

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Sétif 1 – Ferhat ABBAS
Faculty of Sciences
Department of Computer Science



MASTER'S THESIS

Presented by :
ZITOUNI Ahmed Faouzi
SEDJAL Moheamed Aymen Dhaya Eddin

Field of Study : Computer Science
Specialty : Data Engineering and Web Technologies

Title :
Detecting SQL Injections using Deep Learning

In front of the jury

DR. BENZINE Mehdi	University of Sétif 1 – Ferhat ABBAS	Supervisor
DR. LAKHFIF Abdelaziz	University of Sétif 1 – Ferhat ABBAS	Chairperson
DR. MEDIANI Chahrazed	University of Sétif 1 – Ferhat ABBAS	Examiner

Academic Year: 2024–2025

Dedication

To our parents

Thank you for your continuous support and guidance throughout our studies.

To our brothers and sisters

Thank you for your encouragement and presence during this academic journey.

To our friends

Thank you for your help and for contributing to a positive working environment.

Abstract

SQL injection attacks remain one of the biggest threats to web applications, because they allow the attacker to gain trusted access to data without authorization, which can lead to irreparable damages. As part of this project, we examined how deep learning and machine learning can aid in detecting these attacks automatically. In total, we built and evaluated six models: Logistic Regression, Support Vector Machine (SVM), Multilayer Perceptron (MLP), Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and BERT (Bidirectional Encoder Representations from Transformers). Overall, BERT achieved the highest scores in accuracy, precision, recall, and F1-score. This demonstrates that transformer-based models such as BERT have a better understanding of SQL query structures, which makes them efficient in detecting complex attacks. This study shows how deep learning, especially BERT, can improve web application security.

Keywords: SQL Injection, Web Application Security, Deep Learning, Machine Learning, BERT, Transformer Models, Model Comparison, Threat Detection

Résumé

Les attaques par injection SQL restent l'une des plus grandes menaces pour les applications web, car elles permettent à un attaquant d'accéder de manière non autorisée aux données, ce qui peut entraîner des dommages irréparables. Dans le cadre de ce projet, nous avons étudié comment le deep learning et le machine learning peuvent aider à détecter automatiquement ces attaques. Au total, nous avons construit et évalué six modèles : la régression logistique, la machine à vecteurs de support (SVM), le perceptron multicouche (MLP), le réseau de neurones récurrent (RNN), mémoire longue à court terme (LSTM) et BERT (Bidirectional Encoder Representations from Transformers). Dans l'ensemble, BERT a obtenu les meilleurs résultats en termes de précision, rappel, exactitude et score F1. Cela démontre que les modèles basés sur les transformers, comme BERT, ont une meilleure compréhension de la structure des requêtes SQL, ce qui les rend efficaces pour détecter des attaques complexes. Cette étude montre comment le deep learning, en particulier BERT, peut améliorer la sécurité des applications web.

Mots-clés : Injection SQL, Sécurité des applications web, Apprentissage profond, Apprentissage automatique, BERT, Modèles Transformer, Comparaison de modèles, Détection de menaces

ملخص

تظل هجمات حقن SQL من أخطر التهديدات التي تواجه تطبيقات الويب، حيث تتيح للمهاجم الوصول غير المصرح به إلى البيانات، مما قد يؤدي إلى أضرار جسيمة لا يمكن إصلاحها. في إطار هذا المشروع، قمنا بدراسة كيفية استخدام التعلم العميق والتعلم الآلي للمساعدة في الكشف التلقائي عن هذه الهجمات. طورنا وقمنا بتقييم ستة نماذج: الانحدار اللوجستي، آلة الدعم الناقل (SVM)، الشبكة العصبية متعددة الطبقات (MLP)، الشبكة العصبية التكرارية (RNN)، الذاكرة طويلة المدى (LSTM)، ونموذج BERT (تمثيلات المشفر ثنائية الاتجاه من المحولات). حقق نموذج BERT أفضل النتائج من حيث الدقة، الاستدعاء، الدقة النوعية، ومقياس F1. هذا يوضح أن النماذج المعتمدة على المحولات مثل BERT تمتلك فهماً أفضل لهياكل استعلامات SQL، مما يجعلها فعالة في اكتشاف الهجمات المعقدة. تبرز هذه الدراسة كيف يمكن للتعلم العميق، وخاصة BERT، تحسين أمان تطبيقات الويب.

الكلمات المفتاحية: هجمات حقن SQL، أمان تطبيقات الويب، التعلم العميق، التعلم الآلي، BERT، نماذج المحولات، مقارنة النماذج، اكتشاف التهديدات

General Introduction	1
Chapter 1	2
SQL Injections.....	2
1.1 Introduction	2
1.2 SQL injection.....	2
1.2.1 Definition	2
1.2.2 How SQL Injection Works	3
1.3 Techniques of SQL Injection.....	4
1.3.1 Error-Based SQL Injection	4
1.3.2 Blind SQL Injection	5
1.3.2.1 Content-Based Blind SQL Injection.....	6
1.3.2.2 Time-Based Blind SQL Injection	7
1.3.3 Tautology-Based SQL Injection.....	8
1.3.4 Union-Based SQL Injection.....	9
1.4 Methods to prevent SQL Injection attacks	13
1.4.1 Prepared Statements (with Parameterized Queries)	13
1.4.2 Stored Procedures.....	13
1.4.3 Input validation	14
1.4.4 Escaping All User-Supplied Input	15
1.5 Conclusion.....	15
Chapter 2	16
Deep Learning	16
2.1 Introduction	16
2.2 Machine learning	16
2.2.1 Machine Learning Types	16
2.2.1.1 Supervised Learning (SL):.....	16
2.2.1.2 Unsupervised Learning:.....	18
2.2.1.3 Reinforcement Learning:	18
2.2.2 Machine learning algorithms.....	18
2.2.2.1 Logistic regression	18

