCol = "E")$$

$$Coalesce.run(df, output = "res", session)$$

Financial applications: Except the MLM Bonus mentioned in Section 5.3.1.4, we also support the other financial applications like Bill of Matreials (BOM) (BOM) and Management. In BOM, one input relation is *assembling* table recording the assembling relation between one item i (or a part of i) and its immediate subparts. Another input is *basic* table recording the time cost to deliver the basic parts. The task can be calculating the days required to get the i ready. We can obtain the required time for each subpart of i and use the maximum time as the result. The calculating of time for each subpart of i should be execute recursively. In Management, the input table stores the manager and the employees managed by them, and the task is to calculate the total account of employees directly or indirectly managed by one manager. This also requires recursive execution. In LLib, we provide a class wrapping the execution of the BOM or Management and the usage will be similar to the example introduced in Section 5.3.1.

Other applications: Some other important and classical recursive query applications are also contained in LLib, like same-generation (SG), which identifies pairs of humans with the same hop distance to a common ancestor. The working paradigm with them will be similar to the Spark MLlib and same as the examples in Section 5.3.1. Here, we will not make repetitive illustrations.

5.3.3 Extension of LLib

LLib also provides a unified template, *TempLib*, to follow when contributing a new algorithm to the LLib. All existing algorithms are developed based on the TempLib. In the TempLib, the provided non-abstract utility functions (like arguments parsing function) can be easily exploited while developing a new algorithm. The template as well contains some requirements needed to be followed. The implementation should at least include three functions (run, genDF and genRDD) for execution and persistence. With the template, the implementation becomes quite uncomplicated. As long as users have the Datalog rules for the algorithm, they can add their own algorithm by doing some minor changes to the existing codes for algorithms like TC. The changes may include a different schema mapping function, or a new function to set an input parameter for your application like the function to set the starting point in the SSSP.

5.3.4 Collaboration with other applications

The input and output relations in LLib can be both DataFrames, which makes it possible to add a preprocessing providing the input DataFrames or a subsequent processing which takes the DataFrame generated from LLib. The LLib allows users to exploit the Datalog application as a simple step at any place of their processing pipelines. In this section, we illustrate a concrete example showing the collaborations between LLib and other applications like Spark MLlib or Spark DataFrame Operations. The showed LLib application is the BOM application mentioned previously, and we want to use the linear regression library from MLlib to get the relation between the assembling days and the price of one part. To save the space, we do not show the process to build the session (LLibSession), and load three input relations (dfAssbl, dfBasic, partPrice). The dfAssbl and dfBasic are two relations in the BOM application storing the assembling relations among parts and the delivery time for basic parts. The partPrice stores the price for each part. With Delivery class from LLib, we could get the *dfRes* storing the cost time for each part. Joining the result relation with the partPrice on the part column will generate a new relation with columns of the day and price. The Linear Regression function from MLlib could train on the result DataFrame. In this example, we can find the LLib is able to collaborate fluently with other operations in Spark

like MLlib (LinearRegression), DataFrame Operations (join).

Example 7 - Collaboration between LLib with other Spark libraries.

$$dfAssbl = dfAssbl.where(\text{"}Sub < 7\text{"})$$

$$val\ Delivery = new\ Delivery()$$

$$Delivery.setDirection(PartCol = \text{"}Part\text{"}, SubCol = \text{"}Sub\text{"})$$

$$Delivery.setSecDirection(PartCol = \text{"}Part\text{"}, DaysCol = \text{"}Days\text{"})$$

$$val\ dfRes = Delivery.genDF(Array(dfAssbl, dfBasic), session)$$

$$val\ dfRes = dfRes.join(pricePart)$$

$$val\ parsedData = dfRes.rdd.map(row =>$$

$$LabeledPoint(row.getAs[Int](\text{"}Days\text{"}), Vectors.dense((row.getAs[Int](\text{"}Price\text{"})))))$$

$$val\ numIterations = 10$$

$$val\ stepSize = 0.00000001$$

$$val\ model = LinearRegressionWithSGD.train(parsedData, numIterations, stepSize)$$

$$model.save(session.sparkContext, \text{"}scalaLinearRegressionWithSGDModel\text{"})$$

