Cybersecurity Anomaly and Outlier Detection Validation

17 July 2022

Research Facilitation Laboratory
Army Analytics Group

THIS PAGE INTENTIONALLY LEFT BLANK

Army Analytics Group (AAG) Cooperative Research and Development Agreement (CRADA) with Entanglement Inc Summary Report

Written by:

Clay Stanek, Ph.D.
Chief of Technology, RFL

Reviewed by:

Douglas Bonett, Ph.D.
Chief of Science, RFL

Approved by:

LENZINI.JOSHUA.MARTIN.1186297047 Digitally signed by LENZINI.JOSHUA.MARTIN.1186297047 Date: 2022.09.19 10:53:08 -07'00'
Mr. Joshua M. Lenzini
Director, AAG-RFL
mailto: *Joshua.m.lenzini.civ@army.mil*
707-400-3601

JENSEN.DANIEL.CHRISTIAN.1211690395 Digitally signed by JENSEN.DANIEL.CHRISTIAN.1211690395 Date: 2022.09.20 07:14:35 -04'00'
Mr. Dan Jensen
Executive Director, AAG
mailto: daniel.c.jensen.civ@army.mil
707-980-4416

Research Facilitation Laboratory

UNCLASSIFIED

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** 17 Jul 2022	**3. REPORT TYPE AND DATES COVERED** External Technical Report	
4. TITLE AND SUBTITLE Cyber Anomaly and Cyber Telemetry Prototypes		**5. FUNDING NUMBERS** #	
6. AUTHOR(S) Clay Stanek			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AAG- Research Facilitation Laboratory 20 Ryan Ranch Road, Suite 215 Monterey, CA		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Army Analytics Group & AAG's Research Facilitation Laboratory		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: Distribution unlimited		**12b. DISTRIBUTION CODE** A	
13. ABSTRACT (maximum 200 words) AAG's Research Facilitation Laboratory (RFL) Investigated the next-generation HW capability of Groq combined with novel deep-learning cyber anomaly and telemetry algorithms			
14. SUBJECT TERMS Autoencoders, anomaly, telemetry, Generative Adversarial Networks (GANs), Quadratic Unconstrained Binary Optimization (QUBO), supervised, unsupervised, machine learning, inference			**15. NUMBER OF PAGES 73** **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UU

Contents

Table of Figures

LIST OF ABBREVIATIONS AND ACRONYMS

Abbreviation (Acronym)	Definition
AAG	Army Analytics Group
ACR	Army Central Registry
AE	Autoencoder
AIC	Akaike information criterion
ANOVA	Analysis of variance
b	Unstandardized b (regression coefficient)
CA	Civil Affairs
CI	Confidence Interval
CRADA	Cooperative Research and Development Agreement
DA	Digital Annealer
DoD	Department of Defense
EA	Evolutionary Algorithm
ES	Executive Summary
F	The value of the F statistical test
GAN	Generative Adversarial Network
GPF	General Purpose Force
M1	Survival Analysis Model 1
M2	Survival Analysis Model 2
M	Mean
PPEOV	Personal Protective Equipment Optimization Validation
QUBO	Quadratic Unconstrained Binary Optimization
Std	Standard Deviation
SVM	Support Vector Machine

Executive Summary

Background

In the course of performing a CRADA with the Army Analytics Group and its Research Facilitation Laboratory, Entanglement, Inc. (EI), has demonstrated a dramatically faster and more accurate cybersecurity anomaly detection capability - with far fewer false positives - than any known technology.

Most cybersecurity reporting across the world (including the 2022 Sonicwall Report) concluded that almost every type of cyber-attack rose significantly in 2021, including zero-day and ransomware attacks. All these attacks have a common thread: cyber anomalies. Anomaly detection in cybersecurity is the identification of rare occurrences, items, or events of concern due to characteristics differing from most of the processed data, which allows organizations to track security errors, structural defects and even fraud. The three main forms of anomaly detection are: unsupervised, supervised, and semi-supervised. Security Operations Center (SOC) analysts use each of these approaches to varying degrees of effectiveness in Cybersecurity applications. Systems limited to supervised machine learning tend to flag so many potential anomalies that analysts are left battling an endlessly growing stack of false positive alerts, suffering from cognitive overload.

Excessive logins, spikes in traffic between two points, and an unusually large number of remote logins are a few examples of anomalies. As we learned during the pandemic response in 2020, this latter “anomaly” was necessary for many organizations to keep business moving when workers were stuck at home. Given the challenges presented from the scale of remote working during the COVID-19 Pandemic and the increased cyber threats of 2021, the U.S. Army turned to the private sector to explore a range of possible solutions.

In May of 2021, President Biden issued an Executive Order mandating all federal agencies to adopt zero-trust security. In the third quarter of 2021, a new approach for

cybersecurity was proposed to address the continuous monitoring portion of the recently mandated zero-trust security architecture. If successful, the capability could be applied to and help give real-time situational awareness to larger networks operated by the Army and other federal agencies. It was, in part, based upon research in deep neural networks with the goals of (a) accelerating auto-encoder (AE) functionality; (b) accelerating generative adversarial network (GAN) functionality; and (c) integrating a quantum-inspired optimization algorithm called a support vector machine (SVM). The approach, which included a new application of Quadratic Unconstrained Binary Optimization (QUBO) for cyber security anomaly and outlier detection, was commissioned by the USG. Under the direction of the Office of Business Transformation, the Army Analytics Group (AAG) immediately began working with a wide range of potential sources of emerging technologies that might be employed to defeat the threat of cyber anomalies. In June 2021, the director of the AAG, Mr. Dan Jensen, became aware of a no-cost offer of assistance by Entanglement, Inc., who selected its strategic partner and team participant, Groq, Inc., a U.S. semiconductor company, to provide the Army with novel, groundbreaking proprietary technology, and computational service.

The Entanglement team offered its services to assist the Army in determining an optimal cybersecurity anomaly detection capability within twelve months. In June 2021, AAG and Entanglement extended an existing Cooperative Research and Development Agreement (CRADA) entitled “COVID-19 Resource Allocation Optimization.” The Entanglement team worked for the next several weeks with the AAG’s Research Facilitation Laboratory led by Dr. Clay Stanek and demonstrated significant performance improvements and feasibility in October 2021.

Key Findings, Implications, and Recommendations

The work under the CRADA culminated in the validated capability to solve cybersecurity anomaly detection faster than traditional methods, and with better performance as measured by Key Performance Parameters (KPP’s). The KPP’s covered metrics related

to total inferences per second, percentage of threats detected, accuracy, recall, precision, other confusion matrix-based metrics, and Area Under the Curve (AUC).

With additional variables or larger datasets, the Entanglement/Groq capability offers greater efficiency than traditional methods and can solve otherwise intractable problems at scale. The core technology is a proprietary purpose-built digital circuit design with high degrees of parallelism for solving classes of problems that can be expressed as deep neural network models and Quadratic Unconstrained Binary Optimization (QUBO) problems. Previous AAG efforts showed the ability to detect 120,000 inferences per second. This was the metric used as the benchmark and standard achievable using a QUBO model. Benchmarking was based on a solution set which joins an algorithmic solution with a proprietary quantum inspired chip. The chip solution can scale out to cards, nodes, and beyond. Additionally, the existing solution benchmarked for CRADA feasibility is already in development for next generation updates which will improve modularity and reduce heat signatures.

Within six months Entanglement was able to achieve an anomaly detection rate of 72,000,000 inferences per second and demonstrated the potential to achieve 120,000,000 inferences per second across a wide domain of data processing systems.

The validation cases were constructed from the KDD Cup 1999 (KDD99) dataset and the CICIDS2017 data set. The calculated output demonstrated for the AE and GAN solution was extremely effective in determining anomalies as outlined in the model performance section. The QUBO SVM was built in quantum-ready form and was also effective at anomaly detections and finally was able to do the entire data set calculation in approximately 250 milliseconds.

Introduction

Background

The U.S. Army is teamed with innovative industry partners and has developed a quantum-inspired and quantum-ready and accelerated AI computing environment (Accelerated AI Platform) that offers speed, scale and accuracy for optimization and AI problems. The platform's analytics capabilities will assist the Government in quickly optimizing and accelerating AI and machine learning processes using Entanglement's solver NGQ™ (quantum optimization), and Neural Network applications for Cyber Threat Detection, specifically real-time anomaly detection.

AAG seeks to apply NGQ™ and novel AI hardware processors to three areas that would support the continuous monitoring portion of Zero Trust architectures. This includes an anomaly detection algorithm capable of continuously vetting all users on a network and their actions. A similar algorithmic framework will be suitable for demonstrating Intrusion Detection Systems (IDS) and expanded threat awareness at network endpoints to improve the processing of telemetry data dramatically within current cyber operations. In demonstrating such a capability, this work will have engineered a new class of anomaly detection algorithms capable of use not only for cyber, but for many problems where the events of interest happen very infrequently but are of great significance when they do.

Some of the distinct advantages which the Entanglement solution provides are:

- Uninterrupted Security: Stack-on Threat Models which prevent having to interrupt security operations to retrain for a new threat detected.
- No increase in latency when model increases performance.
- Unified versus distributed use of memory.
- Scalable architecture to include chip, card, node, and rack.
- Fast model context switching.
- Quantum-inspired and Quantum-ready.

- Utilizes next-gen QUBO (Quadratic Unconstrained Binary Optimization) solver.
- Hardware/Software deployable solution available today.

Overall, the technology created through the U.S. Army CRADA is an enterprise cybersecurity solution that gives total situational awareness over the enterprise to detect and resolve anomalies in support of a zero-trust environment.

Project Aims and Guiding Questions

The project has 3 guiding questions that provide structure and clarity for the analyses:

1. **Can an anomaly detection solution, with new hardware, be used to implement an autoencoder (AE) for cybersecurity with greater performance than existing systems?**
2. **Can an algorithmic framework suitable for demonstrating Intrusion Detection Systems (IDS) and expanded threat awareness, at network endpoints, be implemented into a Generative Adversarial Network for cyber anomaly detection with greater performance than existing systems?**
3. **Can unsupervised cyber-telemetry algorithms be formulated in QUBO form to perform sparse correlations of data?**

Project Roadmap

Modeling phase: Collect, cleanse and create a standardized anomaly and telemetry data set to use as input into SW algorithm models that will be implemented in hardware (HW) using Groq's Tensor Streaming Processor. Algorithms include deep-learning enabled and quantum ready for application to the anomaly detection problem.
Run phase: Run the HW models created in the previous step. Design and create a scalable solution that works for large datasets. Fine-tune any parameters (both business related, hyperparameter and QUBO related) to obtain the best possible solution and performance.
Report: Define and compute Key Performance Metrics and perform any further fine-tuning of the parameters.

Validation: Validate the full solutions by creating and comparing against an alternate approach based on classical algorithms on NVIDIA or standard CPU technology. Documentation of all the findings.

Input Data Sets

The KDD99 Data Set

We describe the input data sets with anomaly rates and types of anomalies. The KDD99 dataset consists of normal data points and points that have been labeled as Denial of Service (DoS), Remote to User (R2L), User to Root (U2R), and Probing (Probe) by logging network packet information. More information about the dataset can be found at https://kdd.ics.uci.edu/databases/kddcup99/task.html.

Examining Columns From Five Normal Examples in the KDD99 Data Set

First, let's look at the column names in the KDD data set.

col_names =

["duration","protocol_type","service","flag","src_bytes","dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins","logged_in","num_compromised","root_shell","su_attempted","num_root","num_file_creations","num_shells","num_access_files","num_outbound_cmds","is_host_login","is_guest_login","count","srv_count","serror_rate","srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate","diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count","dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate","dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate","dst_host_rerror_rate", "dst_host_srv_rerror_rate","label"]

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	root_shell	su_attempted	num_root	num_file_creations
0	0	tcp	http	SF	215	45076	0	0	0	0	0	1	0	0	0	0	0
1	0	tcp	http	SF	162	4528	0	0	0	0	0	1	0	0	0	0	0
2	0	tcp	http	SF	236	1228	0	0	0	0	0	1	0	0	0	0	0
3	0	tcp	http	SF	233	2032	0	0	0	0	0	1	0	0	0	0	0
4	0	tcp	http	SF	239	486	0	0	0	0	0	1	0	0	0	0	0

num_shells	num_access_files	num_outbound_cmds	is_host_login	is_guest_login	count	srv_count	serror_rate	srv_serror_rate	rerror_rate	srv_rerror_rate	same_srv_rate	diff_srv_rate	srv_diff_host_rate
0	0	0	0	0	1	1	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0	0	0	0	0	2	2	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0	0	0	0	0	1	1	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0	0	0	0	0	2	2	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0	0	0	0	0	3	3	0.0	0.0	0.0	0.0	1.0	0.0	0.0

dst_host_count	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_same_src_port_rate	dst_host_srv_diff_host_rate	dst_host_serror_rate	dst_host_srv_serror_rate	dst_host_rerror_rate
0	0	0.0	0.0	0.00	0.0	0.0	0.0	0.0
1	1	1.0	0.0	1.00	0.0	0.0	0.0	0.0
2	2	1.0	0.0	0.50	0.0	0.0	0.0	0.0
3	3	1.0	0.0	0.33	0.0	0.0	0.0	0.0
4	4	1.0	0.0	0.25	0.0	0.0	0.0	0.0

dst_host_srv_rerror_rate	label
0.0	normal.
0.0	normal.
0.0	normal.
0.0	normal.
0.0	normal.

Figure 1 Five Examples from Beginning of KDD99 Data Set

The number of columns is too wide to fit in one image so there are a series of ‘strips’ that work from left to right, top to bottom that compose Figure 1. The last column in each sample is called the ‘label’, which tells us whether the sample is normal or what specific type of anomaly it is. We will discuss anomaly types below.