2.2.2.2 Support Vector Machine (SVM)	19
2.2.3 Real-world machine learning use cases	20
2.3 Deep Learning	21
2.3.1 Artificial Neural Networks (ANNs)	22
2.3.4 Deep Learning Training Cycle.....	24
2.3.2 Activation functions	24
2.3.2.1 Non-Linear Activation Functions	25
2.3.3 Deep learning architectures.....	27
2.3.3.1 Recurrent Neural Networks	27
2.3.3.2 Long Short-Term Memory Networks	30
2.3.3.4 Transformers	32
2.3.3.5 BERT	37
2.4 Related works.....	39
2.4.1 Introduction	39
2.4.2 Detection Approaches Using Machine Learning.....	39
2.4.3 Deep Learning for SQL Injection Detection.....	40
2.5 Conclusion.....	41
Chapter 3	42
Conception and Implementation	42
3.1 Introduction	42
3.2. Dataset	42
3.3 Development Environment Overview	43
3.3.1 Programming language.....	43
3.3.2 Libraries Used	44
3.3.3 Development setup.....	44
3.3.3.1 Visual Studio Code	44
3.3.3.2 Jupyter Notebook	45
3.3.3.3 Google Colab	45
3.4 Models Implemented	45
3.4.1 Support Vector Machine (SVM).....	46
3.4.2 Logistic Regression (LR)	46

3.4.3 Multilayer Perceptron (MLP).....	46
3.4.4 Recurrent Neural Network (RNN)	47
3.4.5 Long Short-Term Memory (LSTM)	48
3.4.6 BERT.....	49
3.4.6.1 Why BERT for SQL Injection Detection.....	49
3.4.6.2 BERT Code and Implementation	50
3.5 Comparative Summary of Model Architectures	52
3.6 Conclusion.....	54
Chapter 4	55
Performance Evaluation and Results.....	55
4.1 Introduction	55
4.2 Performance Evaluation Metrics	55
4.3 Evaluation on Test Set (20%)	57
4.3.1 Results Presentation	57
4.3.2 Visual Performance Analysis.....	60
4.3.3 Performance Analysis of BERT	61
4.4.3.1 Training Loss Curve	61
4.4.3.2 Model Evaluation on New Unseen Data	61
4.4 Conclusion.....	64
General Conclusion.....	65
References.....	66

List of figures

Figure 1. 1 SQL Injection attack	2
Figure 2.1 Linear Regression Example – House Price Prediction	17
Figure 2.2 Classification Example – Spam vs Not Spam Emails	17
Figure 2.3 Example Curve: How Study Hours Affect Exam Success Rate	19
Figure 2.4 SVM: Optimal Hyperplane and Support Vectors	20
Figure 2.5 Neuron Computation in an Artificial Neural Network.....	22
Figure 2.6 The Mathematic of Neural Networks	22
Figure 2.7 Basic Structure of an Artificial Neural Network	23
Figure 2.8 Sigmoid Activation Function.....	25
Figure 2.9 Tanh (Hyperbolic Tangent) Function	26
Figure 2.10 ReLU (Rectified Linear Unit) Function	26
Figure 2.11 Recurrent Neural Network (RNN) Architecture	28
Figure 2.12 Types of RNN's.....	29
Figure 2.13 LSTM Gate Mechanisms	31
Figure 2.14 Mathematical Formulation of LSTM Gates and States.....	31
Figure 2.15 LSTM Layer Architecture and Operations	32
Figure 2.16 Transformer Model Architecture with Multi-Head Attention	33
Figure 2.17 Scaled Dot-Product Attention Mechanisms	34
Figure 2.18 Self-Attention Equation.....	34
Figure 2.19 Scaled Dot-Product Attention Mechanisms	35
Figure 2.20 Multi-Head Attention Equation	35
Figure 2.21 Multi-Head Attention Mechanisms.....	36
Figure 2.22 Feedforward Network Equation	36
Figure 2.23 Main BERT Models	37
Figure 3.1 Label Distribution of The Dataset	43
Figure 4.1 Confusion Marix	55
Figure 4.2 F1-Score Across Cross-Validation Folds for the SVM Model	60
Figure 4.3 LSTM Training and Validation Loss over Epochs	60
Figure 4.4 Training Loss Curve for the Fine-Tuned BERT Model	61
Figure 4.5 Confusion Matrix on Unseen Data	62

List of tables

Table 1.1 Results of a SELECT query without UNION	11
Table 1.2 Result of user query after a Union based SQL injection	12
Table 3.1 Overview of HyperParameters and Architectures Used in Model Implementation ..	52
Table 3.2 Optimization Techniques, Regularization Methods, and Training Epochs Used per Model.....	53
Table 4.1 Performance Comparison of Models on SQL Injection Dataset	57
Table 4.2 Proposed Models vs. Existing Work.....	59
Table 4.3 Performance Comparison of Models on Unseen SQL Injection Dataset	63