5.3.5 Multiple Datalog applications

A Datalog application can not only work with other Spark libraries but also work with other Datalog applications. The working paradigms of multiple Datalog applications can be in sequential or orthogonal ordering. When they work in sequence, one Datalog application will take another one's result dataset as the input. If they work in an orthogonal ordering, one application's input will be irrelevant with the data processed in other Datalog applications. Both the working paradigms can work with the LLib library, as the library operate on the DataFrame and it is simple to transfer

data among applications. This cannot be achieved with the previous Datalog platforms, for they take each Datlog application as a single job to execute, which can hardly collaborate.

5.3.6 User Defined Datalog Function

Although we have provided all the common Datalog functions in our mind, users may also want to define their own function so that they can use it for futural development. To serve the user-defined datalog function (UDDF), we provide two general classes SingleTableUDDF and MultipleTablesUDDF, which wrap the necessary components to execute the Datalog queries and persist the results to files or a new DataFrame. The SingleTable is exploited when the UDDF has one input relation while the MultipleTables is exploited when the UUDF has more than one input relations. While utilizing the two classes to specify new Datalog applications, the only required information include Datalog rules and the schema of the basic table (or input table).

We will show the UDDF with a running example as follows. In the Example 8, we suppose user want to define their SG and Delivery application. Suppose the *df* stores the parent-children relations among people. The *dfAssbl*, *dfBasic* stores the assembling relation among parts and the cost days to deliver the basic parts. The session is a LLibSession. The process to create the df, dfAssbl, dfBasic and session is omitted. For the UDDF of SG, we build an executable object by the SinlgeTableUDDF class and specify the input relation's schema by registerDatalogTable function. The rules are provided afterwards, which contain the input relation. During execution, the input table (df) and the query to trigger the table persistence should be provided. Similarly, for the Delivery, expecting dfAssbl and dfBasic as the input datasets, users are required to build an executable object and register two relations separated by the comma signal. The exeuction can be trigger once the rules and final query are provided. The two executable object (UDDFSG, UDDFDelivery) are reusable for the future development. If necessary, users can build as many UDDFs as they want in a single program.

Example 8 - User-defined Datalog functions.

$// \ Datalog \ application \ with \ one \ table.$

$val \ UDDFSG = new \ SingleTableUDDF()$

$UDDFSG.registerDatalogTable(\text{"}rel(Parent : integer, Child : integer)\text{"})$

$UDDFSG.rule(\text{"}sg(X,Y) \leftarrow rel(Parent,X), rel(Parent,Y), X = Y.sg(X,Y) \leftarrow rel(A,X),$

$sg(A,B), rel(B,Y).\text{"})$

$UDDFSG.run(sourceDataFrame = df, output = \text{"}UDDFSG\text{"}, session, query = \text{"}sg(X,Y).\text{"})$

$// \ Datalog \ application \ with \ multiple \ tables.$

$val \ UDDFDelivery = new \ MultipleTablesUDDF()$

$UDDFDelivery.registerDatalogTables(\text{"}assbl(Part : integer, Sub : integer),$

$basic(Part : integer, Days : integer)\text{"})$

$UDDFDelivery.rule(\text{"}actualdays(Part, mmax < Days >) \leftarrow basic(Part, Days).$

$actualdays(Part, mmax < Days >) \leftarrow assbl(Part, Sub), actualdays(Sub, Days).\text{"})$

$UDDFDelivery.run(Array(dfAssbl, dfBasic), output = \text{"}UDDFDelivery\text{"}, session,$

$query = \text{"}actualdays(P,D).\text{"})$

5.4 LFrame

DataFrame, a widely used data structure, is always supported in different data analytic facilities like Spark SQL, R, Python, but not in Datalog platforms. In this section, we provide a data structure

LFrame, wrapping both the Datalog transactions and the common DataFrame transactions. The rest is organized as follows: Section 5.4.1 describes how to convert a normal DataFrame variable to the LFrame variable. Section 5.4.2 discusses unary operations that operate on one LFrame, while the Section 5.4.3 discusses the N-ary operations using with more than one LFrame. In each of the Section 5.4.2 and Section 5.4.3, we describe one concrete example to illustrate the development with LFrame.

