SQL Injection & Web Application Security: A Python-Based Network Traffic Detection Model

Nyki Anderson
Old Dominion University

SQL Injection & Web Application Security:
A Python-Based Network Traffic Detection Model[1]

Nyki L. Anderson
Mentor/Collaborator: Rui Ning
Computer Science Department, Old Dominion University
COVA CCI Cybersecurity Undergraduate Research Project
April 14th 2023

Abstract

The Internet of Things (IoT) presents a great many challenges in cybersecurity as the world grows more and more digitally dependent. Personally identifiable information (PII) (i,e,. names, addresses, emails, credit card numbers) is stored in databases across websites the world over. The greatest threat to privacy, according to the Open Worldwide Application Security Project (OWASP) is SQL injection attacks (SQLIA) [1]. In these sorts of attacks, hackers use malicious statements entered into forms, search bars, and other browser input mediums to trick the web application server into divulging database assets. A proposed defense against such exploitation is convolution neural network modeling. We have written a proof of concept, Python-based program that takes advantage of the PyTorch package's built-in convolution layered modeling classes. The model has been trained on a dataset of four known classifications and after reaching maturity underwent blind validation on a separate dataset 1000 times. The model was able to reach up to 81% accuracy by correctly reporting the packet classification. We believe the same behavior can be mapped to malicious SQLIA in other datasets by marking features in web traffic with abnormally large packet sizes, network errors, and unrecognized server responses. The research presented herein serves to corroborate related research in the field employing similar neural network and deep learning techniques to today's greatest threat to cybersecurity.

[1] For link to source code, email the author: nande010@odu.edu.

1. Introduction

The Internet of Things (IoT) has changed the way the world interacts: the way it does business, the way we exchange information, and the way private data is stored. From e-commerce giants and financial institutions to blogs and mom-pop websites; our data is everywhere. Thanks to tools like WordPress, GoDaddy, and GoogleDomains (just to name a few), anyone, at any skill level can throw up a website in a few minutes. This makes the cybersecurity landscape a minefield of unsecured web applications with few restrictions on how personal information should be handled.

Generally, any web applications that require an exchange of personal data (i.e., email address, username, password) must store these credentials in a database structure of some sort. Every internet form or query bar we enter input into communicates this information through the domain's server and then to the database. Through various techniques, properly trained programmers design rules that protect these assets from being intercepted or manipulated but the average site owner is often unaware of the protocols that they employ on their sites to protect user information.

The direct result of this information free-for-all is an inundation of hackers exploiting storage resources through a type of attack called SQL injection. In its simplest form, an SQL injection attack (SQLIA) involves typing malicious input into a form to change the way the server interprets it. This can either lead to uncredentialed access to the internal site and, if the attacker is so inclined, a complete wipe of the database. Hackers are motivated by the potential to "drop" personally identifiable information into their laps and can even leverage admin access to get deeper control of the server or network.

Machine learning and deep learning are burgeoning fields of study that have applications in medicine, finance, and social engineering. With proper parameterization and "training" these algorithms have the potential to solve problems dynamically and thus provide a unique opportunity to be applied to active network traffic analysis. This paper will focus on one such implementation of deep learning that has less than 0.12 loss and 81% accuracy in detecting malicious network behavior. We will first discuss the ways in which SQLIA has shaped web application programming practices, methods by which hackers can penetrate these defenses, explore related research, and then introduce the ways in which our work has added to this body of knowledge. The aim of this paper is both to bring accessibility to the field of machine learning as well as to inspire further exploration of these types of active SQLIA detection tactics.

2. Background

The Open Worldwide Application Security Project (OWASP) rates SQLIA as the number one threat to internet security and will be the focus of work [1]. The cybersecurity industry has long been in an arms race against hackers whose attack vectors have become exceedingly complex. As technology becomes available to the public, so do the tools for ill-intending individuals to abuse our information and privacy. Machine learning has even been employed on the "black hat"[2] side to crawl websites for vulnerabilities that can later be targets for an attack [1]. Since our focus was SQLIA, we will provide examples of common attack queries and how they can corrupt database resources or otherwise lead to uncredentialed access. But first, it is worth understanding how web forms are supposed to operate so as to realize why these particular

[2] Slang for a malicious hacker.

attacks are so pervasive and successful. Next, we will demystify a type of deep learning called neural network layering. Plus, we will quickly acknowledge the work already being done in this field to combat bad actors on the IoT.