Examining Columns From Five Normal and Five Anomalous Examples in the KDD99 Data Set

Now, we print out five anomalous examples with five normal examples for feature comparison. The first five are the ‘portsweep’ anomaly and the last five are normal examples. This is shown in Figure 2

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	root_shell	su_attempted	num_root	num_file_creations
226552	0	tcp	ftp_data	REJ	0	0	0	0	0	0	0	0	0	0	0	0	0
226553	1	tcp	ftp_data	RSTR	0	0	0	0	0	0	0	0	0	0	0	0	0
226554	0	tcp	private	REJ	0	0	0	0	0	0	0	0	0	0	0	0	0
226555	1	tcp	private	RSTR	0	0	0	0	0	0	0	0	0	0	0	0	0
226556	0	tcp	private	REJ	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	tcp	http	SF	215	45076	0	0	0	0	0	1	0	0	0	0	0
1	0	tcp	http	SF	162	4528	0	0	0	0	0	1	0	0	0	0	0
2	0	tcp	http	SF	236	1228	0	0	0	0	0	1	0	0	0	0	0
3	0	tcp	http	SF	233	2032	0	0	0	0	0	1	0	0	0	0	0
4	0	tcp	http	SF	239	486	0	0	0	0	0	1	0	0	0	0	0

num_shells	num_access_files	num_outbound_cmds	is_host_login	is_guest_login	count	srv_count	serror_rate	srv_serror_rate	rerror_rate	srv_rerror_rate	same_srv_rate	diff_srv_rate	srv_diff_host_rate
0	0	0	0	0	1	1	0.0	0.0	1.0	1.0	1.00	0.00	0.0
0	0	0	0	0	2	2	0.0	0.0	1.0	1.0	1.00	0.00	0.0
0	0	0	0	0	3	1	0.0	0.0	1.0	1.0	0.33	0.67	0.0
0	0	0	0	0	4	2	0.0	0.0	1.0	1.0	0.50	0.50	0.0
0	0	0	0	0	5	1	0.0	0.0	1.0	1.0	0.20	0.60	0.0
0	0	0	0	0	1	1	0.0	0.0	0.0	0.0	1.00	0.00	0.0
0	0	0	0	0	2	2	0.0	0.0	0.0	0.0	1.00	0.00	0.0
0	0	0	0	0	1	1	0.0	0.0	0.0	0.0	1.00	0.00	0.0
0	0	0	0	0	2	2	0.0	0.0	0.0	0.0	1.00	0.00	0.0
0	0	0	0	0	3	3	0.0	0.0	0.0	0.0	1.00	0.00	0.0

dst_host_count	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_same_src_port_rate	dst_host_srv_diff_host_rate	dst_host_serror_rate	dst_host_srv_serror_rate	dst_host_rerror_rate
171	62	0.27	0.02	0.01	0.03	0.01	0.0	0.29
172	62	0.27	0.02	0.01	0.03	0.01	0.0	0.30
173	1	0.01	0.03	0.02	0.00	0.01	0.0	0.30
174	2	0.01	0.03	0.02	0.00	0.01	0.0	0.30
175	1	0.01	0.03	0.03	0.00	0.01	0.0	0.31
0	0	0.00	0.00	0.00	0.00	0.00	0.0	0.00
1	1	1.00	0.00	1.00	0.00	0.00	0.0	0.00
2	2	1.00	0.00	0.50	0.00	0.00	0.0	0.00
3	3	1.00	0.00	0.33	0.00	0.00	0.0	0.00
4	4	1.00	0.00	0.25	0.00	0.00	0.0	0.00

srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_same_src_port_rate	dst_host_srv_diff_host_rate	dst_host_serror_rate	dst_host_srv_serror_rate	dst_host_rerror_rate	dst_host_srv_rerror_rate	label
62	0.27	0.02	0.01	0.03	0.01	0.0	0.29	0.02	portsweep.
62	0.27	0.02	0.01	0.03	0.01	0.0	0.30	0.03	portsweep.
1	0.01	0.03	0.02	0.00	0.01	0.0	0.30	1.00	portsweep.
2	0.01	0.03	0.02	0.00	0.01	0.0	0.30	1.00	portsweep.
1	0.01	0.03	0.03	0.00	0.01	0.0	0.31	1.00	portsweep.
0	0.00	0.00	0.00	0.00	0.00	0.0	0.00	0.00	normal.
1	1.00	0.00	1.00	0.00	0.00	0.0	0.00	0.00	normal.
2	1.00	0.00	0.50	0.00	0.00	0.0	0.00	0.00	normal.
3	1.00	0.00	0.33	0.00	0.00	0.0	0.00	0.00	normal.
4	1.00	0.00	0.25	0.00	0.00	0.0	0.00	0.00	normal.

Figure 2 Comparison of features between anomaly and normal example

Here are the 23 types of labels in the KDD99 dataset:

['back.' 'buffer_overflow.' 'ftp_write.' 'guess_passwd.' 'imap.' 'ipsweep.' 'land.' 'loadmodule.' 'multihop.' 'neptune.' 'nmap.' 'normal.' 'perl.' 'phf.' 'pod.' 'portsweep.' 'rootkit.' 'satan.' 'smurf.' 'spy.' 'teardrop.' 'warezclient.' 'warezmaster.']

Note that the label ‘normal’ is the 11th type of label. All other labels and their corresponding number in the list are examples of anomalies. Said another way, label type ‘*normal’* is enumerated with 11. All other label numbers refer to types of anomalies.

One-Hot Encoding of Categorical Features

For the columns that are categorical, we must one-hot encode them:

cat_vars = ['protocol_type', 'service', 'flag', 'land', 'logged_in','is_host_login', 'is_guest_login']

Here in Figure 3, One-hot Encoded Features, we show the first five examples as their categories look under one-hot encoding.

	land	logged_in	is_host_login	is_guest_login	protocol_type_icmp	protocol_type_tcp	protocol_type_udp	service_IRC	service_X11	service_Z39_50	service_aol	service_auth	service_bgp	service_courier
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0	0	0	0
3	0	1	0	0	0	1	0	0	0	0	0	0	0	0
4	0	1	0	0	0	1	0	0	0	0	0	0	0	0

service_csnet_ns	service_ctf	service_daytime	service_discard	service_domain	service_domain_u	service_echo	service_eco_i	service_ecr_i	service_efs	service_exec	service_finger	service_ftp	service_ftp_data
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

service_gopher	service_harvest	service_hostnames	service_http	service_http_2784	service_http_443	service_http_8001	service_imap4	service_iso_tsap	service_klogin	service_kshell	service_ldap	service_link
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0

service_login	service_mtp	service_name	service_netbios_dgm	service_netbios_ns	service_netbios_ssn	service_netstat	service_nnsp	service_nntp	service_ntp_u	service_other	service_pm_dump	service_pop_2
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

service_pop_3	service_printer	service_private	service_red_i	service_remote_job	service_rje	service_shell	service_smtp	service_sql_net	service_ssh	service_sunrpc	service_supdup	service_systat	service_telnet
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

service_time	service_urh_i	service_urp_i	service_uucp	service_uucp_path	service_vmnet	service_whois	flag_OTH	flag_REJ	flag_RSTO	flag_RSTOS0	flag_RSTR	flag_S0	flag_S1	flag_S2	flag_S3	flag_SF	flag_SH
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Figure 3 One-Hot Encoded Columns of KDD99 Data Set

```
# concatenate numeric and the encoded categorical variables
numeric_cat_data = pd.concat([numeric_data, cat_data], axis=1)


# here we do a quick sanity check that the data has been concatenated correctly by checking the dimension
of the matrices
print(cat_data.shape):  (4898431, 88)
print(numeric_data.shape):  (4898431, 34)
print(numeric_cat_data.shape): (4898431, 122)
```

For example, the categorical variable "protocol_type" is split into three categories, protocol_type_icmp, protocol_type_tcp and protocol_type_udp. Now that the one hot encoding of the categorical data is done, we need to merge the numerical data from the original data.

Notice that the one-hot encoding increased the number of feature columns from an original 41 to 122. Most of them come from the one-hot encoding of the different types of category for '*service'* and '*flag'*.

Splitting of Test and Train Data Sets

Now let's split the data into training set and test set in the ratio of 75:25. We will be using LabelEncoder, fit_transform and train_test_split from scikit-learn Python Machine Learning package. Here are the dimensions on the training data set, training data set label, test data set, and test data set label. We can see they are in a ratio of 3:1. Recall that *y_train* and *y_test* are the labels for each row sample. If it is normal, the number '11' appears as the label. Otherwise, the associated enumeration with each particular anomaly occurs in the *y_test* and *y_train* sets.

```
print(x_train.shape):  (3673823, 122)
print(y_train.shape): (3673823, 1)
print(x_test.shape):  (1224608, 122)
print(y_test.shape):  (1224608, 1)
```

Saving the Data for Later Training and Inference with AE and GAN

And the last step of our data preprocessing is to save the full KDD99 dataset that has been one-hot encoded as what is known in Python as a Pickle file. It is just a binary file type that is optimized for I/O in Python.

```
# save the datasets for later use
preprocessed_data ={
        'x_train':x_train,'y_train':y_train,'x_test':x_test, 'y_test':y_test, 'le':le
}

# pickle the preprocessed_data
path = 'preprocessed_data_full.pkl'
out = open(path, 'wb')
pickle.dump(preprocessed_data, out)
out.close()
```

Distribution of Normal and Anomalous Examples in Existing File

At this current state, let's take a look at how many normal and anomalous examples appear in our training and test set of 4898431 examples.

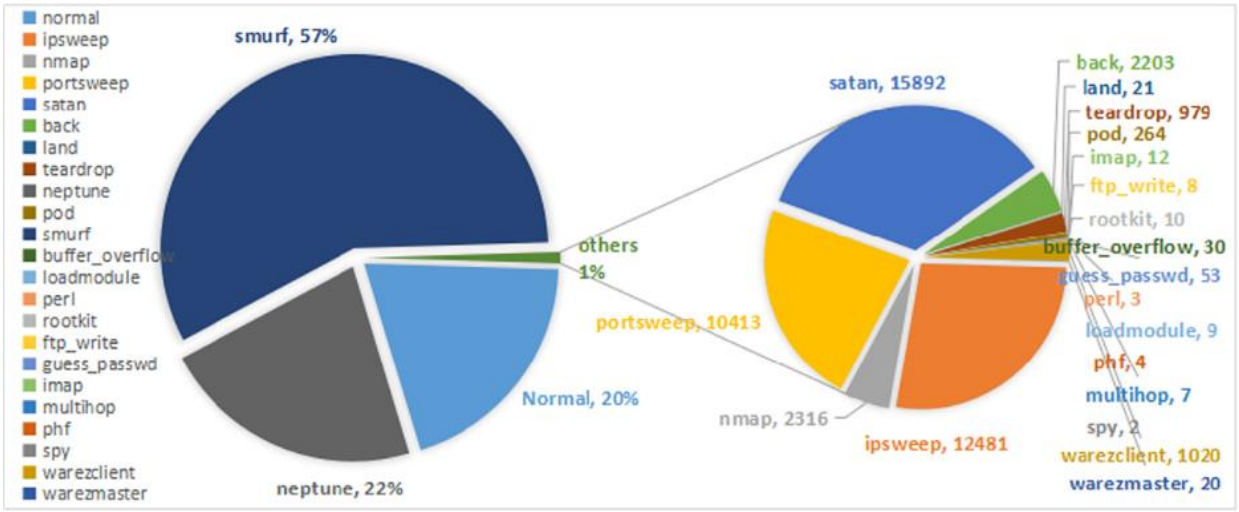


Figure 4 Pie Chart Showing Components of Normal and Anomalous Examples in KDD99 Data Set

Figure 4 is a very important figure. The KDD99 data set at this point only contains 20% normal examples! This will require addressing and is the most subtle part of working with

the training of the AE and GAN. Therefore, we will have to do one more step with the data before using it for training: we will need to resample it so the ratio of anomalous to normal examples is in the range of 1 to 100 (1%) and at most 1 to 20 (5%). The AE will be very sensitive to this ratio while the GAN will not be anywhere near as sensitive. In the tests run by RFL and subsequently by Groq, the base rate was chosen to be 1%. We will discuss this in the next section.

In the cell below, choose to either use 1% or 5% anomaly in data set by setting the *pct_anomalies* parameter to .01 or .05 respectively.

```
pct_anomalies = .01

!python preprocess_data.py --pct_anomalies $pct_anomalies

filename = './preprocessed_data_full.pkl'
input_file = open(filename,'rb')
preprocessed_data = pickle.load(input_file)
input_file.close()
```

Recall that we have constructed training and test sets where we removed most of anomalous data in the KDD99 dataset. This lets us simulate a more realistic anomaly detection problem where anomalies only comprise a small percentage of the data. We also trained a label encoder on the anomalous labels. This will allow us to go back and forth between labels and their encoded values.

Most of the data preprocessing has already been done. We one-hot encoded the categorical variables and separated the labels off from the input data. For training deep autoencoder (AE) models, the input data will also have to be scaled between 0 and 1.

```
# Normalize the testing and training data using the MinMaxScaler from the scikit learn package
scaler = MinMaxScaler()

# Make sure to only fit the scaler on the training data
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# convert the data to FP32
x_train = x_train.astype(np.float32)
x_test = x_test.astype(np.float32)
```

The preprocessing is complete and the KDD99 data set has been configured with 1% anomalies, appropriately scaled, and ready for testing with Autoencoders and Generative Adversarial Network anomaly detection.

Processing of the CICIDS2017 Data Set

Evaluations of eleven datasets since 1998 show that most are out of date and unreliable. Some of these datasets suffer from the lack of traffic diversity and volumes, some do not cover the variety of known attacks, while others anonymize packet payload data, which cannot reflect the current trends. Some are also lacking feature set and metadata.

CICIDS2017 dataset contains benign and the most up-to-date common attacks, which resembles the true real-world data (PCAPs). It also includes the results of the network traffic analysis using CICFlowMeter with labeled flows based on the time stamp, source, and destination IPs, source and destination ports, protocols and attack (CSV files). Also available is the extracted features definition.

Generating realistic background traffic was the top priority in building this dataset. We have used our proposed B-Profile system (Sharafaldin, et al. 2016) to profile the abstract

behavior of human interactions and generates naturalistic benign background traffic. For this dataset, we built the abstract behavior of 25 users based on the HTTP, HTTPS, FTP, SSH, and email protocols.