5.4.1 Conversion from DataFrame to LFrame

To build an LFrame variable, we show the process by the following example. The entry point of the application using LFrame is the LLibSession, same as the one in LLib, which makes it possible to use both LLib and LFrame within one execution environment. The session can load the data to a DataFrame variable df, but the df cannot execute any logical transaction. To construct an LFrame variable with the df, a built-in function wrapperDF in LLibSession can be utilized with the session and df as the inputs. The session is transferred to provide the Datalog running environment and the df is to supply the dataset with schema. The constructed LFrame variable, lframe, could support various Datalog operations like specifying the rules, input tables, persisting the result to a file, etc. These operations are categorised to unary operations and N-ary operations and introduced later.

Example 9.1 - Development with LFrame: Conversion from DataFrame to LFrame.

$$val\ session = LLibSession.builder().appName(\text{"}LFrame\text{"}).master(\text{"}local[*]\text{"}).getOrCreate()$$

$$var\ schema = StructType(List(StructField(\text{"}Parent\text{"}, IntegerType, true),$$

$$StructField(\text{"}Child\text{"}, IntegerType, true)))$$

$$var\ df = session.read.format(\text{"}csv\text{"}).option(\text{"}header\text{"}, \text{"}false\text{"}).schema(schema).load(\text{"}sg\text{"})$$

$$var\ lframe = LLibSession.wrapperDF(df, session)$$

5.4.2 LFrame: Unary Operation

The unary operation only involves one LFrame object each time. With the lframe in Sec. 5.4.1, we implement the SG application as following to show the typical provided unary operations.

Example 9.2 - Development with LFrame: Unary operations with SG application.

1 : $// \ Datalog \ operations \ for \ SG \ application.$

2 : $lframe = lframe.registerCurLFrame(\text{"}lframe1(Parent : integer, Chile : integer)\text{"})$

3 : $lframe = lframe.rules(\text{"}sg(X,Y) \leftarrow lframe1(Parent,X), lframe1(Parent,Y), X = Y\text{"})$

4 : $lframe = lframe.rules(\text{"}sg(X,Y) \leftarrow rel(A,X), sg(A,B), rel(B,Y)\text{"})$

5 : $lframe = lframe.rules(\text{"}sg(X,Y) \leftarrow sg(X,Y)\text{"})$

6 : $lframe = lframe.delRule(3)$

7 : $lframe = lframe.query(\text{"}sg(X,Y)\text{"})$

8 :

9 : $// \ Execution \ and \ persistence.$

10 : $lframe.run(output = \text{"}SG\text{"})$

11 : $val \ dfRes = lframe.genDF()$

12 : $val \ rddRes = lframe.genRDD()$

13 :

14 : $// \ Normal \ DataFrame \ operations.$

15 : $lframe.nonRecursive().where(\text{"}X < 10\text{"}).select(\text{"}X\text{"}).collect().foreach(println)$

There are six categories of unary operations included in the above example.

- **Registering one LFrame as a base relation.** In the line 2, the registerCurLFrame is to register the lframe as a base relation so that the Datalog rules can utilize it as a given dataset. While registering, users are free to rename the dataset's columns and the LFrame will map the columns according to the order of appearance.

- **Appending rules.** The main body (line 3 to 5) of a Datalog program is a finite set of rules. To state the rules, the function called "rules" can be exploited. Whenever the function is utilized, a new rule will be appended to the existing rule sets owned by the lframe.

- **Removing rules.** If one rule is wrongly appended, it can be removed by the delRule (line 6) function using its index. The provided index variable can be also an array, which removes more than one rule.

- **Specifying output relation.** To evaluate the application, the query function (line 7) assists to point out the output relation, which is the sg relation in the SG application. The generated result LFrame will store the output dataset in the schema specified in query, (X, Y).