2.1. Query Languages & Database Security

Engineering databases and their management can be a tall task, especially when put in the care of hundreds or even millions of peoples' personal information. When it comes to personally identifiable information (PII) (i.e., credit card information, medical records), there are regulations that stipulate how companies and organizations must structure their network and server protocols to best safeguard against breaches. However, these sanctions only apply to a small subset of the web applications that we access on a daily basis. The vast majority of sites that we use are not even regulated or exist in a loophole that exempts their owners from abiding by any sort of standard [1].

Database implementations can vary widely but essentially involve the same types of server-database translations of information. Relational or object-oriented platforms that employ structured query languages (SQL) are the most popular. Some of the big names are MySQL, MariaDB, and PostgreSQL. Relatively recently though, there has been a push for no-SQL platforms such as Amazon's DynamoDB and MongoDB. We will focus on the former as their penetration via SQLIA is more direct and intuitive. This is not to say that non-relational databases are more secure, they just require different tactics that are outside the scope of this paper. MySQL is the most broadly adopted SQL implementation and is the basis for most other relational query languages so we utilize its syntax for our examples.

Whenever a database element, such as a username or password, needs to be either accessed or stored, a specially formatted query needs to be crafted: denoting the location of the data allocation (in which database and in which table), the members to access (what field name or column) and usually provides rules for finding said member (or excluding other members). A simple MySQL query follows the general form seen in Figure 1.This query is raw SQL and does not apply any sort of security awareness[3]. All database accesses are structured in this way at some point in the backend of a site. There are certain best practices that can limit a site's SQL attack surface but nonetheless, 70% of web applications are unsecure [1]. With this background, it is now time to see how SQL can be manipulated when programmers (and amateur website owners) do use responsible database management.

Figure 1. A Simple MySQL Query

```
SELECT * FROM login_details WHERE username = "john_doe" AND password = "password"
```

This query is requesting all data fields in the login_details table where the username is $USERNAME and the password is $PASSWORD. The "$" indicates that the input comes from a variable, most likely user input from a form field.

[3] There are two main protocols that are supposedly more secure ways of communicating with a database, namely, prepared statements and MySQLi but these are beyond the scope of this paper. They basically obfuscate the member names that are being queried until right before they are actually being accessed or stored. These protocols are now the industry standard but can still be vulnerable to certain attack queries if other safeguards are not put in place.

2.2. SQL and GET Injections

What makes SQL so vulnerable is simply the accessibility of the language. Hackers can learn MySQL just as any other person and with that knowledge comes the ability to construct their own queries that violate the intention of the database manager. Without any form of security, they can directly emulate the SQL code and inject dangerous commands that are indiscernible from that used by the programmer. The simplest way for a hacker to surreptitiously introduce their own query when submitting a form is shown in Figure 2. Note, that this is a simple form-type injection, but SQLIAs are not restricted to form fields[4]. Form fields are generally handled via "POST" requests via the server which are private by default. Meaning they are not visible to the user and are transferred via strict HTTPS protocols. It is more difficult to intercept POST requests but the example in Figure 2 can still be successful if more is not done to restrict user input.

Figure 2. Simple Example of Comment Injection[5] [1]

```
[1] SELECT * FROM users WHERE username = $USERNAME AND password = $PASSWORD;
[2] 1 or 1=1; #
[3] SELECT * FROM users WHERE username = 1 or 1=1; # AND password = $PASSWORD;
```

Line 1; The original database query server-side requesting all fields for a certain username and password combination.
Line 2: The attackers input into the username field of a form.
Line 3: This is what the query looks like after the attacker submits the form. The "#" character is the comment operator in MySQL, therefore everything after it is not read by the language interpreter. The attack works because then the interpreter evaluates "id = 1 or 1=1" and since 1=1, the query is true so all of the fields in the table users are now returned to the intruder.