General Introduction

Web applications became a critical part of our routine, supporting everything from e-commerce via social networks and banking to even healthcare services. Their spread has brought about a revolutionary change in the way we relate to each other, communicate, and do business. At the same time, this growing dependency on web platforms also makes them very attractive targets for cyberattacks. Among these threats, SQL injection remains the highest and most dangerous vulnerability, posing a few severe dangers to data security and user privacy. To counter SQL injection attacks, input validation and parameterization of queries have been used as base security measures, in addition to the extra firewalls for protection. While some of these older techniques can protect against risks, many examples have failed to detect many cases when faced with smarter and constantly evolving attacks. Given the very dynamic trends in the cyber world, there lies a need for more advanced and flexible measures for detecting real-time malicious SQL queries.

Modern breakthroughs in AI and deep learning have created new room for cybersecurity with intelligent models capable of learning and adapting toward highly complex patterns of attack behavior. Among them, transformer architectures, especially BERT (Bidirectional Encoder Representations from Transformers), can truly be said to stand out in their ability to understand textual patterns and contextual relationships. Although intended for NLP tasks, BERT has potential applications in cybersecurity, including cyberattack and malicious query classification.

This research aims to apply deep learning techniques to establish an intelligent detection system for SQL injection. The focus of training on real-world datasets is to create a highly robust detection framework that distinguishes between legitimate SQL queries and malicious ones with great precision. The principal advantage of our method is its ability to dynamically adapt to new attack patterns, thus adding enhanced security to web applications, unlike the conventional rule-based approach.

Chapter 1 introduces SQL injection attacks, covering definitions, types, real-world examples, and traditional detection methods.

Chapter 2 explains key concepts in Machine Learning and Deep Learning and presents the neural network architectures used in our experiments.

Chapter 3 describes the dataset, preprocessing steps, and technical details, including model architectures and hyperparameters.

Chapter 4 presents and analyzes the results using various evaluation metrics and discusses model performance.

Through this study, we aim to demonstrate the effectiveness of deep learning techniques in cybersecurity, particularly for the early detection of SQL injection attacks.

Chapter 1

SQL Injections

1.1 Introduction

With increasingly digital living, web applications are at the core of day-to-day life from managing finances and online purchasing to collaborating and communicating. This ease of the virtual world comes with inherent security challenges. Cyber attackers persistently evolve their methods to exploit weaknesses, thereby endangering unauthorized data access, downtime of services, and irreparable damage to reputation.

1.2 SQL injection

1.2.1 Definition

An SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands[1].

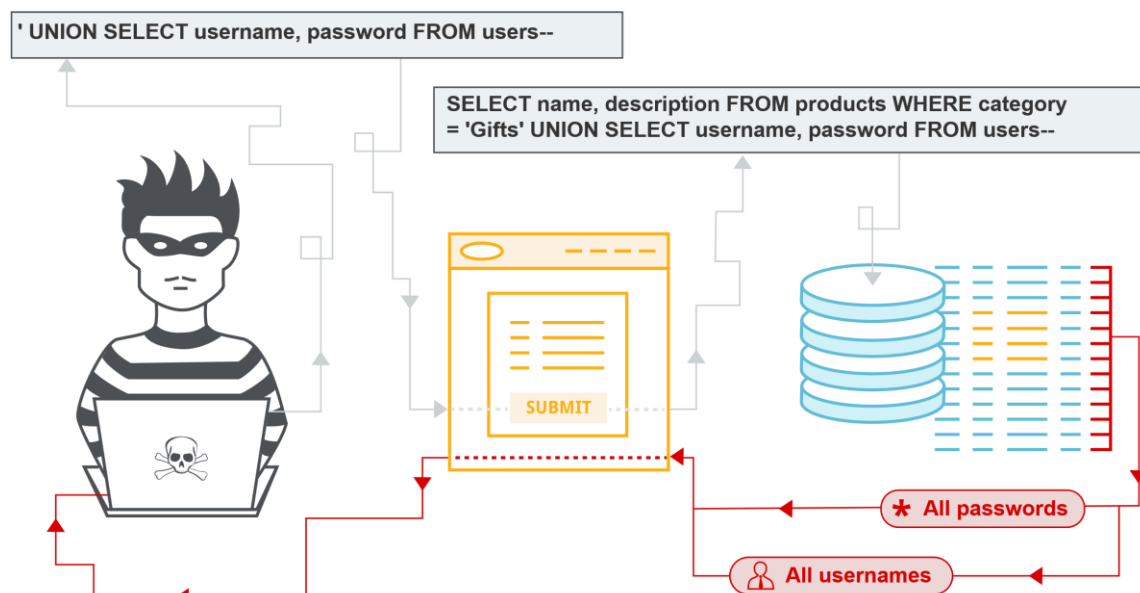


Figure 1. 1 SQL Injection attack

1.2.2 How SQL Injection Works

It typically involves the following steps:

1. **Identification of vulnerable inputs:** Attackers first identify inputs within the web application that are vulnerable to SQL injections. These inputs such as text fields in a form, URL parameters, or any other input mechanisms.
2. **Crafting the malicious SQL query:** Once a vulnerable input is identified, attackers create an SQL statement intended to be inserted into the query executed by the application. This statement aims to alter the original SQL query to perform actions unintended by the application developers.
3. **Bypassing application security measures:** Attackers often have to bypass protections like input validation or escaping special characters. They achieve this through techniques like string concatenation or utilizing SQL syntax to comment out parts of the original query.
4. **Executing the malicious query:** When the application executes the SQL query, it includes the attacker's malicious input. This modified query can perform actions such as unauthorized viewing of data, deletion of data, or even database schema alterations.
5. **Extracting or manipulating data:** Depending on the attack, the outcome might be the extraction of sensitive information (like user credentials), altering existing data, adding new data, or even deleting significant portions of the database.
6. **Exploiting database server vulnerabilities:** Advanced SQL injections may exploit vulnerabilities in the database server, extending the attack beyond the database to the server level. This can include executing commands on the operating system or accessing other parts of the server's file system.

This process leverages the dynamic execution of SQL in applications where user inputs are directly included in SQL statements without proper validation or escaping. It takes advantage of how SQL queries are built, often in a way that the developers did not anticipate[2].

Real-Life SQL Injection Attack Examples

Over the past 20 years, many SQL injection attacks have targeted large websites, business and social media platforms. Some of these attacks led to serious data breaches. A few notable examples are listed below :

- **GhostShell university attack (2012) :** Team GhostShell, a hacker collective, conducted a major SQL injection attack targeting 53 universities worldwide. They stole and published 36,000 personal records, including data from students, faculty, and staff.
- **Cisco Prime License Manager vulnerability (2018) :** A critical SQL injection vulnerability was found in Cisco Prime License Manager, a tool used to manage software licenses. Attackers could use this flaw to gain shell access to systems, potentially leading to full system control. Cisco quickly patched the vulnerability.
- **7-Eleven breach (2007) :** A group of attackers used SQL injection to compromise the payment systems of several companies, including the 7-Eleven retail chain. This breach led to the theft of over 130 million credit card numbers. The attack was one of the largest data breaches of its time.

- **HBGary hack (2011)** : Hackers associated with the Anonymous group exploited an SQL injection vulnerability to breach the IT security firm HBGary. They took down the company's website and leaked confidential internal communications.[3]

1.3 Techniques of SQL Injection

1.3.1 Error-Based SQL Injection

Error-based SQL injection is a type of security vulnerability and attack that occurs when an attacker injects malicious SQL statements into a web application's input fields, causing the application to generate SQL errors.

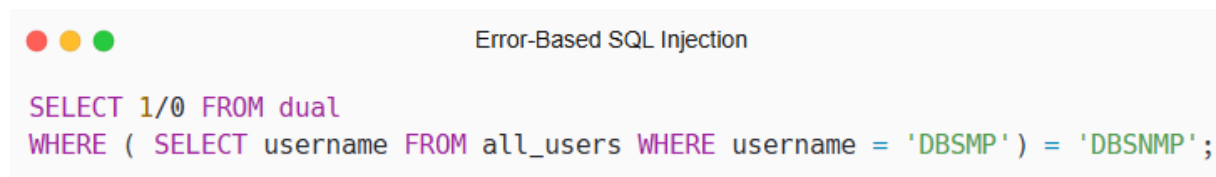
These errors can reveal sensitive information about the application's database structure, data, or configuration[4].

How It Works:

Injection point: An attacker identifies a vulnerable input field, such as a search box or login form, where user input is directly incorporated into SQL queries.

Injecting malicious code: The attacker inputs carefully crafted SQL code as part of their input. This code is designed to cause SQL syntax errors when the application processes it.

Example[5]: Conditional Errors in Oracle/MS-SQL



```
SELECT 1/0 FROM dual
WHERE ( SELECT username FROM all_users WHERE username = 'DBSNMP' ) = 'DBSNMP';
```

Explanation:

1. **Subquery:**
 - The subquery (SELECT username FROM all_users WHERE username = 'DBSNMP') checks if a user named **DBSNMP** exists in the **all_users** table.
2. **Condition:**
 - If the user **DBSNMP** exists, the condition (SELECT username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP' evaluates to **TRUE**.
3. **Error Induction:**
 - When the condition is **TRUE**, the database evaluates the expression **1/0**, which causes a **divide-by-zero error**.
 - If the condition is **FALSE** (i.e., the user does not exist), the expression **1/0** is **not evaluated**, and no error occurs.

Detection:

- If the application returns an **HTTP 500 error** or a database error message, the attacker can infer that the condition is **TRUE** (i.e., the user **DBSNMP** exists).
- If no error occurs, the condition is **FALSE** (i.e., the user does not exist).

Advanced Use Case: Data Exfiltration


Scenario:

- A web application allows users to sort search results using a sort parameter:



```
/search.jsp?department=30&sort=ename
```

- The backend SQL query:



```
SELECT ename, job, deptno, hiredate FROM emp
WHERE deptno = ?
ORDER BY [param_sort] DESC;
```

Malicious Injection:

The attacker injects a payload into the sort parameter to test a condition:



```
/search.jsp?department=20&sort=(SELECT 1/0 FROM dual
WHERE (SELECT SUBSTR(MAX(object_name),1,1) FROM user_objects)='Y');
```

Explanation:

1. Subquery :

- The subquery (SELECT SUBSTR(MAX(object_name),1,1) FROM user_objects) extracts the **first character** of the largest object name in the **user_objects** table.