- **Execution and persistence.** In DataFrame, the evaluations are lazy. Similarly, the LFrame's evaluation is lazy until the run action is triggered. As shown in the second part of the code (line 9 to 12), the run function will store the result relation to a file and if a user want to restore the result to a DataFrame or RDD, the genDF or genRDD function can be considered.

- **Non-recursive transactions.** The design of LFrame is to wrap both the Datalog transactions and the normal DataFrame transactions. When a normal DataFrame transaction is required, like the line 15 shows, the nonRecusive function will convert the execution to the normal DataFrame execution and accept all the DataFrame operators afterwards.

Compare the LFrame with DataFrame, we can find they work in analogous ways but LFrame can support more operations. A more compact way to express the transactions from line 2 to 7 is to list them back to the lframe one by one like the line 15.

5.4.3 LFrame: N-ary Operation

In the previous SG example, only one relation is contained in the Datalog application, while in this section, we illustrate the transaction (registerMoreDFs) embroiling more relations like the join operator of the relational database or DataFrame. We adopt the Delivery Datalog application as a running example, for it contains both the assembly and the basic relations. The assembly relation stores assembling relation among parts and the delivery days of basic parts are stored in the basic relation. The expected output should be the time spent for each part.

Example 9.3 - Development with LFrame: *N*-ary operations with Delivery application.

```
1 :  var schemaAssbl = StructType(List(StructField("Part", IntegerType, true),
2 :    StructField("Sub", IntegerType, true)))
3 :  var dfAssbl = session.read.format("csv").option("header","false").
4 :    schema(schemaAssbl).load("assbl")
5 :  var schemaBasic = StructType(List(StructField("Part", IntegerType, true),
6 :    StructField("Days", IntegerType, true)))
7 :  var dfBasic = session.read.format("csv").
8 :    option("header","false").schema(schemaBasic).load("basic.csv")
9 :
10 :  // Datalog operations for Delivery application.
11 :  var lfAssembly = DatalogLibSession.wrapperDF(dfAssbl, session)
12 :  lfAssembly = lfAssembly.registerCurDF("assbl(Part : integer, Sub : integer)")
13 :  lfAssembly = lfAssembly.registerMoreDFs(otherDF = Array(dfBasic),
14 :    registers = Array("basic(Part : integer, Days : integer)"))
15 :  lfAssembly = lfAssembly.rules("actualdays(Part, mmax < Days >)
16 :    ← basic(Part, Days)")
17 :  lfAssembly = lfAssembly.rules("actualdays(Part, mmax < Days >)
18 :    ← assbl(Part, Sub), actualdays(Sub, Days)")
19 :  lfAssembly = lfAssembly.query("actualdays(P, D)")
20 :  lfAssembly.run(output = "Delivery1")
```

In the example, we do not show the process of establishing the execution environment (session) to save space. We firstly load two relations to DataFrame objects dfAssbl and dfBasic (line 1 to 8). Then, the dfAssbl is converted to an LFrame variable with Datalog functions encapsulated. To exploit the relation as the base relation, the registerCurDF function is exploited in line 12.

The above mentioned steps solely rely on one relation, however, the basic relation is required in the Delivery application. To involve more datasets, the registerMoreDFs function (line 13) need to be utilized. The input parameters include an array of DataFrames to be registered and another array of schemas exploited to register them. The two arrays are ordered and have one-to-one correspondence. Thereupon, the rules can take these DataFrames as the base relations to use. To construct the rule sets, the rules function is adopted multiple times. Eventually, the output relation specified by query function will be stored to the address contained in the run function (line 19 to 20).