To prevent the direct injection of unintended queries, programmers design complex "rules" for each element to restrict the kinds of input a user can enter. Commonly, the rules exclude special symbols and non-standard encodings because such symbols can be reserved words or operators in MySQL (like the "#" character). This is a form of "escaping" or input validation, where user input and dynamic or variable input is parsed for non-standard input. It is worth noting that even a highly skilled programmer using these best practices are not capable of completely securing their database traffic. This is due in part to the innumerable methods and combinations of input that an attacker could use and the limited ability to infer every potential one.

In fact, many injections take advantage of a server request called a "GET" request which accepts input (usually) from the URL box. GET requests are difficult to prevent because often the purpose of a URL is to map the site pages to unique addresses and HTTP/HTTPS protocol

[4] The focus of our code, in fact, is a type of injection that occurs in the URL box, whereby an intruder intercepts information on the way to the server via a GET request.
[5] The Open Worldwide Application Security Project (OWASP) is the authority on web security prevention and awareness. All examples of SQLIA utilized in this paper have been referenced from their cite.

includes certain special characters (including “%” and “#”) to denote the different address scopes. The next SQLIA example would be successful if the programmer was expecting a URL mapping and did not compare it against a list of acceptable mappings.

Figure 3. GET Request Injection [1]

```
[1] http://testsite.com/index.php?page=contact.php
[2] http://testsite.com/?page=http://evilsite.com/evilcode.php
```

Line 1: This is the intended response for a given action on the site. This programmer is expecting “contact.php” to be sent to the PHP function include() which accepts a PHP file and enables the server to run the code in that file. Line 2: The attacker has instead inserted the URL to a dangerous file that could do anything. For example, phpinfo() is used to get information about the server configuration which could lead to a more direct attack on the server itself.

Though the example in Figure 3 did not involve an SQLIA, many GET requests are used to serve assets to or from the database. These requests look essentially the same because, in a URL, the identifier before the “=” operator is the variable that will hold the desired information that follows it. A common example of using GET to access the database is when a user clicks to view their profile on the site. You may notice when you do this that your username or other identifier is appended to the end of the URL for mapping purposes. In this way, programmers allow a sometimes inappropriate amount of transparency to a perceptive hacker about server-side processes. The theme of SQLIA is one of programmers that are oblivious to the real danger they are imparting on, not only, the security of the site but also their responsibility to protect your data.

2.3. SQLIA Detection via Web Traffic Monitoring

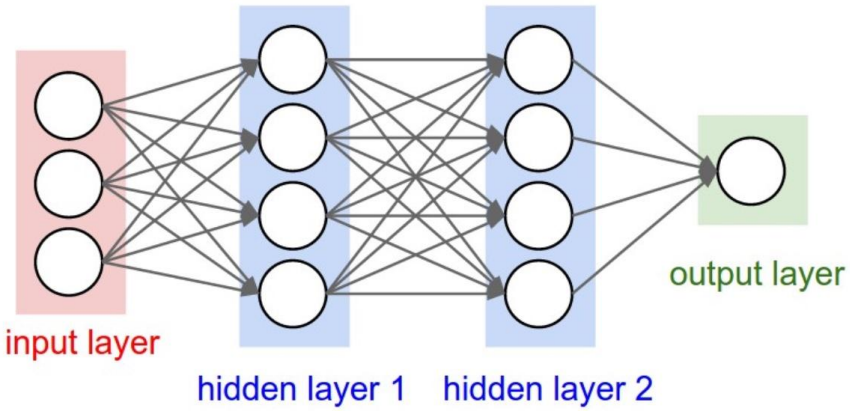
SQLIA detection and prevention technology is an underdeveloped area of cybersecurity. Current methods include bag-of-words comparisons and web application firewalls (WAF). Both of these techniques rely on databases of known suspicious SQL strings and have to be constantly updated to include new queries. Using these methods requires frequent, active monitoring and review of error logs and web traffic which requires many man hours dedicated to sorting out of place server responses from benign communication. If these routines were to be automated, it would require the ability to filter client-side server responses and detect unusual activity based on a known baseline. This is a popular method because it gauges improper client-side activity by what is expected from each site individually. However, it requires tons of network traffic for the baseline and is not particularly maintainable as new SQLIAs are discovered. Research is being conducted that blends these techniques with sophisticated machine learning algorithms in an attempt to actively parse web traffic for risky responses with limited human intervention. We will discuss some of these papers as they align with our objective to treat SQLIA as a network traffic analysis problem in section 4. Related Research.