2. Condition :

- If the first character is 'Y', the condition evaluates to **TRUE**, and the database attempts to evaluate **1/0**, causing a **divide-by-zero error**.
- If the first character is not 'Y', no error occurs, and the query returns results normally.

3. Inference :

- The attacker can use this technique to **brute-force** each character of the object name by testing different values (e.g., 'A', 'B', 'C', etc.).

1.3.2 Blind SQL Injection

Blind SQL (Structured Query Language) injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query's syntax is incorrect. Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the data retrieved from the database is not inserted in the response. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible [6].

1.3.2.1 Content-Based Blind SQL Injection

How It Works:

Unlike traditional SQL injection, where database error messages expose data directly, blind SQL injection does not return query results to the user. Attackers exploit this by providing conditional queries and observing the application's response to infer data from the database.

Technical Explanation:

Blind SQL injection relies on evaluating conditions based on the application's responses. Attackers inject SQL conditions and observe response differences (e.g., message change or page behavior) to deduce data.

Example:


A vulnerable web application allows the attacker to inject SQL on a URL parameter that fetches data based on an id. The attacker confirms the vulnerability by injecting true/false statements in the id parameter to identify between valid and invalid SQL queries.

The attacker first sends a request like:



```
http://newspaper.com/items.php?id=2
```

This executes:



```
SELECT title, description, body FROM items WHERE ID = 2;
```

Next, the attacker tests for SQL injection by adding a false condition:



```
http://newspaper.com/items.php?id=2 and 1=2
```

The query becomes:



```
SELECT title, description, body FROM items WHERE ID = 2 AND 1 = 2;
```

Since $1=2$ is false, the page returns no content, confirming the injection.

Then, the attacker tests a true condition:



```
http://newspaper.com/items.php?id=2 and 1=1
```

The query becomes:



```
SELECT title, description, body FROM items WHERE ID = 2 AND 1 = 1;
```

This query retrieves the anticipated data, revealing the vulnerability. Contrasting the output of these two injections, the attacker can determine that the page is vulnerable to SQL injection and proceed to pull data from the database[6].

Explanation:

Here, the attacker uses a spurious condition ($1=2$) to decide whether the page is vulnerable to SQL injection. Since no information is returned, the attacker confirms the vulnerability. A real condition ($1=1$) provides expected information, ascertaining that the injection has been successful. The attacker now iterates data, e.g., table names or other confidential data, using similar true/false conditions based on the database schema.

1.3.2.2 Time-Based Blind SQL Injection

How It Works:

In time-based blind SQL injection, attackers use SQL functions like `SLEEP()` to introduce a delay in the server's response. If the delay occurs, it indicates the injected SQL condition is true; if not, it is false. This helps attackers extract data even when no visible content is returned.

Technical Explanation:

The attacker sends queries that include conditional delays, such as:

`xyz' AND IF(1=1, SLEEP(5), 0)` – The server delays for 5 seconds, confirming the condition is true.

`xyz' AND IF(1=2, SLEEP(5), 0)` – No delay occurs, confirming the condition is false.


Example:

Consider a web application that retrieves user information based on a user ID provided in the URL:



```
http://example.com/user.php?id=1
```

The corresponding SQL query might be:



```
SELECT * FROM users WHERE id = 1;
```

If the application is vulnerable to Time-Based Blind SQL Injection, an attacker can manipulate the `id` parameter to include a time delay function.

Malicious Injection:




```
http://example.com/user.php?id=1; IF(1=1, SLEEP(5), 0);
```

In this example, the injected SQL statement includes a conditional function that causes the database to pause for 5 seconds if the condition `1=1` is true.

Explanation:

Here, the attacker is inserting a conditional SQL function that will intentionally delay the database's response time. The inserted SQL query is:



```
SELECT * FROM users WHERE id = 1; IF(1=1, SLEEP(5), 0);
```

The `IF(1=1, SLEEP(5), 0)` function will evaluate the condition `1=1`, which is always fulfilled. So the `SLEEP(5)` function will be invoked, and the database will take 5 seconds to reply.

If the application responds in 5 seconds, the attacker confirms successful injection and exposure of the application to Time-Based Blind SQL Injection.

The attackers may use this technique to provide educated guesses on the structure and content of the database even when direct extraction is not possible.

1.3.3 Tautology-Based SQL Injection

The term 'tautology' originates from the field of logic, where it is used to describe a statement that is always true, regardless of the truth values of its components. In other words, a tautological statement is one that is true by virtue of its logical form alone [7].

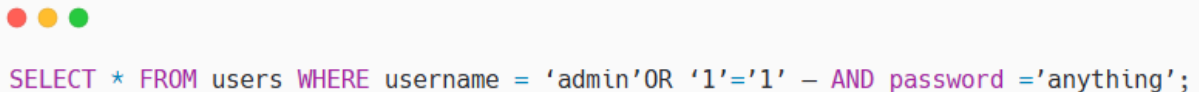
How It Works:

This attack exploits the use of tautological SQL statements.

Always result in true and thus bypasses authentication and other security measures.

Technical Explanation:

A tautology is a logical statement that remains true under any combination of values. Malicious users insert such statements into SQL queries, compelling the database to execute and authenticate unauthorized requests.