5.5 Conclusion

In this chapter, we have shown the LLib encapsulating typical Datalog algorithms, which follows the data scientists' customs and requires less logical programming expertise. For the possible extensions of LLib, we provide not only a unified template for normalizing the contribution, but also some utilities to simplify the extending process. With some running examples, we demonstrate the benefit of designing the LLib as the DataFrame-based API is that the Datalog algorithm can become a single step within the data processing pipelines collaborating with other Spark operations. For the audience who has more logical programming experience and would like to design their own applications, we also provide an LFrame API containing both logical and DataFrame-based operations. A simple conversion approach between the DataFrame and LFrame is also given for a simplified developing. Moving forward, we have plenty of new algorithms and logical oeprations to add for the LLib and LFrame separately. We plan to make the LLib and LFrame more general for different languages like Python, R. We also would like to implement the two interfaces on other distributed platforms or crossing various platforms in a way like a polystore system (Duggan et al., 2015)

CHAPTER 6

Conclusion

Bibliography

1st deep learning and security workshop. https://www.ieee-security.org/TC/SPW2018/DLS/. 20

Recursion example: bill of materials. https://www.ibm.com/support/knowledgecenter/SSAE4W˙9.6.0/db2/rbafzrecurse.htm. 69, 85

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283. 2

Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V., Bu, Y., Carey, M., Cetindil, I., Cheelangi, M., Faraaz, K., et al. (2014). Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14):1905–1916. 68

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. (2016). Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. 2

arbor. Arbor confirms 1.7Tbps DDoS attack. https://asert.arbornetworks.com/netscout-arbor-confirms-1-7-tbps-ddos-attack-terabit-attack-era-upon-us/. 18

Ark. Ark ipv4 routed topology dataset. http://www.caida.org/data/active/ipv4˙routed˙24˙topology˙dataset.xml. 13, 21, 27, 35

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. 72

Bai, Y., Goldman, S., and Zhang, L. (2017). Tapas: Two-pass approximate adaptive sampling for softmax. *arXiv preprint arXiv:1707.03073*. 48, 51

Baker, F. and Savola, P. (2004). Ingress filtering for multihomed networks. BCP 84, RFC Editor. http://www.rfc-editor.org/rfc/rfc3704.txt.

Bakhtiary, A. H., Lapedriza, A., and Masip, D. (2015). Speeding up neural networks for large scale classification using wta hashing. *arXiv preprint arXiv:1504.07488*. 51

Barkan, O. and Koenigstein, N. (2016). Item2vec: neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE. 49, 52, 54

Bengio, Y. e. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722. 48, 50, 51

Blanc, G. and Rendle, S. (2017). Adaptive sampled softmax with kernel based sampling. *arXiv preprint arXiv:1712.00527*. 48, 50, 51, 64

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146. 61, 65

Bose, A. J., Ling, H., and Cao, Y. (2018). Adversarial contrastive estimation. *arXiv preprint arXiv:1805.03642*. 51

Bremler-Barr, A. and Levy, H. (2005). Spoofing Prevention Method. In *IEEE Infocom*. 18, 20

Bruzzone, L. and Prieto, D. F. (1999). An incremental-learning neural network for the classification of remote-sensing images. *Pattern Recognition Letters*. 17

Caruana, R. (1997). Multitask learning. *Machine learning*, 28(1):41–75. 38

Chan, H. Y. and Woodland, P. (2004). Improving broadcast news transcription by lightly supervised discriminative training. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages I–737. IEEE. 37

Charniak, E. and Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 173–180. Association for Computational Linguistics. 37, 40

Chen, W., Grangier, D., and Auli, M. (2015). Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906*. 48

Chen, Y., Das, S., Dhar, P., Saddik, A. E., and Nayak, A. (2008). Detecting and preventing ip-spoofed distributed dos attacks. *International Journal of Network Security*, 7(1):70–81. 18

cloudflare. The root cause of large DDoS IP Spoofing. https://blog.cloudflare.com/the-root-cause-of-large-ddos-ip-spoofing/. 18, 21

Collins, M., Roark, B., and Saraclar, M. (2005). Discriminative syntactic language modeling for speech recognition. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 507–514. Association for Computational Linguistics. 37

Consens, M. P. and Mendelzon, A. O. (1990). Low complexity aggregation in graphlog and datalog. In *International Conference on Database Theory*, pages 379–394. Springer. 69

Costa, M., Castro, M., Rowstron, A., and Key, P. (2004). PIC: Practical Internet coordinates for distance estimation. In *ICDCS*. 6, 21, 26

Cunha, I., Teixeira, R., Veitch, D., and Diot, C. (2011). Predicting and tracking internet path changes. In *ACM Sigcomm*. 34