3. Neural Networks and Model Design Principles

Machine learning and neural networking may seem like advanced topics but they are actually quite accessible. They are merely mathematical techniques used to solve problems which involve multiple parameters or inputs that must converge to describe the behavior of a dataset, and do so independently. These algorithms are said to "learn" through a process known as training which confers most of the advanced mathematics involved on a well-understood dataset that mimics the behavior of the dataset wished to be understood. Predictions are made and evaluated, parameters are adjusted (hopefully in a way that improves the next prediction), and the model is fed more of the dataset. This continues until error has been reduced to an acceptable value. We designed a neural network or deep learning algorithm that uses a decentralized structure of nodes which do not make decisions by themselves, but through their convolution. In this way, the problem is split into separate layers that receive inputs and produce outputs that may be fed into other layers or into the eventual prediction itself. A diagram describing our model's neural network layering can be found in Figure 6.

Neural networks have three kinds of layers, the visible or input layers, hidden layers, and the output layer (see Figure 4). The interplay between these layers gives the neural network its name as it emulates the human brain. All input has noise and unpatterned information that needs to be filtered out such that a model can properly identify predictable features. Convolution layers are the layers that filter subsets of complex and abstract features into a more useful output while maintaining all the necessary information. The purpose of the convolution layer is to apply filters that define patterns for the model to detect. By contrast, a fully-connected layer's purpose is to use the convolution layer's features to actually make predictions through backpropagation. This means that after a prediction is made, the model uses the previous parameters and their respective weights and biases to update the new parameters, then passes them to the model (called the forward pass) to be used in the next prediction. The assignment of weights and biases is a highly tenuous process but is essential for the model to effectively learn from each iteration.

Figure 4: The Three Types of Layers in a Neural Network [7]



Updating models based on previous outputs, weights and biases, multiple input parameters, and the interplay between the hidden layers is difficult to maintain manually. The computations are simple but the sheer number of variables makes it impossible to solve without some routine keeping track of it all.

We chose to use gradient descent which is used in mathematics to regressively optimize parameters until it reaches a local minimum. This process is broken up into epochs, delimited from the forward pass to the next forward pass. Parameterization is handled within the gradient algorithm itself but there are values called hyperparameters that actually help the algorithm make the "decisions" for how much to update the parameters each epoch. These variables include the number of datapoints to include in each batch, the number of batches per epoch, the number of epochs, the drop rate, and the step size or learning rate and are manipulated by the programmer until an acceptable training accuracy has been reached. We will discuss our hyperparameters and their fine-tuning in section 5.1. Hyperparameters & Layer Features.

4. Related Research

As pervasive as SQLIA is and how simple it is for any reasonably knowledgeable hacker to find the cracks in a programmer's logic, a fair amount of research has already gone into using machine learning techniques to monitor network traffic. Debashish Das et. al. attempted "to classify the SQL Injection attacks based on the vulnerabilities in web applications." They achieved a high success rate using dynamic query matching to a database of valid or trusted strings and syntactic rules [2]. The dataset used in their research required a considerable amount of information about the sites they analyzed to generate effective query-lists and based the success of their algorithms on how close they matched the actual classification. Umar Farooq went a more statistical route. Assigning tokens to their data and utilizing various gradient descent algorithms to characterize their training [3]. This approach achieved a high accuracy (above 99%) for detection and a lower degree of error but utilized several parallel learning approaches to hone in on the most effective. Volkova et. al. used natural language processing and developed an algorithm to break down URLs to determine suspicious network activity in each segment. They too explored different machine learning techniques, achieving 99% successful detections from bag-of-words methods and only 55% from pattern matching [4]. It is not clear from the body of research whether a strictly stochastic training technique or pre-processed pattern and character matching is a more effective approach to model design.