A terminal window with a light gray background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a SQL query in a monospaced font: `SELECT * FROM users WHERE username = 'admin'OR '1'='1' - AND password ='anything';`

OR '1'='1' in this example guarantees the condition will always be true, giving access.

Real-World Scenario: An attacker hacks a login form on a web page, bypassing user authentication and accessing an administrator account.

1.3.4 Union-Based SQL Injection

How It Works:

The UNION operator is used in SQL to combine the results of two or more SELECT statements into a single result set. When a web application contains a SQL injection vulnerability that occurs in a SELECT statement, attackers can utilize this operator to insert an additional query and merge its outcome with the outcome of the initial query[5].

Technical Explanation:

Using this method, malicious users can retrieve unauthorized data from the database. UNION-based SQL injection is widely supported by all the major database management systems (DBMS) and is generally the best way to extract specific database contents when query results are directly presented on the application interface.


Example 1[8]: Extracting the Current Database User**Scenario:**

A web application shows product information based on a product ID passed in the URL. The application is susceptible to SQL injection since it puts user input directly into the SQL query without sanitizing. The attacker finds such vulnerability and chooses to exploit it in order to get the current database user, which can assist them in knowing the access level they have and strategize future attacks.

The attacker knows that the application has a database backend (for example, Microsoft SQL Server, MySQL, or Oracle) and wishes to extract the username of the account which issued the queries.

Malicious Injection:

The attacker sends the following malicious URL to the application:




```
http://www.victim.com/products.asp?id=12+union+select+NULL,system_user,NULL,NULL
```

Explanation:

1. Original Query:

The application executes the following query to retrieve product details:




```
SELECT id, type, description, price FROM products WHERE id = 12
```

This query returns the details of the product with ID 12.

2. Injected Query

The attacker appends a UNION SELECT statement to the original query to retrieve the current database user:




```
UNION SELECT NULL, system_user, NULL, NULL
```

- The **UNION** operator combines the results of the original query with the results of the injected query.
- The **system_user** function (or equivalent, depending on the database) retrieves the username of the current database user.
- The **NULL** values are used to match the number of columns in the original query (since the injected query only needs one column for the username, but the original query returns four columns).

3. Combined Query:

The database executes the following combined query:



```
SELECT id, type, description, price FROM products WHERE id = 12  
UNION  
SELECT NULL, system_user, NULL, NULL
```

Result:

Id	Type	Description	Price
12	Book	SQL Injection Attacks	50
NULL	db_user	NULL	NULL

Table 1.1 Results of a SELECT query without UNION**Example 2: Extracting Multiple Rows from the customers Table****Scenario:**

A web application displays product details based on a product **ID** passed in the URL. The application is vulnerable to SQL injection because it directly incorporates user input into the SQL query without proper sanitization. The attacker discovers this vulnerability and decides to exploit it to extract sensitive customer data from the **customers** table in the database.

The attacker's goal is to retrieve the full list of customers (first and last names) from the database.

Malicious Injection:

The attacker sends the following malicious URL to the application:



```
http://www.victim.com/products.asp?  
id=12+union+select+userid,first_name,second_name,NULL+from+customers
```

Explanation:**1. Original Query:**

The application executes the following query to retrieve product details:



```
SELECT id, type, description, price FROM products WHERE id = 12
```

This query returns the details of the product with ID = 12.

2. Injected Query:

The attacker appends a UNION SELECT statement to the original query to retrieve data from the customers table:



```
UNION SELECT userid, first_name, second_name, NULL FROM customers
```

- The **UNION** operator combines the results of the original query with the results of the injected query.
- The **NULL** value is used to match the number of columns in the original query (since the customers table has only three columns, but the original query returns four columns).

3. Combined Query:

The database executes the following combined query:



```
SELECT id, type, description, price FROM products WHERE id = 12  
UNION  
SELECT userid, first_name, second_name, NULL FROM customers
```

Result:

id	Type	Description	Price
12	Book	SQL Injection Attacks	50
1	Charles	Smith	NULL
2	Lydia	Clayton	NULL
3	Bernard	Jones	NULL

Table 1.2 Result of user query after a Union based SQL injection

4 Explanation of the Result:

- The first row represents the product details from the original query.
- The subsequent rows represent the data extracted from **the customers** table, including **userid**, **first_name**, and **second_name**.
- The **NULL** value in the **Price** column is used to align the injected data with the original query's column structure.

1.4 Methods to prevent SQL Injection attacks

Attackers can use SQL injection on an application if it has dynamic database queries that use string concatenation and user supplied input. To avoid SQL injection flaws.

There are simple techniques for preventing SQL injection vulnerabilities and they can be used with practically any kind of programming language and any type of database [9].

1.4.1 Prepared Statements (with Parameterized Queries)

When developers are taught how to write database queries, they should be told to use prepared statements with variable binding (also known as parameterized queries). Prepared statements are simple to write and easier to understand than dynamic queries, and parameterized queries force the developer to define all SQL code first and pass in each parameter to the query later.

If database queries use this coding style, the database will always distinguish between code and data, regardless of what user input is supplied. Also, prepared statements ensure that an attacker cannot change the intent of a query, even if SQL commands are inserted by an attacker.

In PHP, PHP Data Objects (PDO) offer a more effective approach to database interactions. By providing methods that simplify parameterized queries, PDO ensures that user input is always treated as data rather than executable SQL code and enhances code readability and also ensures greater portability across multiple databases.