The data capturing period started at 9 a.m., Monday, July 3, 2017, and ended at 5 p.m. on Friday, July 7, 2017, for a total of 5 days. Monday is the normal day and only includes benign traffic. The implemented attacks include Brute Force FTP, Brute Force SSH, DoS, Heartbleed, Web Attack, Infiltration, Botnet and DDoS. They have been executed both morning and afternoon on Tuesday, Wednesday, Thursday and Friday.

In their recent dataset evaluation framework (Gharib et al., 2016), they have identified eleven criteria that are necessary for building a reliable benchmark dataset. None of the previous IDS datasets could cover all of the 10 criteria. In the following, we briefly outline these criteria:

1. Complete Network configuration: A complete network topology includes Modem, Firewall, Switches, Routers, and presence of a variety of operating systems such as Windows, Ubuntu, and Mac OS X.
2. Complete Traffic: By having a user profiling agent and 12 different machines in Victim-Network and real attacks from the Attack-Network.
3. Labelled Dataset: Section 4 and Table 2 show the benign and attack labels for each day. Also, the details of the attack timing will be published on the dataset document.
4. Complete Interaction: We covered both within and between internal LAN by having two different networks and Internet communication as well.
5. Complete Capture: Because we used the mirror port, such as a tapping system, all traffics have been captured and recorded on the storage server.
6. Available Protocols: Provided the presence of all commonly available protocols, such as HTTP, HTTPS, FTP, SSH and email protocols.

7. Attack Diversity: Included the most common attacks based on the 2016 McAfee report, such as Web-based, Brute force, DoS, DDoS, Infiltration, Heart-bleed, Bot, and Scan covered in this dataset.
8. Heterogeneity: Captured the network traffic from the main Switch and memory dump and system calls from all victim machines, during the execution of the attack.
9. Feature Set: Extracted more than 80 network flow features from the generated network traffic using CICFlowMeter and delivered the network flow dataset as a CSV file. See our PCAP analyzer and CSV generator.
10. MetaData: Completely explained the dataset which includes the time, attacks, flows and labels in the published paper.

Below is the python code required to load the CICIDS2017 data set and preprocess it for loading into the SVM QUBO solver.

```
processedData=pd.read_csv("F:/benchmarks/MachineLearningCVE/processedData.csv")
finalData= maximum_absolute_scaling(processedData)
finalData=finalData.dropna()
RowOfP=finalData.loc[finalData['LabelN']==1]
print('Number of positive:', len(RowOfP))
RowOfP1=RowOfP.reset_index(drop=False)
percentageOfPositive=0.001
NumberOfRemovedPositive=int(len(RowOfP1) -␣
,→percentageOfPositive*len(processedData))
realDropList=[]
randomlist = random.sample(range(1, len(RowOfP1)), NumberOfRemovedPositive)
for index in randomlist :
realIndexOfElement = RowOfP1['index'][index]
realDropList.append(realIndexOfElement)
finalData=finalData.drop(realDropList)
Y=finalData['LabelN']
finalProcessedData =finalData.drop(columns=['LabelN'])
print('Number of Normal ', collections.Counter(Y)[0])
print('Number of Anomalous ', collections.Counter(Y)[1])
x_train, x_test, y_train, y_test = train_test_split(finalProcessedData␣
,→,Y,test_size=0.3,random_state=41)
print("Length of Train Data:", len(x_train))
print("Length of Test Data:", len(x_test))
finalProcessedData
```

```
Number of positive: 291001
Number of Normal 1399824
Number of Anomalous 1693
Length of Train Data: 981061
Length of Test Data: 420456
```

```
print('Shape of Independent features data : ' + str(x.shape))
import matplotlib.pyplot as plt
import seaborn as sns
# corr = data.corr()
# ax, fig = plt.subplots(figsize=(15,15))
# sns.heatmap(corr, vmin=-1, cmap='coolwarm', annot=True)
# plt.show()
```

```
print('Shape of Independent features data : ' + str(x.shape))
import matplotlib.pyplot as plt
import seaborn as sns
# corr = data.corr()
# ax, fig = plt.subplots(figsize=(15,15))
# sns.heatmap(corr, vmin=-1, cmap='coolwarm', annot=True)
# plt.show()
```

```
Shape of Independent features data : (2000, 77)
Shape of Independent features Train data : (1600, 77)
Shape of Dependent features Train data : (1600, 1)
Shape of Independent features Test data: (400, 77)
Shape of Dependent features Test data: (400, 1)
```

We see there is an 80/20 split of the training to the test data and there are 77 features in the data itself. Below we provide examples from 10 rows and a sample of 20 of the 77 columns.

	Flow Duration	Total Fwd Packets	Total Backward Packets
0	2.500000e-08	0.000009	0.000000
1	9.083334e-07	0.000005	0.000003
2	4.333334e-07	0.000005	0.000003
3	2.833333e-07	0.000005	0.000003
4	2.500000e-08	0.000009	0.000000
1692126	4.083334e-07	0.000005	0.000010
1692127	1.808333e-06	0.000009	0.000003
1692128	1.156289e-02	0.000187	0.000158
1692129	1.725000e-06	0.000005	0.000003
1692130	4.166667e-07	0.000005	0.000007

	Total Length of Fwd Packets	Total Length of Bwd Packets
0	9.302326e-07	0.000000e+00
1	4.651163e-07	9.153974e-09
2	4.651163e-07	9.153974e-09
3	4.651163e-07	9.153974e-09
4	9.302326e-07	0.000000e+00
1692126	4.651163e-07	2.746192e-08
1692127	2.403101e-06	9.153974e-09
1692128	2.114729e-04	1.012124e-05
1692129	0.000000e+00	0.000000e+00
1692130	0.000000e+00	0.000000e+00

	Fwd Packet Length Max	Fwd Packet Length Min \
0	0.000242	0.002581
1	0.000242	0.002581

2	0.000242	0.002581
3	0.000242	0.002581
4	0.000242	0.002581
1692126	0.000242	0.002581
1692127	0.001249	0.000000
1692128	0.018372	0.000000
1692129	0.000000	0.000000
1692130	0.000000	0.000000

	Fwd Packet Length Mean	**Fwd Packet Length Std **
0	0.001010	0.000000
1	0.001010	0.000000
2	0.001010	0.000000
3	0.001010	0.000000
4	0.001010	0.000000
1692126	0.001010	0.000000
1692127	0.002609	0.003076
1692128	0.011200	0.015456
1692129	0.000000	0.000000
1692130	0.000000	0.000000

	Bwd Packet Length Max	**act_data_pkt_fwd**	**min_seg_size_forward**
0	0.000000 ...	0.000005	2.384208e-07
1	0.000457 ...	0.000000	2.384208e-07
2	0.000457 ...	0.000000	2.384208e-07
3	0.000457 ...	0.000000	2.384208e-07
4	0.000000 ...	0.000005	2.384208e-07
1692126	0.000457 ...	0.000000	2.384208e-07
1692127	0.000457 ...	0.000000	3.814732e-07
1692128	0.074277 ...	0.000112	3.814732e-07
1692129	0.000000 ...	0.000000	3.814732e-07
1692130	0.000000 ...	0.000000	3.814732e-07

	Active Mean	**Active Std**	**Active Max**	**Active Min**	**Idle Mean**
0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0
1692126	0.0	0.0	0.0	0.0	0.0
1692127	0.0	0.0	0.0	0.0	0.0
1692128	0.0	0.0	0.0	0.0	0.0
1692129	0.0	0.0	0.0	0.0	0.0
1692130	0.0	0.0	0.0	0.0	0.0

	Idle Std	**Idle Max**	**Idle Min**
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	0.0	0.0
4	0.0	0.0	0.0
1692126	0.0	0.0	0.0
1692127	0.0	0.0	0.0
1692128	0.0	0.0	0.0
1692129	0.0	0.0	0.0
1692130	0.0	0.0	0.0

[1401517 rows x 77 columns]

Metrics and Key Performance Parameters (KPPs)

For non-QUBO performance, Confusion matrix, accuracy, precision, sensitivity, Matthew's Correlation coefficient, ROC and AUC curves are used. We define these metrics in more detail below.

When measuring performance in AI/ML, the most common metric is the receiver operator characteristic (ROC) curve. It is simply a plot of the number of false negatives against the number of true positives. A perfect classifier is a vertical line that goes from (0,0) to (0,1) and then stays at y=1 for all values of x. A classifier that is done by flipping a coin is a line with 45 degree slope running from (0,0) to (1,1).

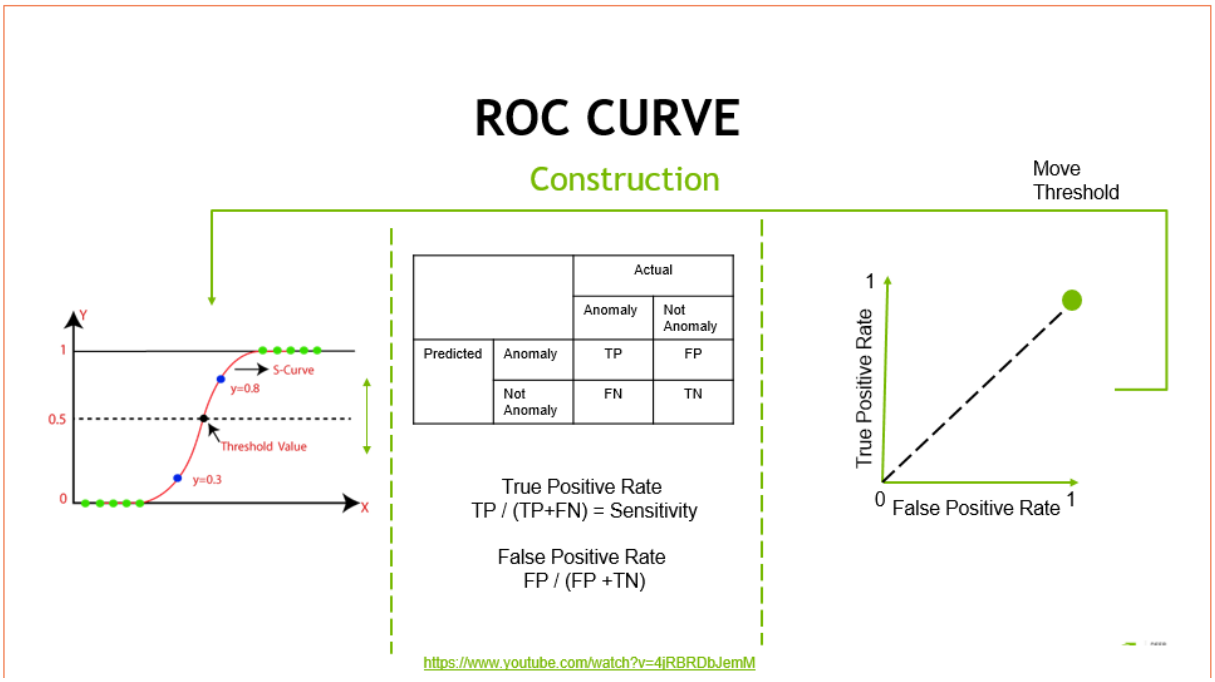


Figure 5 Definition of the ROC Curve

The second most common method for showing performance is to produce a confusion matrix, which is in the middle of Figure 5. It is a 2x2 table which bins the classification decisions of the algorithm. A perfect classifier has all the samples along the left diagonal and 0's on the off-diagonal. A labelled definition of the confusion matrix is shown in

Figure 6. The True Positives (TP) and True Negatives (TN) are along the main diagonal while the misclassifications False Positives (FP) and False Negatives (FN) are on the off diagonal. From these 4 terms, we can compute several metrics that are commonly used to describe machine learning performance. Some of the key metrics are given in Figure 7.

		Predicted condition	
	Total population = P + N	Positive (PP)	Negative (PN)
Actual condition	Positive (P)	True positive (TP)	False negative (FN)
	Negative (N)	False positive (FP)	True negative (TN)

Figure 6 Definition of a Confusion Matrix

sensitivity, recall, hit rate, or true positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

specificity, selectivity or true negative rate (TNR)

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

precision or positive predictive value (PPV)

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} = 1 - \mathrm{FDR}$$

negative predictive value (NPV)

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}} = 1 - \mathrm{FOR}$$

miss rate or false negative rate (FNR)

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{P}} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}} = 1 - \mathrm{TPR}$$

fall-out or false positive rate (FPR)

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = 1 - \mathrm{TNR}$$

false discovery rate (FDR)

$$\mathrm{FDR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TP}} = 1 - \mathrm{PPV}$$

Prevalence

$$\frac{\mathrm{P}}{\mathrm{P} + \mathrm{N}}$$

accuracy (ACC)

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

balanced accuracy (BA)

$$\mathrm{BA} = \frac{TPR + TNR}{2}$$

F1 score

is the harmonic mean of precision and sensitivity:

$$\mathrm{F_1} = 2 \times \frac{\mathrm{PPV} \times \mathrm{TPR}}{\mathrm{PPV} + \mathrm{TPR}} = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

Figure 7 Various metrics derived from confusion matrix

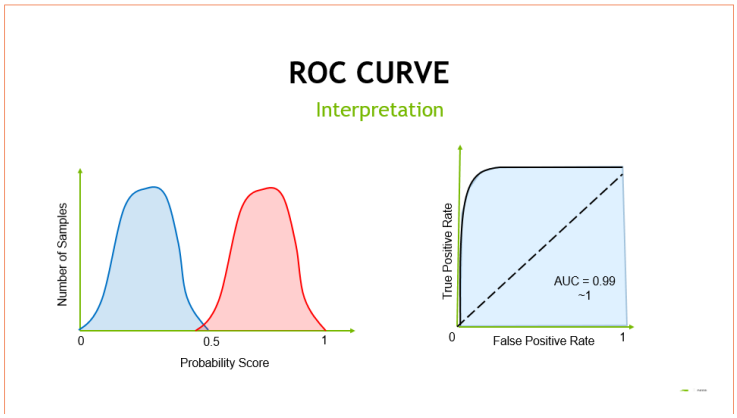


Figure 8 ROC Curve and AUC for two highly separable classes

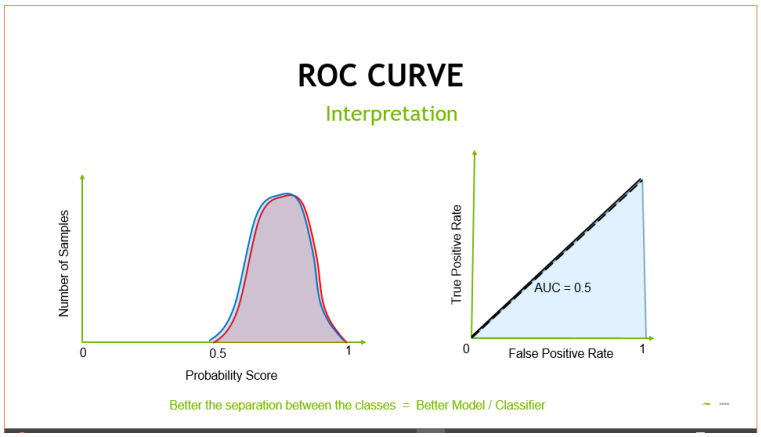


Figure 9 ROC and AUC for two non-separable classes

The area under the ROC curve or AUC is a great metric for determining how well your classification model is performing, even in the case of imbalanced classes. A score of 1 means your model is performing perfectly (a near-perfect model is shown in Figure 8), while a score of .5 means that your model is the same as randomly guessing. This is depicted in Figure 9 showing the line y=x as the ROC curve and the blue area below that curve is the AUC or .5.