Dabek, F., Cox, R., Kaashoek, F., and Morris, R. (2004). Vivaldi: a decentralized network coordinate system. In *SIGCOMM*. 4, 5, 6, 7, 14, 16, 17, 18, 21, 26

de Brébisson, A. and Vincent, P. (2015). An exploration of softmax alternatives belonging to the spherical loss family. *arXiv preprint arXiv:1511.05042*. 51

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. 72

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. 2

Dikici, E., Semerci, M., Saraçlar, M., and Alpaydin, E. (2012). Classification and ranking approaches to discriminative language modeling for asr. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(2):291–300. 37

Duan, Z., Yuan, X., and Chandrashekar, J. (2008). Controlling IP Spoofing through Interdomain Packet Filters. *IEEE Transactions on Dependable and Secure Computing*, 5(1). 18, 20

Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. (2015). The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16. 95

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*. 5

Eriksson, B., Barford, P., and Nowak, R. (2008). Network Discovery from Passive Measurements. In *ACM Sigcomm*. 4, 7

Eriksson, B., Barford, P., and Nowak, R. (2009). Estimating Hop Distance Between Arbitrary Host Pairs. In *IEEE Infocom*. 5, 7, 14, 16, 17, 19, 21, 26, 35

Eriksson, B., Barford, P., Nowak, R., and Crovella, M. (2007). Learning Network Structure from Passive Measurements. In *IMC*. 4

fastText (2019). Facebook fasttext project source code. 65

Feamster, N. and Rexford, J. (2017). Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583*. 4

Feng, W., Kaiser, E., Feng, W., and Luu, A. (2005). The Design and Implementation of Network Puzzles. In *Infocom*. 18, 20

Ferguson, P. and Senie, D. (2000). Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. BCP 38, RFC Editor. http://www.rfc-editor.org/rfc/rfc2827.txt. 18

Frankel, S. and Krishnan, S. (2011). IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, RFC Editor. http://www.rfc-editor.org/rfc/rfc6071.txt. 18, 20

github. February 28th DDoS incident. https://githubengineering.com/ddos-incident-report/. 18

Goldberg, Y. and Levy, O. (2014). word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*. 55

Gong, P., Ye, J., and Zhang, C. (2013). Multi-stage multi-task feature learning. *The Journal of Machine Learning Research*, 14(1):2979–3010. 38

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613. 72

Gonzalez, R., Manco, F., Garcia-Duran, A., Mendes, J., Huici, F., Niccolini, S., and Niepert, M. (2017). Net2vec: Deep learning for the network. In *ACM Big-DAMA'17*. 33

Goodman, J. Classes for fast maximum entropy training. *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*. 51

Grave, E., Joulin, A., Cissé, M., Jégou, H., et al. (2017). Efficient softmax approximation for gpus. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1302–1310. JMLR. org. 48

Grover, A. and Leskovec, J. (2016a). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 49, 54, 61, 66

Grover, A. and Leskovec, J. (2016b). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM. 52, 54

Gu, J., Watanabe, Y. H., Mazza, W. A., Shkapsky, A., Yang, M., Ding, L., and Zaniolo, C. (2019). Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on

spark. In *Proceedings of the 2019 International Conference on Management of Data*, pages 467–484. ACM. 69, 72

Gutmann, M. and Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304. 51, 53

Hakkani-Tür, D., Béchet, F., Riccardi, G., and Tur, G. (2006). Beyond asr 1-best: Using word confusion networks in spoken language understanding. *Computer Speech & Language*, 20(4):495–514. 47

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. 2

Jin, C., Wang, H., and Shin, K. G. Hop-count filtering: An effective defense against spoofed ddos traffic. 18, 19, 21, 22, 27, 28, 29

Jin, C., Wang, H., and Shin, K. G. (2003). Hop-count filtering: An effective defense against spoofed ddos traffic. In *CCS*. 4, 8, 16

John, W., Dusi, M., and claffy, k. (2010). Estimating Routing Symmetry on Single Links by Passive Flow Measurements. In *ACM International Workshop on TRaffic Analysis and Classification (TRAC)*. 35