5. Methods & Tools

Python is a powerful programming language with tons of support for deep learning model training and testing including an extensive package of classes called PyTorch [6]. We used mini-batch gradient descent to train our network traffic detection model. Model training, and gradient descent especially, can really tax a CPU which is used to performing general computations. Because each epoch involves updating multiple parameters, computing a prediction, and then comparing that prediction to the expected value, the computations can quickly add up. Luckily, the repetitive and independent nature of each node (layer) allowed us to implement a simple design tweak with major performance implications; CUDA. CUDA is NVIDIA's parallel computing API that makes use of the amplified processing power inherent in their graphics cards to split computation handling between cores. PyTorch has built-in CUDA support[6], and thanks to Google Colab resources cloud computing resources, we were able to take advantage of its unmatched computing speeds on a relatively large recursion.

[6] Since our model implements CUDA and we understand that not all users will be able to run the code with a GPU, we have designed our program to detect what device is available (CPU or GPU) and then send the data to that device where necessary..

We adopted our dataset from a GitHub repository which consisted of a flow sequence network (FS-Net) used for encrypted traffic classification [5]. The traffic is classified into four types which we trained our model to predict. Given that our dataset does not necessarily include malicious traffic as one of these classifications, our work is a proof of concept but could be tested on such a dataset in the future. The theory is that if our model can correctly differentiate between different types of web traffic, it could detect a classification of malicious packets sent to the server via SQLIA. For now, we will focus on how we trained our model and discuss SQLIA traffic detection in our section 6. Implications for SQLIA Detection.

5.1. Hyperparameters & Layer Features

Hyperparameters, as previously mentioned, are those that determine how often and to what degree changes should be made in the weighted parameters in order to learn most effectively. Our algorithm makes use of three such hyperparameters (plus one which was assisted by the dataset). Namely they are learning rate, the number of epochs, and the dropout rate[7]. The dataset driven hyperparameters was batch size or the number of packets per batch. Aiming for mini-batch gradient descent, we needed one batch per epoch. This turned out to be a 260 packet batch size. The rest of our hyperparameters were adjusted through trial and error until overfitting was no longer an issue. The model had a tendency to be overtrained with relatively small steps from any of the hyperparameters. The final values for each of our hyperparameters can be reviewed in Figure 5 and will be discussed in more detail as they pertain to the characterization of our model training. In future implementations, we wish to explore automated adjustment of learning rate and dropout rate to make the process more efficient and produce optimized results. The current values sufficed for our purposes though and have produced an acceptable model.