Prepared Statements example using php[10] :

```
< ?php
$dbh = new PDO('mysql:dbname=testdb;host=127.0.0.1', $user, $password);
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name,:value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);
$stmt->execute();
```

1.4.2 Stored Procedures

Though stored procedures are not always safe from SQL injection, developers can use certain standard stored procedure programming constructs. This approach has the same effect as using parameterized queries, as long as the stored procedures are implemented safely (which is the norm for most stored procedure languages).

Safe Approach to Stored Procedures :

If stored procedures are needed, the safest approach to using them requires the developer to build SQL statements with parameters that are automatically parameterized, unless the developer does something largely out of the norm. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, then called from the application.

Since prepared statements and safe stored procedures are equally effective in preventing SQL injection, your organization should choose the approach that makes the most sense for you.

The following code example call a stored procedure with an input/output parameter using PHP[11].

A code block with a light gray background and rounded corners. At the top left, there are three colored circles: red, yellow, and green. The code is written in a syntax-highlighted style with various colors for different parts of the code (tags, strings, variables, comments, etc.).

```
<?php
$stmt = $dbh->prepare( "CALL sp_takes_string_returns_string(?)" );
$value = 'hello';
$stmt->bindParam(1, $value, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 4000);

// Call the stored procedure
$stmt->execute( );

print "procedure returned: $value\n";
?>
```

1.4.3 Input validation

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.

Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.

Data from all potentially untrusted sources should be subject to input validation, including not only Internet-facing web clients but also backend feeds over extranets, from suppliers, partners, vendors or regulators, each of which may be compromised on their own and start sending malformed data.

Example validating email addresses with `filter_var()` using PHP[12]


```
<?php
$email_a = 'joe@example.com';
$email_b = 'bogus';

if (filter_var($email_a, FILTER_VALIDATE_EMAIL)) {
    echo "Email address '$email_a' is considered valid.\n";
}
if (filter_var($email_b, FILTER_VALIDATE_EMAIL)) {
    echo "Email address '$email_b' is considered valid.\n";
} else {
    echo "Email address '$email_b' is considered invalid.\n";
}
?>
```

1.4.4 Escaping All User-Supplied Input

In this approach, the developer will escape all user input before putting it in a query. It is very database specific in its implementation. This methodology is frail compared to other defenses, and we CANNOT guarantee that this option will prevent all SQL injections in all situations.

If an application is built from scratch or requires low risk tolerance, it should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you.

1.5 Conclusion

SQL injection remains a very important security threat to web applications. Some of the basic security measures that, if applied correctly, can protect against these attacks are input validation, prepared statements, and escaping user inputs. Other steps include restricting the functions that the user is allowed to perform. But no single method can fully guarantee protection from attacks of multitude types and changing SQL injection techniques. An attacker's primary goal is searching for a new way to bypass security, especially when not properly applied; this may result in a serious breach of sensitive data. For these reasons, these security steps must be followed consistently, regularly reviewed as part of a full security strategy, and supported by AI-based tools that help catch and block advanced and automated SQL injection attacks.

To gain a better understanding of how AI-based tools operate, it is critical to explore the areas of machine learning and deep learning. These enable systems to learn from data, identify complex patterns, and upgrade themselves over time. A new dimension that significantly enhances cybersecurity. Hence, the following chapter will explain the background of machine learning, its various types and algorithms, and then proceed to deep learning architectures.

Chapter 2

Machine Learning and Deep Learning

2.1 Introduction

The field of **Artificial Intelligence (AI)**, using the strongest tools available in computer science, works toward imitating intelligence in a human being. These systems can perform a variety of tasks typically attributed to human cognitive abilities, such as decision-making, pattern recognition, and problem-solving. Artificial intelligence has come a long way over the years, fueling innovations such as self-driving cars, intelligent virtual assistants, and highly advanced recommendation systems, revolutionizing industries and everyday life.

In this part, basic machine-learning (ML) methodologies are looked into, a major subfield of AI. We will describe the three paradigms of learning: supervised learning, unsupervised learning, and reinforcement learning. Standard algorithms in machine learning will also be addressed followed by a transition into deep learning (DL), which is an enhanced version of ML that exploits multi-layer neural networks. The immediate goal in this instance is to firmly establish some of the fundamental concepts of these methods and their frameworks, in preparation for their application to real-world problems, including cybersecurity and SQL injection detection.

2.2 Machine learning

The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience. In recent years many successful machine learning applications have been developed, ranging from data-mining programs that learn to detect fraudulent credit card transactions, to information-filtering systems that learn users' reading preferences, to autonomous vehicles that learn to drive on public highways. At the same time, there have been important advances in the theory and algorithms that form the foundations of this field.[13]

2.2.1 Machine Learning Types

In machine learning, this kind of learning process is usually classed into three main parts which are supervised learning, unsupervised learning, and reinforcement learning. Each type serves a certain distinct purpose and is used with certain types of problems. Besides those three types, hybrid approaches such as commodity inclusion and special techniques have also emerged to maintain and solve more complex issues.

2.2.1.1 Supervised Learning (SL):

Supervised learning (SL) is a machine learning approach that leverages labelled data to educate a system in forecasting outcomes based on its training. It closely mimics the process of

human learning under the guidance of an instructor, employing specific instances to deduce overarching principles. SL is typically divided into two main categories.