For computational performance, we used the wall time for processing packets in inferences per second.

The QUBO performance on the SVM algorithm we measured as metrics:

1. objective function value
2. Round-trip time taken for overall optimization (QUBO processing time + Network transit time+ CPU processing time)

3. Confusion matrix, accuracy, precision, sensitivity, Matthew's Correlation coefficient, ROC and AUC curves
4. Wall time for processing packets in inferences per second

Technical Modeling

Next, we explain the derivation of the autoencoder algorithm and any transformations required for implementation on Groq HW for anomaly detection as well as the generative adversarial network anomaly detection method and finally the QUBO Support Vector Machine (SVM) solver.

Autoencoder Model for Anomaly Detection

Here we will show how the data preprocessed above goes through one final step before it is ready for application to train. We will then describe the training process. Finally, we will provide performance assessment against the test set and will discuss the sensitivity of the base rate to the performance with an autoencoder.

In the real-world, *labeled* data can be expensive and hard to come by. Especially with network security, zero-day attacks can be the most challenging and also the most important attacks to detect. Since, by definition, these attacks are happening for the first time, there will be no way to have labels from them.

So how do we approach *this* problem?

For starters, we could have security analysts investigate the network packets and label anomalous ones. But that solution doesn't scale and our models might have difficulty identifying attacks that haven't occurred before.

Our solution *needs to use* "unsupervised learning." Unsupervised learning is the class of machine learning and deep learning algorithms that enable us to draw inferences from our dataset without labels.

In this lab we will use autoencoders to detect anomalies in the KDD99 dataset. There are a lot of advantages to using autoencoders for detecting anomalies. One main advantage is that AEs can learn non-linear relationships in the data.

Autoencoders are a subset of neural network architectures shown in Figure 10 where the output dimension is the same as the input dimension. Autoencoders have two networks, an encoder and a decoder. The encoder encodes its input data into a smaller dimensional space, called the latent space. The decoder network tries to reconstruct the original data from the latent encoding. Typically, the encoder and decoder are symmetric, and the latent space is a bottleneck. The autoencoder has to learn essential characteristics of the data to be able to do a high-quality reconstruction of the data during decode.

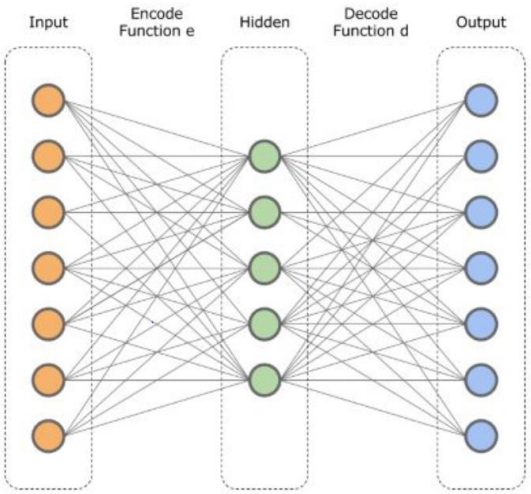


Figure 10 Depiction of typical autoencoder

While we will not be using the labels in the KDD99 dataset explicitly for model training, we will be using them to evaluate how well our model is doing at detecting the anomalies. We will also use the labels to see if the AE is embedding the anomalies in latent space according to the type of anomaly.

Note that we will be using Keras as the deep learning framework for this lab. Keras is an open source neural network library written in Python and it is designed to enable fast experimentation with deep neural networks.

Using autoencoders, we will explore the questions of *how rare is rare enough for an anomaly?* And *how does that impact our ability to identify multiple classes of anomalies?*.

In the cell below, choose either use 1% or 5% anomaly in the data set by setting the *pct_anomalies* parameter to .01 or .05 respectively. We'll use 1% first.

```
pct_anomalies = .01
!python preprocess_data.py --pct_anomalies $pct_anomalies
filename = './preprocessed_data_full.pkl'
input_file = open(filename,'rb')
preprocessed_data = pickle.load(input_file)
input_file.close()
```

Recall that we have constructed train and test sets where we removed most of anomalous data in the KDD99 dataset. This lets us simulate a more realistic anomaly detection problem where anomalies only comprise a small percentage of the data. We also trained a label encoder on the anomalous labels. This will allow us to go back and forth between labels and their encoded values.

Most of the data preprocessing has already been done earlier as described. We one-hot encoded the categorical variables and separated the labels off from the input data, then scaled the data between 0 and 1.

Next we will chose the hyperparameters for the Keras autoencoder model.

- **batch_size**: this determines how many datapoints we use for each gradient update. Choosing a large batch size will make the model train faster but it might not result in the best accuracy or generalization.
- **latent_dim**: this determines the size of our bottleneck. Higher values add network capacity while lower values increase the efficiency of the encoding.

- **max_epochs**: should be high enough for the network to learn from the data, but not so high as to overfit the training data or diverge to a worse result

```
input_dim = x_train.shape[1]
# model hyperparameters
batch_size = 512
latent_dim = 4
max_epochs = 10
```

Below in Figure 11 we provide a detailed layer by layer buildup of the autoencoder along with the number of nodes going into a layer and the number of nodes coming out of the layer. Note that there is a dropout layer in between every dense node layer. We choose a dropout rate of 10% as a way to keep the autoencoder from overtraining on the data. It is considered a best practice, but the dropout rate can vary quite a bit from application to application for optimum performance. It has been seen as high as 40% in some problems.

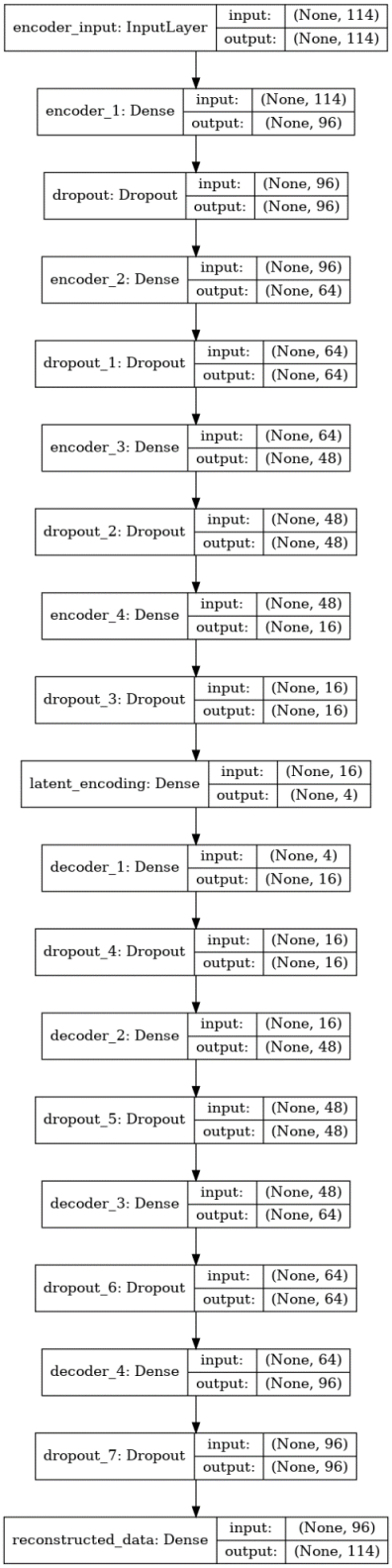


Figure 11 Graphical visualization of autoencoder layers

To actually train the autoencoder, we run the following Python Keras command 'fit' method:

```
train_history = autoencoder_model.fit(x_train, x_train,
    shuffle=True,
    epochs=max_epochs,
    batch_size=batch_size,
    validation_data=(x_test, x_test),
    callbacks=[tensorboard_callback])
```

Notice that *x_train* appears twice. The second occurrence is the same as saying '*y_desired'*. That is because we are using as a metric the difference between what goes into the encoder and what comes out the decoder and have earlier selected the mean square error as the key performance metric we will be optimizing against. Ideally, coming out of the autoencoder would be the values identical to the input values, *x_train*. Also, notice there is no use of the labels in any of the training (*y_test*, *y_train*). That means that the *x_train* data contains both normal data and anomalous data in a ratio of 100:1.

```
plt.plot(train_history.history['loss'])
plt.plot(train_history.history['val_loss'])
plt.legend(['loss on train data', 'loss on validation data'])
```

Let's inspect the loss on the train and validation sets. You should see the loss on the training data and the loss on the validation data converging towards zero. Notice that the training loss is actually higher than the validation loss. That's because when we train the network, we are using dropout, which again, "is a way to control overfitting by randomly omitting subsets of features at each iteration of a training procedure." When we validate, we remove the dropout, which gives our network its full strength. The x axis represents the number of training epocs.

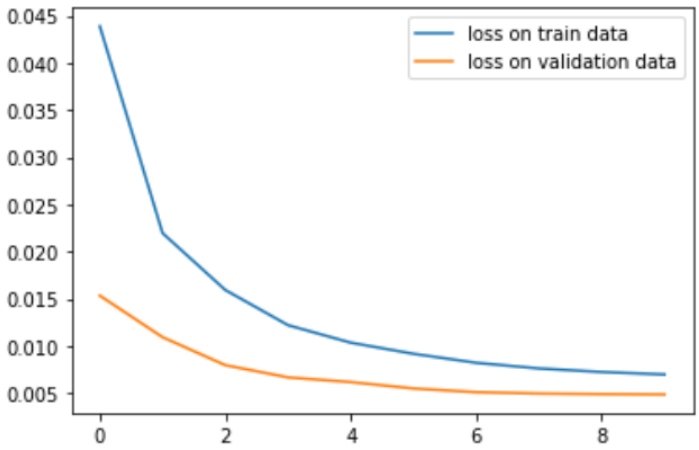


Figure 12 Loss on training and validation data

Generative Adversarial Network Model for Anomaly Detection

Second, we explain the derivation of the GAN algorithm and any transformations required for implementation on Groq HW for anomaly detection.

Here we will show how the data above goes through one final step before it is ready for application to train. We will then describe the training process. Finally, we will provide performance assessment against the test set and will discuss the sensitivity of the base rate to the performance with a GAN. In the previous section, we tried our hand at unsupervised anomaly detection using deep autoencoders on the KDD-99 network intrusion dataset.

We addressed the issue of unlabeled training data through the use of deep autoencoders in the second section. However, unsupervised methods such as PCA and autoencoders tend to be effective only on highly correlated data such as the KDD dataset.

"Adversarial training (also called GAN for Generative Adversarial Networks), and the variations that are now being proposed, is the most interesting idea in the last 10 years in ML, in my opinion.". Yann LeCun, 2016.

What do GANs bring to the table and how are they different from Deep Autoencoders?

GANs are generative models that generate samples similar to the training dataset by learning the true data distribution. So instead of compressing the input into a latent space and classifying the test samples based on the reconstruction error, we actually train a classifier (called the discriminator) that outputs a probability score of a sample being Normal or Anomalous. As we will see later in the lab, this has positioned GANs as very attractive unsupervised learning techniques.

GANs can be pretty tough to train and improving their stability is an active area of research today.

Preprocessing Data For GAN Use

First we start off and load the data file that is in Pickle format and convert it to a Pandas dataframe. Next we, examine during the conversion to a 1% anomaly rate data file, what type of anomalies are present in the file.

```
np.unique(y_train)
array([ 0,  5,  9, 10, 11, 15, 17, 18, 20, 21])
```

This is compared to the complete set of anomaly types in the original file:

```
#Obtain the class number for Normal entries
pd.DataFrame(le.classes_, columns = ['Type'])
```

0	back.	12	perl.
1	buffer_overflow.	13	phf.
2	ftp_write.	14	pod.
3	guess_passwd.	15	portsweep.
4	imap.	16	rootkit.
5	ipsweep.	17	satan.
6	land.	18	smurf.
7	loadmodule.	19	spy.
8	multihop.	20	teardrop.
9	neptune.	21	warezclient.
10	nmap.	22	warezmaster.
11	normal.		

Figure 13 Complete List of Anomaly Types in KDD99

As we can see by comparing the figure with the array above it, we have omitted [1,2,3,4,6,7,8,12,13,14,16,19,22] anomaly types from our existing data file. Now we do a hard binary conversion of the data labels. And we will now split the dataset into normal and anomalous data. We will need to do this in order to be able to train GANs to generate **Normal packets only** and then predict the anomaly based on the Discriminator output.

```
# Converting labels to Binary
y_test[y_test != 11] = 1
y_test[y_test == 11] = 0
y_train[y_train != 11] = 1
y_train[y_train == 11] = 0
#Subsetting only Normal Network packets in our training set
temp_df = x_train.copy()
temp_df['label'] = y_train
temp_df = temp_df.loc[temp_df['label'] == 0]
temp_df = temp_df.drop('label', axis = 1)
x_train = temp_df.copy()
```

Finally, we scale the input training data between 0 and 1 before feeding it to the model.