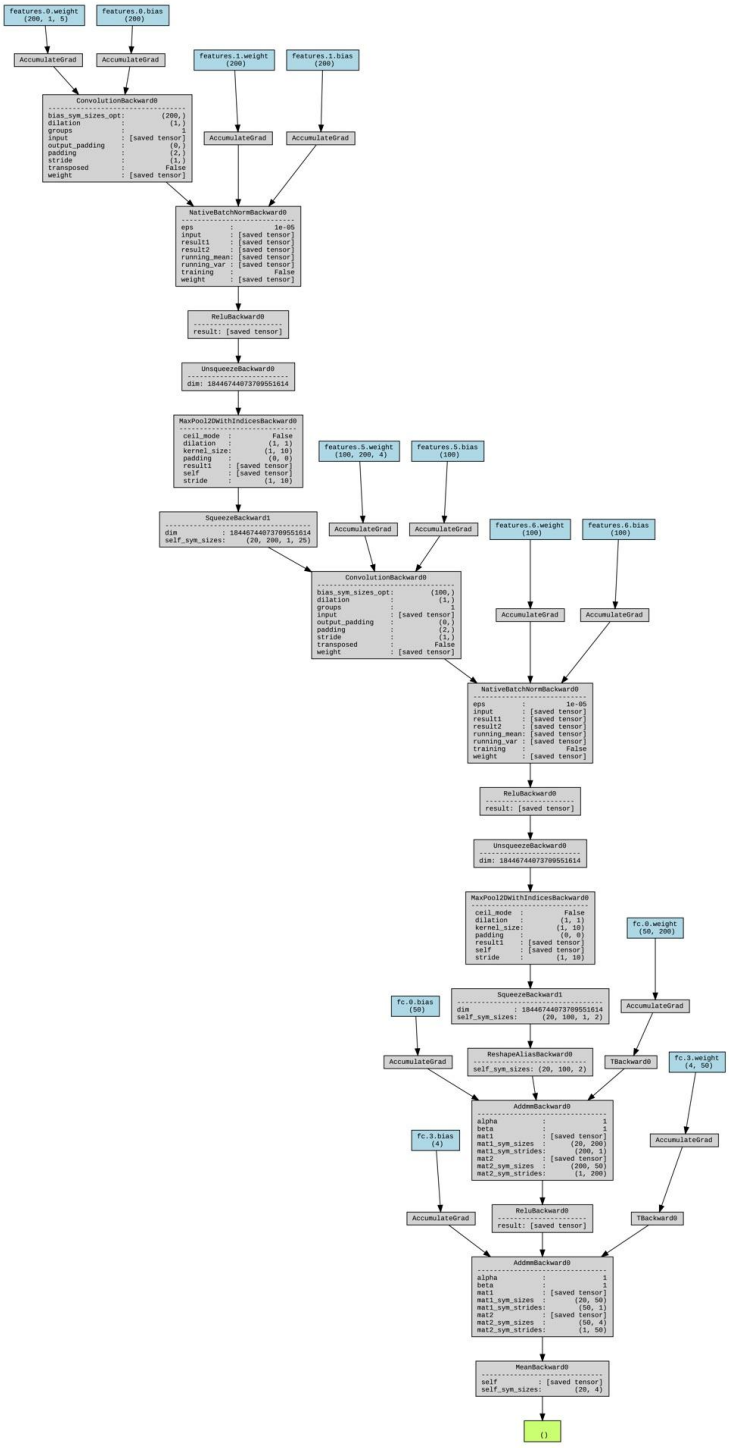
Figure 5: Hyperparameters for Model Training

```
[1] batch_size = 260
[2] learning_rate = 0.001
[3] num_epochs = 80
[4] drop_rate = 0.001
```

The model consists of a convolution layer wrapped inside a sequential class that is forwarded to a fully-connected layer. The sequential class allows us to avoid forwarding each layer to the model one at a time and treats all the convolution processes as a single layer that are applied to each prediction. We built our loss function using PyTorch's built in CrossEntropyLoss[8] function and our optimizer of choice was the Adam[9] stochastic optimization method. A detailed description of our model's convolution network can be seen in Figure 6.

[7] Dropout rate is a method of model regularization that randomly "drops" or ignores nodes during training. It is a computationally cheap technique that prevents overfitting to any one dataset.
[8] The CrossEntropyLoss function helps determine a score between 0 and 1 that summarizes the average difference between the predicted and actual values.
[9] For more information on the Adam optimizer function, read its publication [8].