Jyothi, P., Johnson, L., Chelba, C., and Strope, B. (2012). Large-scale discriminative language model reranking for voice-search. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 41–49. Association for Computational Linguistics. 37

Karpathy, A. and Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In *CVPR*. 5

Killalea, T. (2000). Recommended internet service provider security services and procedures. BCP 46, RFC Editor. 18

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 7, 12

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*. 7

Levchenko, K., Dhamdhere, A., Huffaker, B., claffy, k., Allman, M., and Paxson, V. (2017). PacketLab: A Universal Measurement Endpoint Interface. In *Internet Measurement Conference (IMC)*. 4

Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185. 56

Li, M., Lumezanu, C., Zong, B., and Chen, H. (2018a). Deep Learning IP Network Representations. In *ACM Sigcomm Big-DAMA*. 19, 22, 23, 24, 25, 28, 29, 35

Li, M., Lumezanu, C., Zong, B., and Chen, H. (2018b). Learning IP Network Representations. *SIGCOMM CCR*. 19, 23

Liu, X., Wang, Y., Chen, X., Gales, M. J., and Woodland, P. C. (2014). Efficient lattice rescoring using recurrent neural network language models. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4908–4912. IEEE. 47

Lone, Q., Luckie, M., Korczynski, M., and van Eeten, M. (2017). Using loops observed in traceroute to infer the ability to spoof. In *PAM*. 20, 27

Lumezanu, C., Levin, D., and Spring, N. (2007). PeerWise discovery and negotiation of faster paths. In *HotNets*. 18, 26, 34

Luong, T., Socher, R., and Manning, C. (2013). Better word representations with recursive neural networks for morphology. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 104–113. 65

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330. 64

McKinney, W. et al. Data structures for statistical computing in python. 70

McKinney, W. et al. (2010). Data structures for statistical computing in python. 73

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241. 72

Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. (2017). SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM Sigcomm*. 4

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. 48, 51, 52, 54, 61, 62

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*. 10

Mikolov, T., Le, Q. V., and Sutskever, I. (2013b). Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*. 5

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013c). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119. 48, 49, 52, 56, 65, 66

MLM (2008). Multi-level˙marketing. 79

Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*. 54

Morbini, F., Audhkhasi, K., Artstein, R., Van Segbroeck, M., Sagae, K., Georgiou, P., Traum, D. R., and Narayanan, S. (2012). A reranking approach for recognition and classification of speech input in conversational dialogue systems. In *2012 IEEE Spoken Language Technology Workshop (SLT)*, pages 49–54. IEEE. 37, 40

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer. 48, 50, 51

Mussmann, S., Levy, D., and Ermon, S. (2017). Fast amortized inference and learning in log-linear models with randomly perturbed nearest neighbor search. *arXiv preprint arXiv:1707.03372*. 51

Ng, T. S. E. and Zhang, H. (2002). Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*. 6, 14, 16, 21, 26

node2vec, G. (2019). Node2vec project souce code. https://github.com/snap-stanford/snap.git. 66

Oba, T., Hori, T., and Nakamura, A. (2007). An approach to efficient generation of high-accuracy and compact error-corrective models for speech recognition. In *Eighth Annual Conference of the International Speech Communication Association*. 37

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM. 68

Padmanabhan, R., Dhamdhere, A., Aben, E., kc claffy, and Spring, N. (2016). Reasons dynamic addresses change. In *IMC*. 34

Park, K. and Lee, H. (2001). On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *ACM Sigcomm*. 18

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035. 2

Peng, F., Roy, S., Shahshahani, B., and Beaufays, F. (2013). Search results based n-best hypothesis rescoring with maximum entropy classification. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 422–427. IEEE. 37, 40

Pyxida. Pyxida. http://pyxida.sourceforge.net/. 7, 21, 26

Rawat, A. S., Chen, J., Yu, F. X. X., Suresh, A. T., and Kumar, S. (2019). Sampled softmax with random fourier features. In *Advances in Neural Information Processing Systems*, pages 13834–13844. 48, 50, 51