Summary of the datasets:

- The Training set consists of only normal network packets.
- The Testing set comprises a small number of anomalous network packets of about 1%, reflecting what we see in the real world.

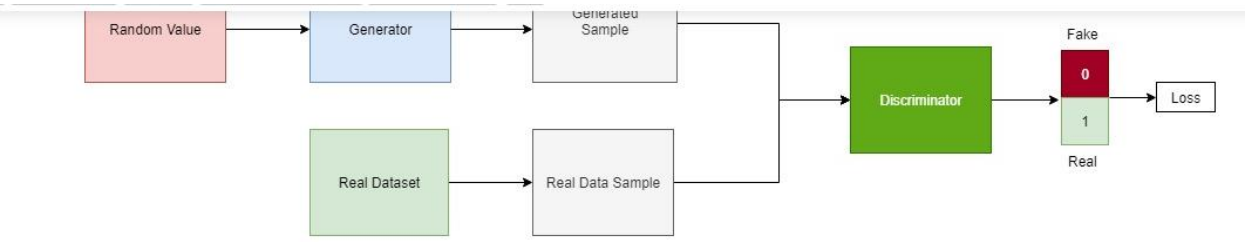
```
# check how many anomalies are in our Testing set

print('Number of Normal Network packets in the Training set:',
x_train.shape[0])

print('Number of Normal Network packets in the Testing set:',
collections.Counter(y_test)[0])
print('Number of Anomalous Network packets in the Testing set:',
collections.Counter(y_test)[1])
```

Number of Normal Network packets in the Training set: 729620
Number of Normal Network packets in the Testing set: 243161
Number of Anomalous Network packets in the Testing set: 2466

The Generative Adversarial Network

The GAN consists of two networks namely:

- The generator G that produces fake samples
- The discriminator D that that receives samples from both G and the dataset.



During Training the two networks have competing goals. The generator tries to fool the

discriminator by outputting values that resemble real data and the discriminator tries to become better at distinguishing between the real and fake data.

Mathematically, this means that the Generator's weights are optimized to maximize the probability that fake data is classified as belonging to the real data. The discriminator's weights are optimized to maximize the probability that the real input data is classified as real while minimizing the probability of fake input data being classified as real.

Optimality is reached when the generator produces an output that the discriminator cannot concretely label as real or fake and this happens when either of the networks cannot improve anymore.

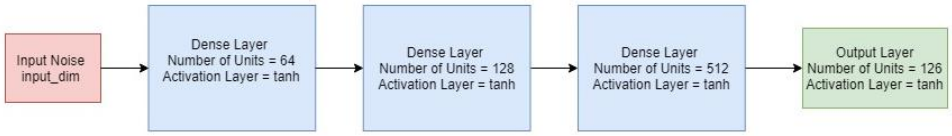
We will be train our GAN on normal network packets. The generator inputs noise and as training progresses the GAN learns the mapping between these random values to the input distribution. The discriminator outputs a score of how likely the generated output resembles the real data.

The Generator is used to synthesize fake data points. It consists of 5 Dense Layers with a hyperbolic tangent activation function (tanh), which forces all about between -1 and 1, and uses binary cross-entropy for calculating the generator loss. Binary cross-entropy loss measures the performance of a two class classification model whose output is a probability value between 0 and 1. A perfect model would have a loss of 0.

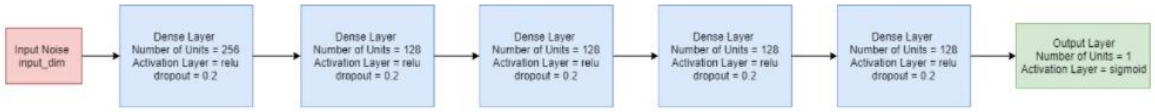
The Discriminator basically outputs the score of a sample belonging to the real dataset or the synthetic dataset. It consists of 6 dense layers-each followed by a dropout layer to help prevent overfitting. The sigmoid activation function is applied to the final layer to obtain a value in the range 0 to 1.

Training the Model

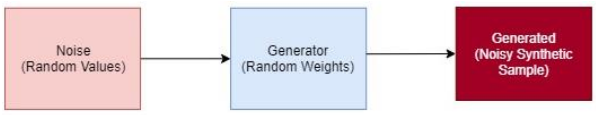
The generator first predicts on a batch of noise samples. As the generator has randomly initialized weights initially, the output of the generator at this stage is nothing but meaningless values.



The Discriminator inputs a stack of samples - the first half of which is the output of the generator and the second half is a batch of data samples from the real dataset. We train the Discriminator on this stack with the target labels 0 (Fake) for half the stack and 1 for the second half of the stack. The result of this is that the Discriminator is able to distinguish between the Real and Fake samples.



The weights of the discriminator are frozen by setting the trainable parameter to False. To train the Generator, we first feed it random noise and let the entire GAN output a probability with the Discriminator weights remaining frozen. As expected this value would be less than 0.5 since the Discriminator was previously set to output a value close to 0 if the input was not genuine.



Now comes the trick. We tell the GAN that the expected output is 1. This results in the errors being backpropagated only to the Generator. With every sample in the batch the generator's weights are tuned such that the output of the GAN is close to 1, meaning the Generator is now learning to produce samples that resemble the real data. This process loops back to the first step for each batch in the training set shown in Figure 14.

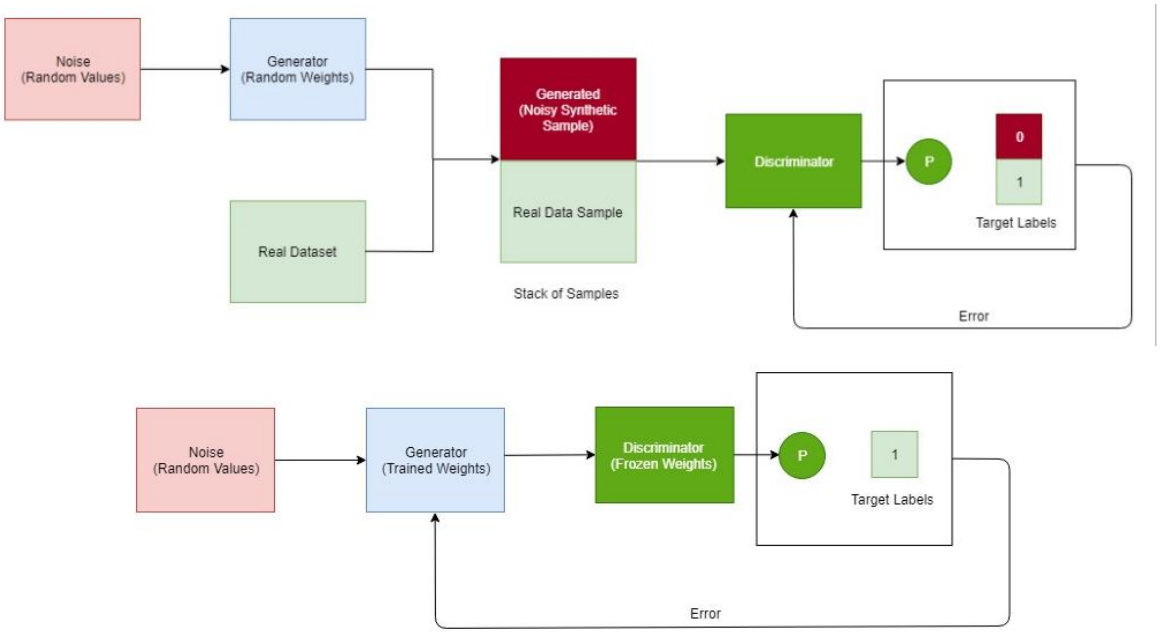


Figure 14 Training of the Discriminator and Generator Neural Networks

Let's look at the training losses for the Generator and the Discriminator components of the GAN

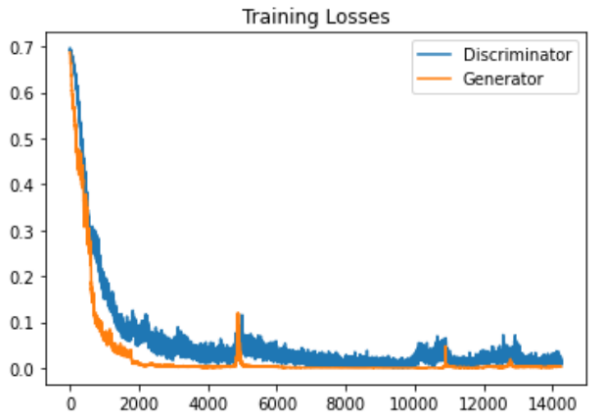


Figure 15 Losses for Training the GAN Components

We see that compared to the autoencoder training losses, there is much more noise while the overall training loss tends to 0 in Figure 15. Notice how even towards the end, there are bumps of more loss and the convergence is not strictly monotonic. This is one of the aspects that makes a GAN so difficult to train.

What was the result of all the training we did?

We now have a generator that can input a random seed value and produce an output that closely resembles the data it was trained on.

The Discriminator that we trained ended up being a very powerful classifier that can tell if a sample point is representative of the true data distribution it was trained on or not and hence can be used for Anomaly Detection. Once training is complete, we have no further need for the trained generator. We discuss the performance results of the GAN in the next major section.

QUBO Support Vector Machine (SVM) Model For Anomaly Detection

Finally, we explain the derivation of the telemetry algorithm and its implementation in QUBO form using the Hamiltonian.

Hamiltonian: The energy function that is minimized by annealer. It encodes the objective and constraints into one function. The equality constraint (supply allocation) becomes a quadratic term to account for the deviations of variables to either side of the target value. The inequality constraint (demand threshold) also becomes a quadratic term but encoded with the help of slack variables.

Background on Support Vector Machines (SVMs) A SVM learns its parameters from a set of annotated training samples

$$D = \{\boldsymbol{x}_n, y_n : n = 0, \ldots, N - 1\}$$

with $\boldsymbol{x}_n \in R^D$ being a feature vector and y_n its label.

A SVM separates the samples of different classes in their feature space by tracing maximum margin hyperplanes. The training consists of solving a quadratic programming (QP) problem

$$L = \frac{1}{2}\sum_{nm} \alpha_n \alpha_m y_n y_m k\,(\mathbf{x}_n, \mathbf{x}_m) - \sum_n \alpha_n \qquad (1)$$

$$\text{subject to} \quad 0 \le \alpha_n \le C \quad \text{and} \quad \sum_n \alpha_n y_n = 0 \qquad (2)$$

For N coefficients $\alpha_n \in \mathbf{R}$, where C is a regularization parameter and k(., .) is a kernel function that enables a SVM to compute non-linear decision functions (by means of the kernel trick). The type of kernel function which is most commonly used is the Radial Basis Function: rbf = $k(\boldsymbol{x}_n, \boldsymbol{x}_m) = e^{-\gamma\|\boldsymbol{x}_n - \boldsymbol{x}_m\|^2}$. The SVM decision boundary is based on the samples corresponding to $\alpha_n \neq 0$ (i.e., support vectors). A typical solution often contains many α_n = 0. The prediction for an arbitrary sample **x** $\in R^D$ can be made by evaluating the decision function (i.e., signed distance between the sample **x** and the decision boundary)

$$f(\mathbf{x}) = \sum_n \alpha_n y_n k\,(\mathbf{x}_n, \mathbf{x}) + b \qquad (3)$$

where the bias b can be computed by

$$b = \frac{\sum_n \alpha_n\,(C - \alpha_n)\,[y_n - \sum_m \alpha_m y_m k\,(\mathbf{x}_m, \mathbf{x}_n)]}{\sum_n \alpha_n\,(C - \alpha_n)} \qquad (4)$$

The class label for **x** predicted is $\bar{y}$= sign(f (**x**)).

Quantum SVM

The DW2000Q QA requires the SVM training to be formulated as a Quadratic Unconstrained Binary Optimization (QUBO) problem which is defined as the minimization of the energy function:

$$E = \sum_{i \le j} a_i Q_{ij} a_j \qquad (5)$$

with $a_i \in \{0, 1\}$ the binary variables of the optimization problem, and Q the QUBO weight matrix (i.e., an upper-triangular matrix of real numbers). Since the solution of Eqs. (1)-(2)

consists of real numbers $\alpha_n \in \mathbf{R}$ and Eq.(4) can only computes discrete solutions, the following encoding is used:

$$\alpha_n = \sum_{k=0}^{K-1} B^k a_{Kn+k} \qquad (6)$$

where $a_{Kn+k} \in \{0, 1\}$ are binary variables, K is the number of binary variables to encode α_n, and B is the base used for the encoding. The formulation of the QP of Eqs. (1)-(2) as QUBO is obtained through the encoding of Eq. (6) and the introduction of a multiplier ξ to include the first constraint of Eq. (2) as a squared penalty term:

$$E = \frac{1}{2}\sum_{nmkj} a_{Kn+k}a_{Km+j}B^{k+j}y_n y_m k(\mathbf{x}_n, \mathbf{x}_m) - \sum_{nk} B^k a_{Kn+k} + \xi\left(\sum_{nk} B^k a_{Kn+k} y_n\right)^2$$

$$= \sum_{n,m=0}^{N-1}\sum_{k,j=0}^{K-1} a_{Kn+k}\tilde{Q}_{Kn+k,Km+j}a_{Km+j} \qquad (8)$$

where $\tilde{Q}$ is a matrix of size KN × KN given by

$$\tilde{Q}_{Kn+k,Km+j} = \frac{1}{2}B^{k+j}y_n y_m\left(k(\mathbf{x}_n, \mathbf{x}_m) + \xi\right) - \delta_{nm}\delta_{kj}B^k \qquad (9)$$

Since $\tilde{Q}$ is symmetric, the upper-triangular QUBO matrix Q is defined by $Q_{ij} = \tilde{Q}_{ij} + \tilde{Q}_{ji}$ for i < j and $Q_{ii} = \widetilde{Q_{ii}}$. The second constraint of Eq. (2) is automatically included in Eq. (8) through the encoding given in Eq. (6), since the maximum for α_n is given by

$$C = \sum_{k=1}^{K} B^k \qquad (10)$$

The last step required to run the optimization on the DW2000Q QA is the embedding procedure. This is necessary because the QUBO problem given in Eq. (5) includes some couplers $Q_{ij} \neq 0$ between qubit i and qubit j for which no physical connection exists on the chip (i.e., constraint of the Chimera topology of the DW2000Q quantum processor). The embedding increases the number of logical connections between the qubits. When no embedding can be found, the number of nonzero couplers n_{cpl} is the parameter that can be reduced until an embedding is found. The DW2000Q QA computes a variety of close-

to-optimal solutions (i.e., different coefficients $\{\alpha_n\}^{(i)}$ obtained from Eq. (6)). Many of these solutions may have a slightly higher energy than the global minimum $\{\alpha_n\}^*$ that can be found by the classical SVM. However, these solutions can still solve the classification problem for the training data. For each run on the DW2000Q QA, the 20 lowest energy samples from 10,000 reads are kept.