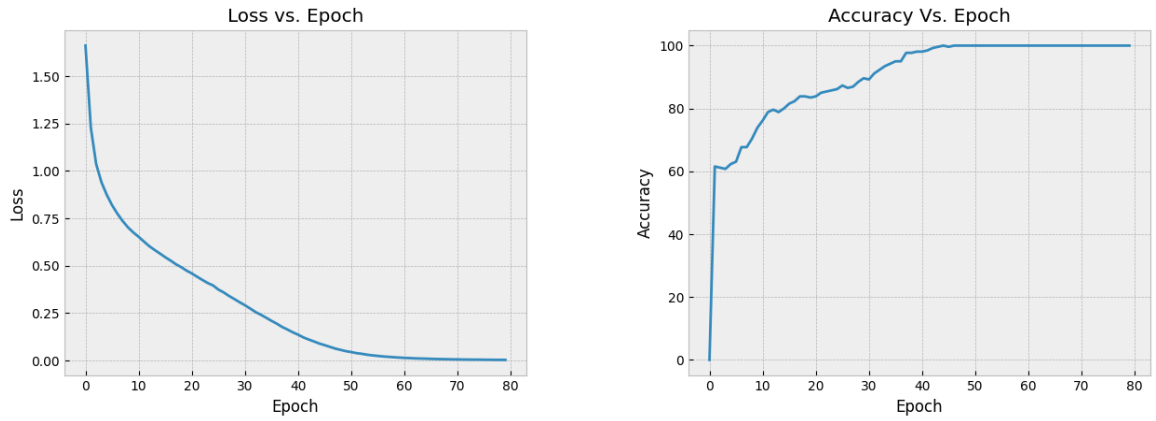
Figure 6: Visualization of Neural Network for Model with Layer Attributes



5.2 Model Training Results

After optimizing the model and settling on the most precise step values for our hyperparameters, we ran the model through the training algorithm for a final time producing the figures that populate this paper. We will first examine Figures 7a and 7b which correspond to the loss function and accuracy of the model, respectively. What we expect from an efficient training algorithm are loss and accuracy curves that are inverses of each other. The loss curve should decrease exponentially from a number no greater than 2 (1+ bias) and an accuracy curve should increase exponentially toward 100%. The smoother the respective descents and ascents, the more efficiently the model learned with each epoch. As seen, our figures, there is little up and down movement between each epoch which is a good indicator that the model has a good learning rate. This is because a stepping curve indicates that the learning rate was too large and kept bouncing round the actual values. So we are seeing from the data that our training was successful.

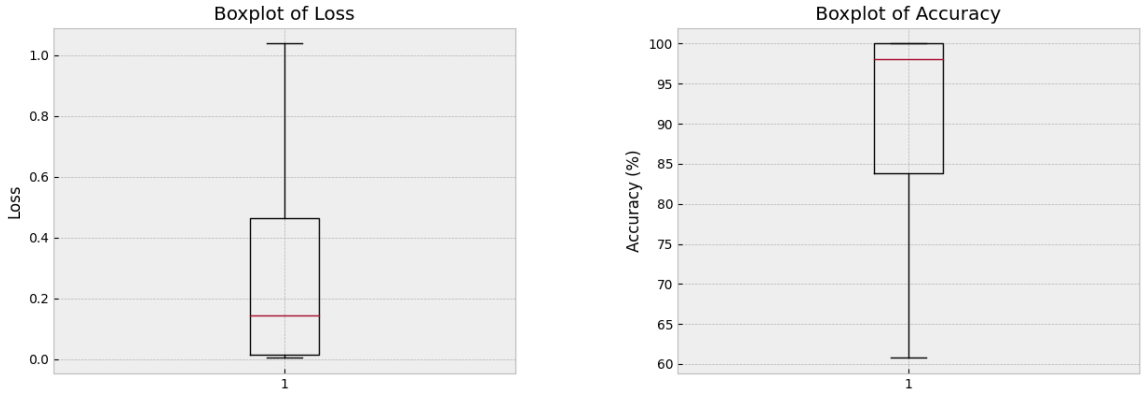
Figure 7: Loss & Accuracy of Predictions Over Time



a. b.

To further characterize the dataset we have included box plots (Figures 8a and 8b) to develop an idea of how much time the model spent making correct predictions. We wanted our training data to spend a good amount of time making predictions 95% and higher. We did this hy increasing the number of epochs gradually until we started losing integrity. This occurred very quickly between 50 and 100 epochs. We had high confidence between our training data average and the validation (or testing) average for these values so we adjusted dropout rate and learning rate alone from this point. Unsurprisingly, this sort of correlation chasing was how much of our hyperparameter tuning was accomplished.