RouteViews. http://www.routeviews.org. 13

Ruiz, F. J. R., Titsias, M. K., Dieng, A. B., and Blei, D. M. (2018). Augment and reduce: Stochastic inference for large categorical distributions. 50

Sak, H., Saraclar, M., and Güngör, T. (2010). On-the-fly lattice rescoring for real-time automatic speech recognition. In *Eleventh annual conference of the international speech communication association*. 37

Sak, H., Saraclar, M., and Gungor, T. (2011a). Discriminative reranking of ASR hypotheses with morpholexical and n-best-list features. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding, ASRU 2011, Waikoloa, HI, USA, December 11-15, 2011*, pages 202–207. 37

Sak, H., Saraçlar, M., and Güngör, T. (2011b). Discriminative reranking of asr hypotheses with morpholexical and n-best-list features. *2011 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 202–207. 37

Schroff, F., Kalenichenko, D., and Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823. 51

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681. 39

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*. 38

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., and Zaniolo, C. (2016). Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149. ACM. 69, 72

Shue, C., Shin, Y., Gupta, M., and Choi, J. Y. (2005). Analysis of ipsec overheads for vpn servers. In *1st IEEE ICNP Workshop on Secure Network Protocols, 2005. (NPSec).* 18

Sommer, R. and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. 20

Spearman, C. (1904). The proof and measurement of association between two things. *American journal of Psychology*, 15(1):72–101. 65

spoofing-ps. Port scanning techniques and the defense against them. https://www.sans.org/reading-room/whitepapers/auditing/port-scanning-techniques-defense-70. 18

spoofing-state. State of IP Spoofing. https://spoofer.caida.org/summary.php. 20, 27

Spring, N., Mahajan, R., and Anderson, T. (2003). Quantifying the Causes of Path Inflation. In *ACM Sigcomm*. 4

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629. 68

Tur, G. and De Mori, R. (2011). *Spoken language understanding: Systems for extracting semantic information from speech*. John Wiley & Sons. 36

Tur, G., Wright, J., Gorin, A., Riccardi, G., and Hakkani-Tür, D. (2002). Improving spoken language understanding using word confusion networks. In *Seventh International Conference on Spoken Language Processing*. 47

Vijayanarasimhan, S., Shlens, J., Monga, R., and Yagnik, J. (2014). Deep networks with large output spaces. *arXiv preprint arXiv:1412.7479*. 51

Vincent, P., De Brébisson, A., and Bouthillier, X. (2015). Efficient exact gradient update for training deep networks with very large sparse targets. In *Advances in Neural Information Processing Systems*, pages 1108–1116. 50, 51

Wang, D., Cui, P., and Zhu, W. (2016a). Structural Deep Network Embedding. In *KDD*. 17

Wang, D., Cui, P., and Zhu, W. (2016b). Structural deep network embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 22, 34

Wang, H., Jin, C., and Shin, K. G. (2007). Defense against spoofed ip traffic using hop-count filtering. *IEEE/ACM Transactions on Networking*, 15:40–53. 18, 21, 26

Wang, L., Li, Y., Huang, J., and Lazebnik, S. (2018). Learning two-branch neural networks for image-text matching tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 5

Wang, Q., Mao, Z., Wang, B., and Guo, L. (2017). Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743. 49

word2vec, G. (2019). Word2vec project source code. https://github.com/tmikolov/word2vec.git. 65

Yaar, A., Perrig, A., and Song, D. (2006). Stackpi: New packet marking and filtering mechanisms for ddos and ip spoofing defense. *IEEE Journal on Selected Areas in Communications*, 24(10):1853–1863. 18

Yang, M., Shkapsky, A., and Zaniolo, C. Parallel bottom-up evaluation of logic programs: Deals on shared-memory multicore machines. 69

Zafarani, R. and Liu, H. (2009). Social computing data repository at ASU. 66

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association. 68

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. Spark: Cluster computing with working sets. 2

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. 72

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. 64

Zhao, X., Sala, A., Zheng, H., and Zhao, B. Y. (2011). Efficient shortest paths on massive social graphs. In *CollaborateCom*. 7