Quantum SVM: Training Phase

To overcome the problem of the limited connectivity of the Chimera graph of the DW2000Q QA the whole training set is split into small disjoint subsets $(D)^{(train,l)}$ of 40 samples, with l = 0, .., int(N/40). The strategy is to build an ensemble of quantum weak SVMs (qeSVMs) where each classifier is trained on $(D)^{(train,l)}$. This is achieved in two steps. First, for each subset $(D)^{(train,l)}$ the twenty best solutions from the DW2000Q QA (i.e.,qSVM(B, K, ξ, γ)#i for i = 0, ..., 19) are combined by averaging over the respective decision function $f^{i,j}(\boldsymbol{x})$ (see Eq. (3)).

Since the decision function is linear in the coefficients and the bias $b^{(l,i)}$ is computed from $\alpha_N^{(l,i)}$ via Eq. (4), this procedure effectively results in one classifier with an effective set of coefficients $\alpha_N^{(l)} = \sum_i \alpha_N^{(l,i)} / 20$ and bias $b^l = \sum_i b^{(l,i)} /20$. Second, an average is made over the int(N/40) subsets. Note, however, that the data points $(\boldsymbol{x}_N^{(l)}, \boldsymbol{y}_N^{(l)}) \in (D)^{(train,l)}$ are now different for each l. The full decision function is

$$F(\mathbf{x}) = \frac{1}{L}\sum_{nl} \alpha_n^{(l)} y_n^{(l)} k\left(\mathbf{x}_n^{(l)}, \mathbf{x}\right) + b, \qquad (11)$$

where $b = \sum_l b^{(l)} /L$. As before, the decision for the class label of a point **x** is obtained through $\bar{t} = sign\big(F(\boldsymbol{x})\big)$.

Key Performance Parameter Comparison

Metrics Output Examination for Objectives 1: the Autoencoder

Let's quickly look at the reconstruction scores

```
# store the reconstruction data in a Pandas dataframe
anomaly_data = pd.DataFrame({'recon_score':reconstruction_scores})
# if our reconstruction scores our normally distributed we can use their statistics
anomaly_data.describe()
```

Table 1 Reconstruction Score and Statistics

	recon_score
count	245627.000000
mean	0.004885
std	0.008157
min	0.000061
25%	0.001334
50%	0.002371
75%	0.004776
max	1.494917

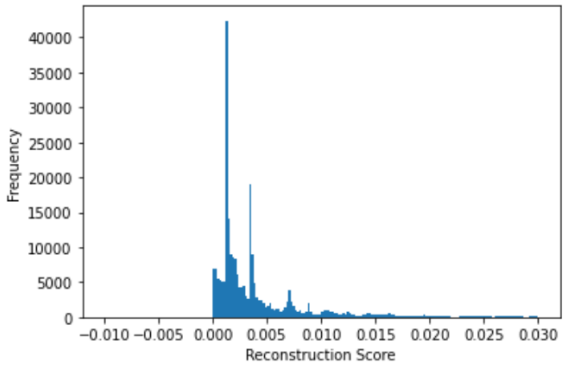


Figure 16 Histogram of Reconstruction Scores

We now use the y labels just in the process of score validation. We map a value of 'normal' to 0 and 'anomalous' to 1. If you recall in the preprocessing, normal had a value of 11, and anomalies had all other values between 0 and 22.

```
binary_labels = convert_label_to_binary(le, y_test)


# add the binary labels to our anomaly dataframe
anomaly_data['binary_labels'] = binary_labels


# let's check if the reconstruction statistics are different for labeled anomalies
# convert our labels to binary
```

	recon_score							
	count	mean	std	min	25%	50%	75%	max
binary_labels								
0	243161.0	0.004382	0.006320	0.000061	0.001334	0.002332	0.004592	1.494917
1	2466.0	0.054481	0.014256	0.007956	0.047247	0.047247	0.047247	0.083877

Figure 17 Values of key statistics for normal and anomalous samples

We can see from Figure 17 above that the anomalous data has a mean reconstruction score of 0.05 while the normal data has a score of **~0.004**. This is a good sign that our autoencoder has learned to reconstruct normal data but fails to reconstruct anomalous data.

```
#Let's compute the Area Under the Curve
fpr, tpr, thresholds = roc_curve(binary_labels, reconstruction_scores)
roc_auc = auc(fpr, tpr)
print(roc_auc)
```

1.000

We are able to generate a perfect AUC score.

So we know there is almost an order of magnitude difference in reconstruction error between normal and anomalous samples. But what is the optimal dividing point between the means of the two groups? There are number of ways to set thresholds. We give two examples below:

```
# We can pick the threshold based on maximizing the true positive rate (tpr)
# and minimizing the false positive rate (fpr)
optimal_threshold_idx = np.argmax(tpr - fpr)
optimal_threshold = thresholds[optimal_threshold_idx]
print(optimal_threshold)
```

0.0342

```
# Or we assume our reconstructions are normally distributed and label anomalies as those
# that are a number of standard deviations away from the mean
recon_mean = np.mean(reconstruction_scores)
recon_stddev = np.std(reconstruction_scores)

stats_threshold = recon_mean + 5*recon_stddev
print(stats_threshold)
```

0.04566

Analyzing the confusion matrix, we see that we get optimal results. The algorithm is labeling most of the normal data and anomalous data correctly while mislabeling very infrequently.

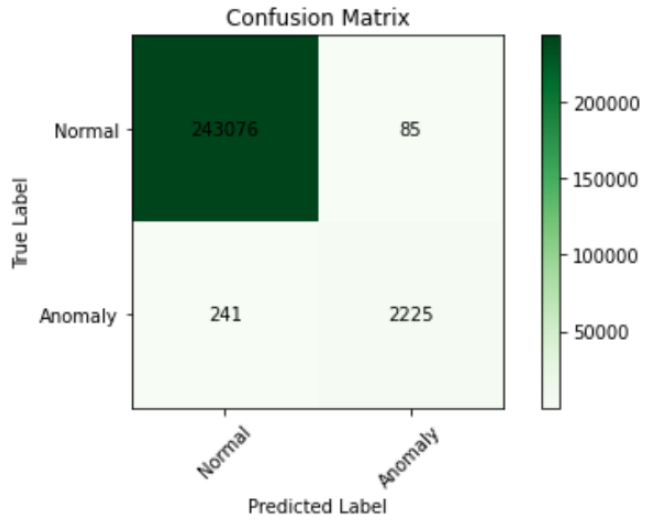


Figure 18 The Confusion Matrix for 1% Anomalies

We have a handful of performance metrics available to us for comparison. These include:

Sensitivity = TP/(TP+FN) = 243076/(243076+85) = 0.999

Precision = TP/(TP+FP) = 243076/(243076+241) = 0.999

Specificity = TN/(TN+FP) = 2225/(2225+241) = 0.902

Accuracy = TP+TN/(P+N) = (243076+2225)/(243076+2225+241+85) = 0.998

F1 Score = 2TP/(2TP+FP+FN)= 2*243076/(2*243076+241+85) = 0.999

Inferences per second = 126,000

The 5% Anomaly Case for Autoencoders

Now we will examine the AE performance when the anomaly rate is 5%. First, we print the reconstruction scores from the 5% data

Table 2 AE Reconstruction Statistics for 5% Case

	recon_score
count	255355.000000
mean	0.005942
std	0.008564
min	0.000169
25%	0.001725
50%	0.002437
75%	0.008025
max	0.906676

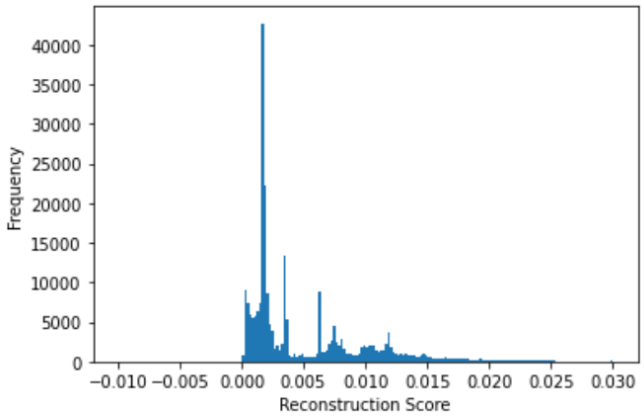


Figure 19 Histogram with 5% anomaly rate

Then we print the mean reconstruction error score and other statistics for the normal and anomalous data samples in Table 3. At first glance, there appears a wide separation between the mean reconstruction errors for normal and anomalous samples (.00516 versus .0214 for an anomaly) and this looks like the separation in the 1% case, but that case is actually (.0043 versus .054) more separated.

Table 3 Reconstruction Error Statistics For 5% Anomaly Rate (Normal=0, Anomaly =1)

	recon_score							
	count	mean	std	min	25%	50%	75%	max
binary_labels								
0	243134.0	0.005162	0.005724	0.000169	0.001703	0.002202	0.008003	0.906676
1	12221.0	0.021452	0.025061	0.001861	0.006252	0.006252	0.031609	0.073151

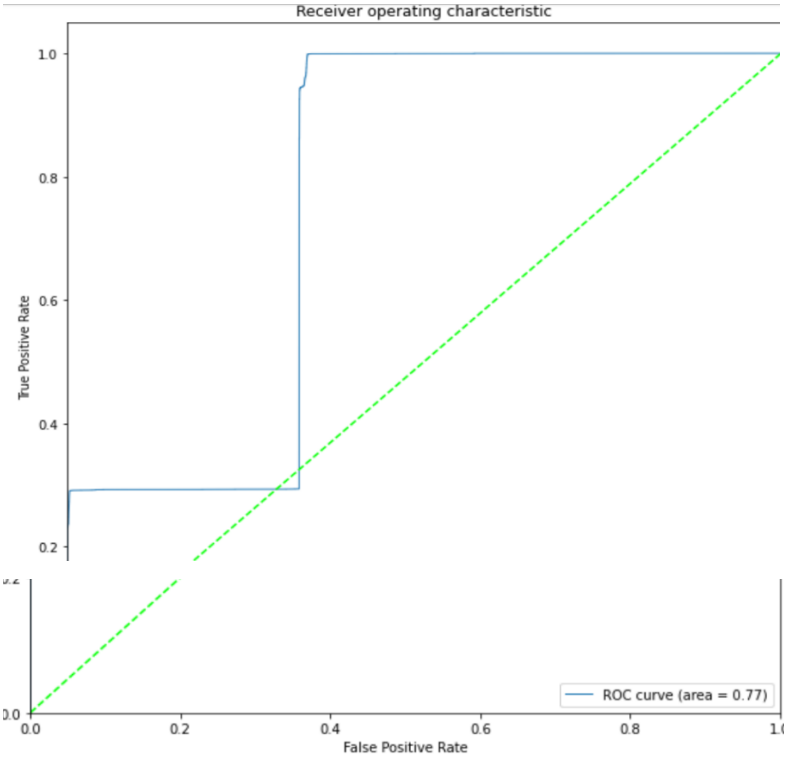


Figure 20 ROC for 5% Anomaly Data Set

With a little bit of manipulation, we can maximize the True Positive Rate for a given threshold while minimizing the False Positive Rate (FPR).

```
# We can pick the threshold based on maximizing the true positive rate (tpr)
# and minimizing the false positive rate (fpr)
optimal_threshold_idx = np.argmax(tpr - fpr)
optimal_threshold = thresholds[optimal_threshold_idx]
print(optimal_threshold)
```

0.005237637

```
# Or we assume our reconstructions are normally distributed and label anomalies as those
# that are a number of standard deviations away from the mean
recon_mean = np.mean(reconstruction_scores)
recon_stddev = np.std(reconstruction_scores)
stats_threshold = recon_mean + 5*recon_stddev
print(stats_threshold)
```

0.0487624048255384

We use the second threshold and generate a new confusion matrix.

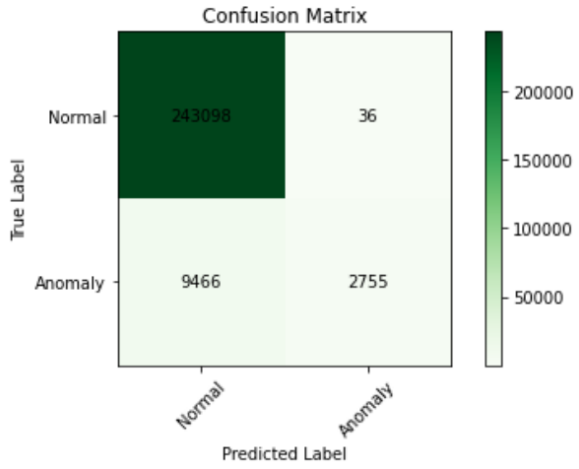


Figure 21 Confusion Matrix for 5% Anomaly Rate

Comparing the off-diagonal terms with the 5% Anomaly Rate in Figure 21, the number of normal samples labelled as Anomalies is 36 and the number of Anomalies labelled as normal is 9466. Compare this to the 1% rate of 85 and 241. Taking normal samples as positives and anomalies as negatives, we see that the **false positive rate is almost 40 times higher**! And from the ROC curve, we see that the AUC drops significantly to 0.77 from a perfect score of 1.0.