Figure 8: Boxplots for Accuracy and Loss

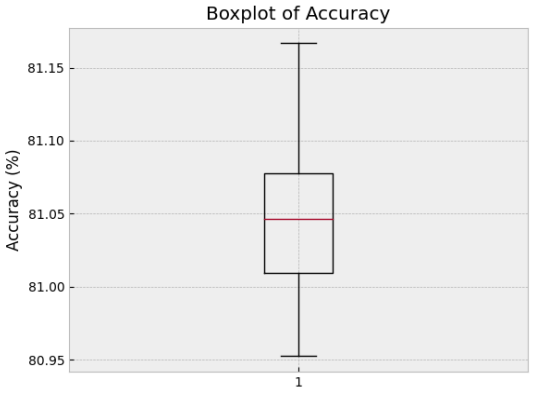


a. b.

5.3. Model Validation

We validated our model through 1000 blind dataset tests (the same dataset shuffled each time). Each test consisted of 30 packets as dictated by the size of the test dataset. We calculated the percentage of correct packet classification predictions for all 1000 tests. And this value was used as our success rate. Because there is no gradient calculated during validation testing, we did not require a loss function or an error calculation (as this could be inferred by the correct prediction accuracy). Therefore, the boxplot for all 1000 tests was the only figure produced to represent the validation phase of our model (Figure 9).

Figure 9: Boxplot for Validation Testing



What's promising about this boxplot is that its y-axis range is quite tight, with the majority of the data +/- 0.05% of the average. This describes both a precise and accurate model, as it consistently predicts the correct packet classification the same percent of the time. It is worth noting that the trained model was quite touchy, especially with the learning rate and dropout rate. There seemed to be a more or less clear direction for the number of epochs but once we leveled this hyperparameter, the others became fixed in their current positions. Small deviations in either direction dropped the accuracy by as much as 30%. This leads us to believe there may be issues in the model features themselves, the weights and biases, or the layering scheme we chose. Without a clear direction to take on these fronts, we decided to leave it as is. A consistent 81% after 1000 validations is still a statistically impressive figure.

6. Implications for SQLIA Detection

Let's not forget the motivation behind this algorithm, to detect malicious network traffic that corresponds to SQLIAs. As a proof of concept, this project works well as we can project that detecting and classifying specific signatures for malicious injections would come in the form of abnormally large packet transfers, unexpected redirections, and network errors. If we could analyze and parse our own dataset of network traffic for several different web applications, we could use our current model to train such a dataset. This unfortunately was beyond the scope of our abilities at the time but is well within them at the conclusion of this project.

7. Conclusion

Prior to this research program, I had zero knowledge about convolution neural networking, gradient descent, or any of the topics explored in this paper. Given more time, our research would have likely improved beyond its current state of 81% accuracy. It was clear that some of our convolution features must have been too strict as hyperparameter tuning was very touchy. But beyond that, we were able to develop a succinct model and program to classify web traffic. Now we look to the future for related work, its implications for improved SQLIA detection, and a securely connected IoT.

References

[1]OWASP Foundation | Open Source Foundation for Application Security. owasp.org. https://owasp.org

[2]Das D, Sharma U, Bhattacharyya DK. An Approach to Detection of SQL Injection Attack Based on Dynamic Query Matching. International Journal of Computer Applications. 2010;1(25):28–34.

[3]Farooq U. Ensemble Machine Learning Approaches for Detection of SQL Injection Attack. Tehnički glasnik. 2021;15(1):112–120. doi:https://doi.org/10.31803/tg-20210205101347

[4]Volkova M, Chmelar P, Sobotka L. Machine Learning Blunts the Needle of Advanced SQL Injections. MENDEL. 2019;25(1):23–30. doi:https://doi.org/10.13164/mendel.2019.1.023

[5]Akrusher. Fs-net. GitHub. 2023 Mar 30. https://github.com/Akrusher/Fs-net.git

[6]PyTorch. www.pytorch.org. https://pytorch.org

[7]CS231n Convolutional Neural Networks for Visual Recognition. cs231n.github.io. https://cs231n.github.io/neural-networks-1/

[8]Kingma DP, Ba J. Adam: A Method for Stochastic Optimization. arXiv.org. 2014 Dec 22. https://arxiv.org/abs/1412.6980