This leads us to a very important take-away: in support of autoencoders, it is a truly unsupervised method. However, the classifier performance is a strong function of the anomaly rate in the data. This should make intuitive sense--the job of the autoencoder is to learn to reconstruct the samples provided in the training data. If too many of those are both normal samples and anomaly samples, then the AE will begin to be able to reconstruct some anomalies with errors that are closer to that of normal samples. This is then a double-edged sword as it's true that it's unsupervised, but there are key hyperparameters such as base rate which have to be learned in the model validation stage to find the optimum performance.

Comparing Figure 16 and Figure 19, one sees that there looks like a larger separation between the normal and anomalous samples for the 5% case. But what has happened is

that many more of the anomalies are closer to the mean of the normal sample and this is what drives the factor of 40.

Metrics Output Examination for Objective 2: the GAN

Let us calculate the mean score for normal and anomalous samples in our test set. Ideally, we would like to see a score close to 1 for normal samples and 0 for anomalous samples. This would mean our classifier is doing well in distinguishing between the 2 classes.

```
pd.options.display.float_format = '{:20,.7f}'.format
results_df = pd.concat([pd.DataFrame(results),pd.DataFrame(y_test)], axis=1)
results_df.columns = ['results','y_test']
print ('Mean score for normal packets :', results_df.loc[results_df['y_test'] == 0, 'results'].mean() )
print ('Mean score for anomalous packets :', results_df.loc[results_df['y_test'] == 1, 'results'].mean())
```

Mean score for normal packets : 0.998
Mean score for anomalous packets : 0.0853

Note that these scores are very close to 1 for normal packets and close to .08 for anomalous packets which is a very encouraging result. But how exactly do we identify our Anomalies?

Although there are several ways to do this, let us use a more straightforward way for detection. Remember 1% of our test set comprised of anomalies. So, the lowest 1% of the scores should ideally constitute anomalies. Let us test our hypothesis below.

```
#Obtaining the lowest 1% score
per = np.percentile(results,1)
y_pred = results.copy()
y_pred = np.array(y_pred)
#Thresholding based on the score
inds = (y_pred > per)
inds_comp = (y_pred <= per)
```

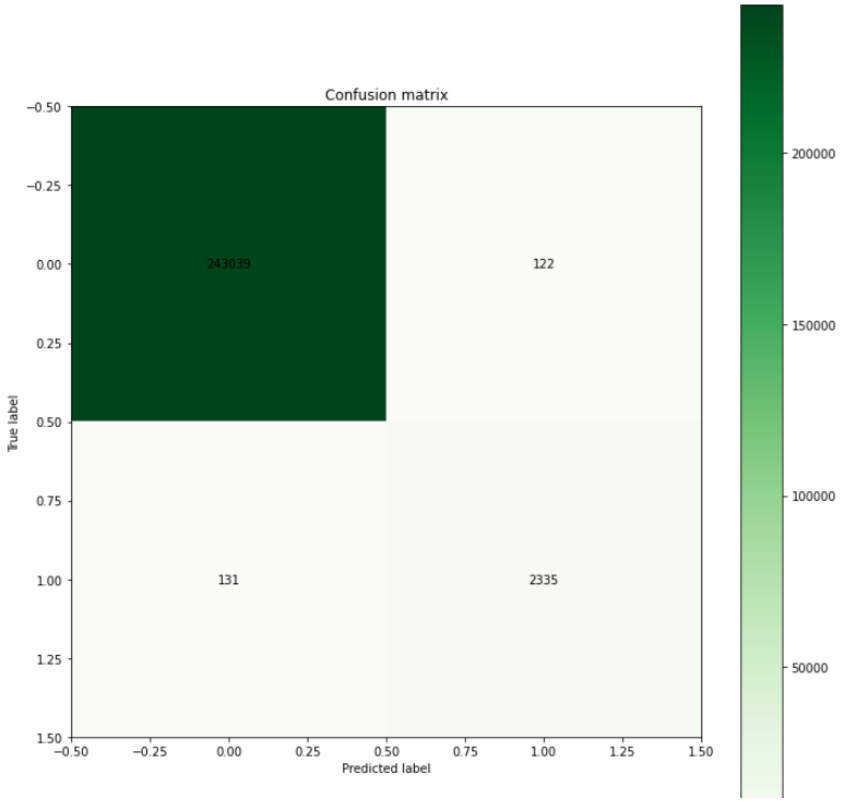


Figure 22 Confusion Matrix for GAN Test Results

We calculate several metrics based on the confusion matrix.

Accuracy Score : 0.9989

Precision : 0.9503

Recall : 0.9469

F1 : 0.9486

All of these results are excellent. And in the Figure 23 below, we see the Area Under the Curve is .97.

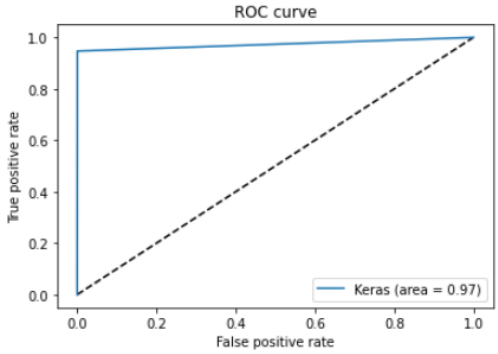


Figure 23 The GAN ROC and AUC

Results for 5% Anomaly Rate with GAN

Figure 24 Training Losses with Batch Number with 5% Anomaly Training Data

The figure above in Figure 24 shows that the generator seems to have less batch to batch variation than in the 1% case. When we look at the end results, we find that the normal/anomalous sample mean is:

Mean score for normal packets : 0.9994

Mean score for anomalous packets : 0.3199

The normal score is still excellent and in line with the 1% results. But the anomalous packet results have come up considerably from their .08 mean score with the 1% training.

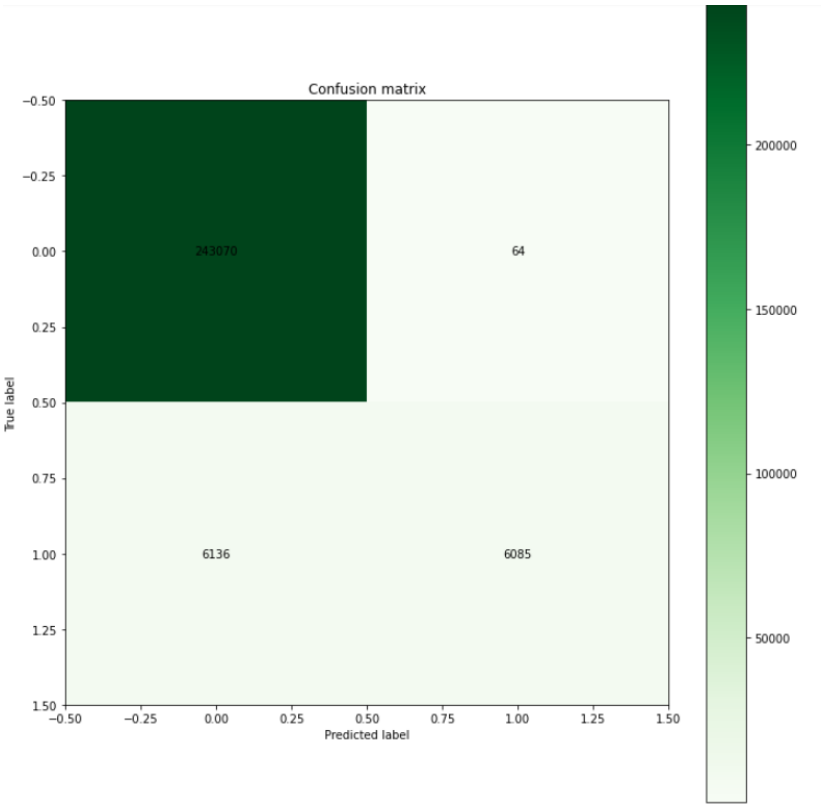


Figure 25 Confusion Matrix for GAN Testing with 5% Anomaly in Test Data

Here are some key metrics for 5% generated from the confusion matrix above in Figure 25:

Accuracy Score : 0.9757

Precision : 0.9895

Recall : 0.4979

F1 : 0.66249

GAN Performance Summary

- We successfully employed state of the art Generative Adversarial Networks for anomaly detection on high dimensional data such as the KDD dataset.
- The GAN is particularly interesting because it sets up a supervised learning problem in order to do unsupervised learning. While it generates fake data and tries to determine if a sample is fake or real based on trivial labels, it really does not know what the different classes in the dataset are.
- On the downside, GANs can be tough to train and suffer from convergence issues particularly because the discriminator during training does not learn as much from the true dataset as it learns to distinguish between the probability distributions.
- What these numbers suggest is that as the anomaly rate increases, there is very little change in performance in the discrimination of normal packets from anomalous packets. However, we see that many more anomalous packets are classified as normal. The results are still better on the whole than the autoencoder, however there is performance degradation.

Metrics Output Examination for Objective 3: the SVM QUBO

We have a similar performance metrics available to us for comparison. To calculate them, we must convert SVM hyperplane distances to posterior classification probabilities via Platt Scaling. These include:

1. Confusion matrix

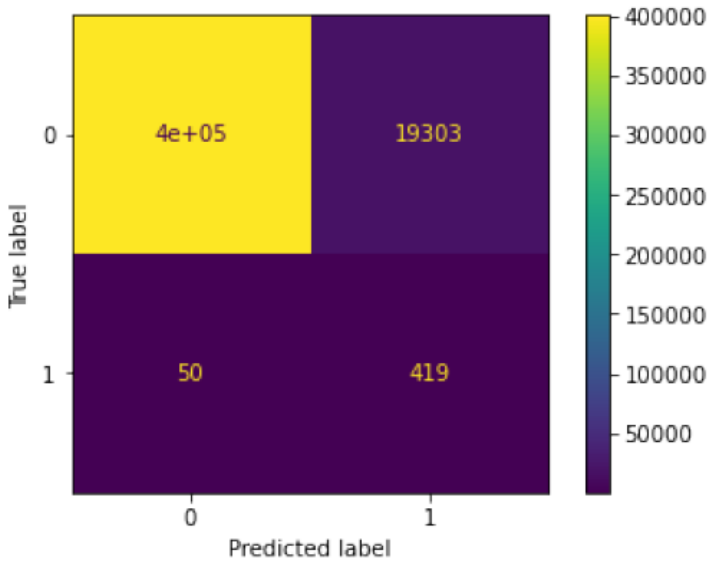


Figure 26 SVM QUBO Confusion Matrix

2. Accuracy: .954
3. Precision: .9998
4. Sensitivity or Recall: .9540
5. Matthew's Correlation coefficient: .976
6. ROC Curve and AUC: .972

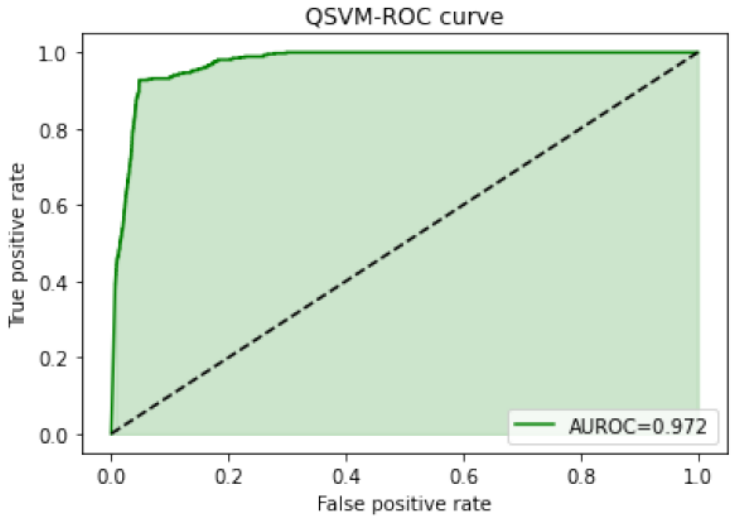


Figure 27 QUBO SVM ROC Curve

7. Wall time for processing packets in inferences per second: 1.85M

Conclusion

Key Findings & Implications

The work under the CRADA validated Entanglement's capability to solve cybersecurity anomaly detection three orders of magnitude faster than traditional methods, and with better performance as measured by Key Performance Parameters (KPP's) that covered metrics related to total inferences per second, accuracy, sensitivity, specificity, precision, F1 score, ROC and AUC values. The values being produced on the test data sets and shown in Table 4 imply a dramatic reduction in false positives-- requiring less human intervention to remove from the investigation queue.

Table 4 Cumulative Metrics for 3 Approaches

	Accuracy	Sensitivity	Specificity	Precision	F1 score	AUC	Inferences/sec
Autoencoder	0.998672	0.9996504	0.902270	0.9990095	0.999	1.0	~70M
GAN	0.9989	0.9468	0.94687	0.95034	0.94860	.97	~70M
QUBO SVM	0.95397	0.95403	0.89339	0.9998	.9800	.972	~1.85M

With additional variables or larger datasets, the Entanglement capability offers greater throughput and efficiency than traditional methods and can solve typically intractable problems at scale. Previous AAG efforts showed the ability to detect 120,000 inferences per second. This was the metric used as the benchmark and standard achievable using both the Groq tensor processing unit architecture and the Quadratic Unconstrained Binary Optimization (QUBO) model architecture. Within six months Entanglement was able to achieve an anomaly detection rate of 72,000,000 inferences per second, and it demonstrated the potential to achieve 120,000,000 inferences per second across a wide domain of data processing systems. The proprietary quantum-inspired chip solution can scale out to cards, nodes, and beyond. Additionally, the existing solution validated by the CRADA is already in development for next generation updates that will improve modularity and reduce heat signatures.

Recommendations

Moving forward, further validation on Elastic logfiles recorded in an actual SOC environment need to be processed and run through each of the three available methods and key performance parameters measured and reported. We are working toward this goal currently.

The second recommendation, which is discussed in detail in the next section, is to investigate the application of the Entanglement solution on the ability to transform generic logfiles from any vendor into true cybersecurity data pipelines with Named Entity Recognition (NER) applied prior to entering into the AI/ML stage for use cases such as anomaly detection, lateral movement detection, sensitive information discovery, or digital fingerprinting.

Limitations and Opportunities

There are two current limitations. The first, which is only a matter of time and not technology or engineering, is demonstration that these algorithms work with log files found in typical SOC's such as those from ArcSight and Elastic. The second is the generation of cybersecurity pipelines from general feeds and log file streams. We have not yet merged a generalized Natural Language Processing (NLP) technology with the model algorithms and HW advancements for the sake of building generic cybersecurity pipelines. This becomes extremely valuable in the creation of Named Entity Recognition from unstructured network activity log files. And speaking of models, we continue to use more and more complex models in production deployments. Just look at the emerging trend of viewing cybersecurity as a natural language problem. But how do you get these more complex models into a production environment and keep them updated? We believe the Morpheus SW framework addresses these challenges for both today and tomorrow in helping to ingest logfiles from multiple sources such as Arcsight, Elastic, etc.

Log Parsing is the First Step in Cybersecurity

What Are Machine Logs?

Machine logs are generated by appliances, applications, machinery, and networking equipment, (switches, routers, firewalls, etc.) Every event, along with its information, is sequentially written to a log file containing all of the logs. Although some logs are written in a structured format (e.g JSON, XML), many applications write logs in an unstructured, hard-to-digest way.

Before Mining Logs, We Must Parse Them

We can use artificial intelligence techniques on logs to detect cybersecurity threats within the network, but we must first take the raw, unstructured logs, and transform them into an easily-digestible, structured format. This transformation is called “log parsing”, where all the different entities, or “fields”, are extracted from unstructured text. This nice structured data can then be fed into downstream cybersecurity pipelines. For example, the image below in Figure 28 shows the desired input for a log parser (an unstructured log), and its output (a structured map from field name to its value):

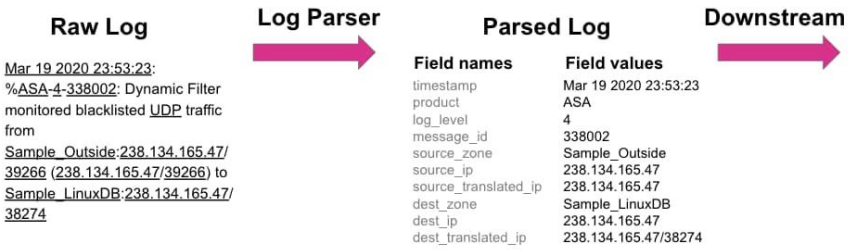


Figure 28 Log file transformation pipeline

The log parser is extracting the following fields: timestamps, dvc (device number), IP addresses, port numbers, etc.

Given the volume (petabytes per day) and value of the data within machine logs, log parsing must be scalable, accurate, and cost efficient. Historically, this has been solved using complex sets of rules, but new approaches, combined with increases in computational power, are enabling fast log parsing using neural networks, providing significant parsing advantages.

Traditionally, log parsing has been done with regular expression matching, called regexes. While this was a great initial solution, regexes come with several big challenges:

1. Writing a set of regexes in the first place is hard. They must not be overly or too loosely constrained, making it extremely difficult to write hundreds of regexes while keeping an appropriate constraining balance.

2. It is hard to manage regexes for every application and its version.

3. Regexes easily break when there is even a slight variation in the input. This can happen because of log format changes, bugs, software updates, etc.

4. It can be more computationally expensive to run hundreds of regexes over every log than running a machine learning model a single time.

Alternatively to regexes, machine learning models are **easier to train and manage** and are **more robust** to changes in the underlying logs. This can be attributed to the representation power of ML models, as well as the similarities in the logs generated by different applications. As an example, consider the similarities between Hadoop and Spark logs below in Figure 29:

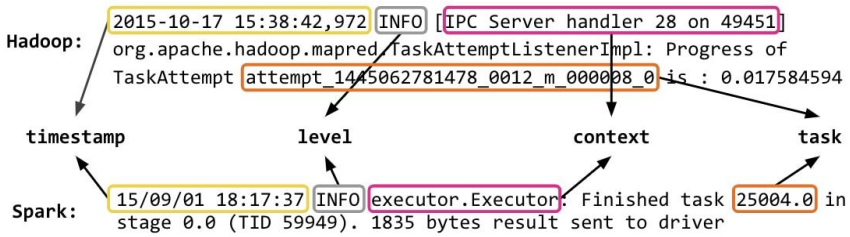


Figure 29 Hadoop and Spark logfile comparison

Although the values may vary across different log types, many fields are still shared. One

can take advantage of this similarity to build robust ML models with a deeper understanding of machine logs.

Solving Log Parsing With ML

We first wondered: how much benefit is there to using ML, and what is the best way to model this problem?

Why Template-Generation Algorithms Don't Solve Our Problem

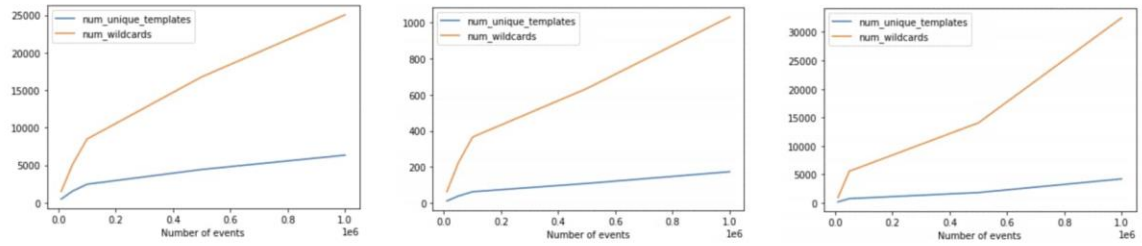
We started by experimenting with unsupervised, template-based log parsing algorithms like Drain and Spell, which automatically generate a set of "templates", or regular expressions, that can parse logs. These templates have a set of "wildcards" denoted by <*>, which represents the location of a variable token within the log.

Since these approaches are unsupervised, additional manual work is required to map from fields to one or more wildcards. For example, the first wildcard in each template above should be labeled as "timestamp," but Drain and Spell cannot produce a label for these wildcards, since the field names do not appear in the logs.

If the number of templates were small, a human could manually label each wildcard. However, because the number of templates and wildcards can grow linearly with respect to the number of logs, this approach quickly becomes infeasible.

The graphs below in Figure 30 plot the number of unique templates (in blue) and the number of wildcards (in orange) generated by Drain vs. the number of events seen. We

can see the number of wildcards increasing linearly:



(a) Android (b) BGL (c) Thunderbird

Figure 30 Wildcards and templates versus events

Beyond the fact that template-based algorithms produce too many templates and don't produce labeled fields, a deeper issue is that these approaches are not truly learning algorithms in that (1) they don't necessarily get better with more data, and (2) there is no practical way to tune them to specific datasets. In fact, as we see in the graphs above, the number of templates proliferates with more data, leading to issues with noise, rather than stabilizing as a complete representation is learned. In addition, some extracted fields and templates might be rare and require more examples, while some are common and easier to extract, but these algorithms aren't able to learn this or to treat these cases differently. While these algorithms do not solve the problem, they can be used for log clustering, which has many other valuable applications.

Shifting to Named Entity Recognition

```
    timestamp          dvc          pr… l. messa…                                pro…
Sep 23 06:43:51 20.159.82.190 %ASA-4-985623: Teardown dynamic ICMP translation
      src_zone      src_ip          s.       dst_zone       dst_ip       dst…
from DECmYViPgf:20.159.82.190/32340 to jHIbO_Vymol:196.45.223.84/7669 duration
duration
06:43:51
    timestamp          dvc          pr… l. messa…   pr…
Sep 22 23:35:31 166.147.204.39 %ASA-4-174986: TCP daemon interface inside:  ad
                address_granted    a.
dress granted vtuq.ddch.oolo (88.215.153.107)
    timestamp          dvc          pr… l. messa…   in_…          mos…
Sep 23 04:53:51 132.245.188.255 %ASA-7-570695: 4406 in use, 1565 most used
```

Figure 31 Cisco firewall logfile example

Since template-based algorithms didn't solve the problem, we shifted our focus to the natural language processing task of named entity recognition (NER). Supervised NER models do exactly what we need: they find entities ("fields", in our case) within the text. Above, we can see an example output from an NER model for Cisco ASA firewall logs in Figure 31.

It is a potentially valuable exercise to determine the feasibility of running Long Short Term Memory (LSTM) and transformer-based (BERT and GPT-3 for example) NLP on the Groq HW. If this is feasible, we now have from raw input to algorithm output an optimized processing chain with limited to no bottlenecks. Investigation into this possibility should be made a priority after completion of the algorithmic validation with Elastic log files is completed.

Additional Pathways

The type of algorithms and artificial intelligence-based processed performed under the CRADA Cybersecurity anomaly and outlier detection can be applied to many U.S. Government agencies for:

1. Generalized Artificial Intelligence through the use of Reinforcement Learning
2. Data Observability applications
3. Anomaly and Outlier detection for IT infrastructure
4. Sensitive Information Detection and Digital Fingerprinting
5. Application and user behavior deviations
6. Insider Threat Detection
7. Financial Risk Mitigation and Planning for multiple assets classes (i.e. energy, fuel, supply chain, etc)
8. Medical & Cancer research
9. 5G/RF: (a) reduction of antenna noise; (b) accelerate beam forming with added intelligence
10. Deep space operations and scheduling

11. Optimized Artificial Intelligence for UAVs
12. Large scale conditional logistics and scheduling in real-time (JADC2, Project Convergence, ABMS, Project Overmatch)
13. Data Residency
14. Community Detection (transportation/logistics, suicide prevention, find and detect adversaries)
15. Accelerated weather and climate modeling and prediction
16. Optimization of networks and network traffic

References

1. Lucas A. Ising formulations of many NP problems. Front Phys. (2014) 12:5.doi: 10.3389/fphy.2014.00005

2. Rosenberg G, Haghnegahdar P, Goddard P, Carr P, Wu K, de Prado ML. Solving the optimal trading trajectory problem using a quantum annealer. IEEE J Select Top Signal Process. (2016) 10:1053.doi: 10.1109/JSTSP.2016.2574703

3. Hernandez M, Zaribafiyan A, Aramon M, Naghibi M. A novel graph-based approach for determining molecular similarity. arXiv:1601.06693. (2016).

4. Hernandez M, Aramon M. Enhancing quantum annealing performance for the molecular similarity problem. Quantum Inform Process. (2017) 16:133. doi: 10.1007/s11128-017-1586-y

5. Perdomo-Ortiz A, Dickson N, Drew-Brook M, Rose G, Aspuru-Guzik A. Finding low-energy conformations of lattice protein models by quantum annealing. Sci Rep. (2012) 2:571. doi: 10.1038/srep00571

6. Li RY, Di Felice R, Rohs R, Lidar DA. Quantum annealing versus classical machine learning applied to a simplified computational biology problem. NPJ Quantum Inf. (2018) 4:14. doi: 10.1038/s41534-018-0060-8

7. Venturelli D, Marchand DJJ, Rojo G. Quantum annealing implementation of job-shop scheduling. arXiv:1506.08479v2. (2015).

8. Neukart F, Von Dollen D, Compostella G, Seidel C, Yarkoni S, Parney B. Traffic flow optimization using a quantum annealer. Front ICT. (2017) 4:29. doi: 10.3389/fict.2017.00029

9. Crawford D, Levit A, Ghadermarzy N, Oberoi JS, Ronagh P. Reinforcement learning using quantum Boltzmann machines. arXiv:1612.05695v2. (2016).

10. Khoshaman A, Vinci W, Denis B, Andriyash E, Amin MH. Quantum variational autoencoder. Quantum Sci Technol. (2019) 4:014001. doi: 10.1088/2058-9565/aada1f

11. Henderson M, Novak J, Cook T. Leveraging adiabatic quantum computation for election forecasting. arXiv:1802.00069. (2018).

12. Levit A, Crawford D, Ghadermarzy N, Oberoi JS, Zahedinejad E, Ronagh P. Free energy-based reinforcement learning using a quantum processor. arXiv:1706.00074. (2017).

13. Matsubara S, Tamura H, Takatsu M, Yoo D, Vatankhahghadim B, Yamasaki H, et al. Ising-model optimizer with parallel-trial bit-sieve engine. In: Complex, Intelligent, and Software Intensive Systems— Proceedings of the
11th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS-2017), Torino (2017). p. 432.

14. Tsukamoto S, Takatsu M, Matsubara S, Tamura H. An accelerator architecture for combinatorial optimization problems. FUJITSU Sci Tech J. (2017) 53:8–13.

15. Katzgraber HG, Trebst S, Huse DA, Troyer M. Feedback optimized parallel tempering Monte Carlo. J Stat Mech. (2006) P03018.
doi: 10.1088/1742-5468/2006/03/P03018

16. Wang W, Machta J, Katzgraber HG. Population annealing: theory and application in spin glasses. Phys Rev E. (2015) 92:063307.
doi: 10.1103/PhysRevE.92.063307

17. Wang W, Machta J, Katzgraber HG. Comparing Monte Carlo methods for finding ground states of Ising spin glasses: population annealing,
simulated annealing, and parallel tempering. Phys Rev E. (2015) 92:013303. doi: 10.1103/PhysRevE.92.013303

18. Karimi H, Rosenberg G, Katzgraber HG. Effective optimization using sample persistence: a case study on quantum annealers and various
Monte Carlo optimization methods. Phys Rev E. (2017) 96:043312. doi: 10.1103/PhysRevE.96.043312

19. Venturelli D, Mandrà S, Knysh S, O'Gorman B, Biswas R, Smelyanskiy V. Quantum optimization of fully connected spin glasses. Phys Rev X. (2015) 5:031040. doi: 10.1103/PhysRevX.5.031040

20. Fujitsu White Paper, Quantum Future-Quantum Present, 2019

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Mr. Josh Lenzini, Director
AAG-Research Facilitation Laboratory
Monterey, California