Regression: Regression is a term used in statistics, which is a type of statistical analysis that aims to understand the relationship between a dependent variable (response variable) and one or more independent variables (predictors) such as in market trends or weather forecasting. The most common type is linear regression

A typical example of regression is **house price prediction**, where features like house size, number of rooms, and location are used to estimate the price.

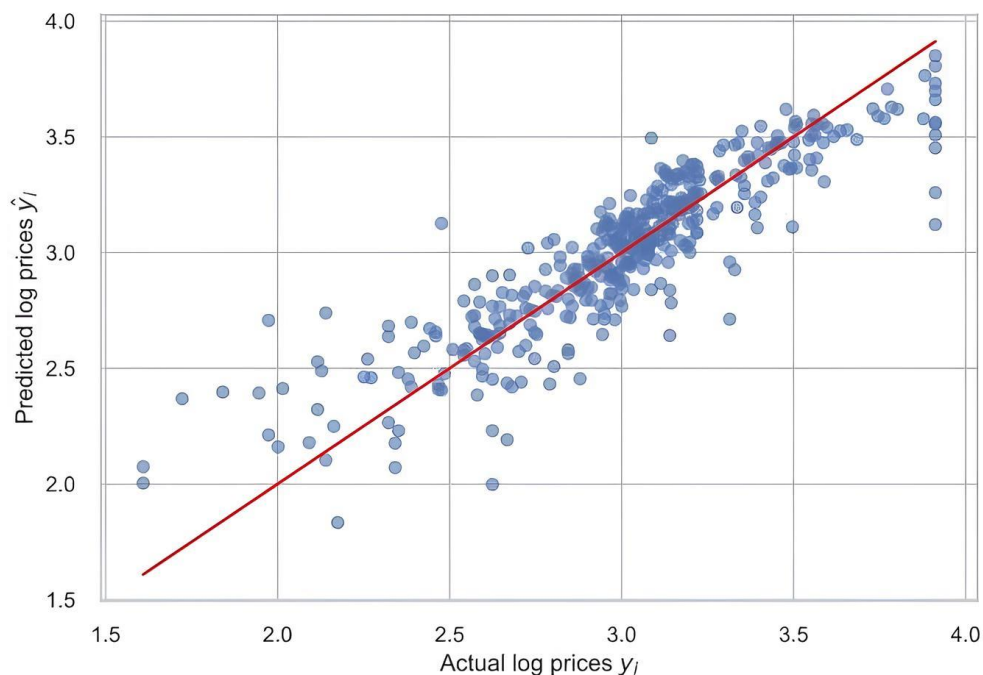


Figure 2.1 Linear Regression Example – House Price Prediction

Classification: Classification is a SL technique that involves categorizing data into distinct classes. It is a predictive process that recognizes and groups data objects into pre-defined categories or labels. This technique is used to predict the outcome of a given problem based on input features. It can be applied to structured or unstructured data, and the classes are commonly known as target, label, or categories. The aim of classification is to assign an unknown pattern to a known class. For example, classifying emails as "spam" or "not spam" is a common application of classification.

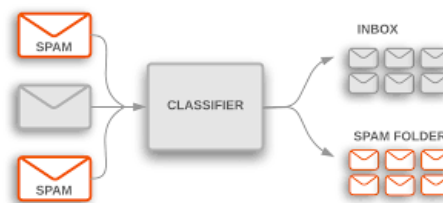


Figure 2.2 Classification Example – Spam vs Not Spam Emails

Both the Classification and Regression algorithms can be used for forecasting in machine learning and operate with the labelled datasets. But the distinction between classification vs regression is how they are used on particular machine learning problems. [14]

2.2.1.2 Unsupervised Learning:

Unsupervised learning, also known as unsupervised machine learning, uses machine learning algorithms to analyze and cluster unlabeled datasets (subsets called clusters). These algorithms discover hidden patterns or data groupings without the need for human intervention.

Unsupervised learning's ability to discover similarities and differences in information make it ideal for exploratory data analysis, cross-selling strategies, customer segmentation, and image and pattern recognition. It's also used to reduce the number of features in a model through the process of dimensionality reduction. Principal component analysis (PCA) and singular value decomposition (SVD) are two common approaches for this. Other algorithms used in unsupervised learning include neural networks, k-means clustering, and probabilistic clustering methods.[15]

2.2.1.3 Reinforcement Learning:

Reinforcement learning problems involve learning how to map situations to actions to maximize a numerical reward signal. These problems are inherently closed-loop, as the system's actions influence its future inputs. Unlike other forms of machine learning, the learner is not explicitly told which actions to take but must discover the best ones through trial and error. In more complex scenarios, actions impact not only immediate rewards but also future states and long-term rewards, making decision-making more challenging.[16]

2.2.2 Machine learning algorithms

2.2.2.1 Logistic regression

Logistic regression is a supervised machine learning algorithm used for classification tasks, predicting the probability that an instance belongs to a specific class. It is a statistical method that analyzes the relationship between independent variables and a categorical outcome.

Logistic regression applies the sigmoid function to map input values to a probability ranging between 0 and 1. Instead of fitting a regression line, it models an "S"-shaped curve to distinguish between classes.[17]

Key Points:

- Logistic regression predicts the output of a categorical dependent variable.
- The outcome is discrete (e.g., Yes/No, 0/1, True/False) but represented as a probability between 0 and 1

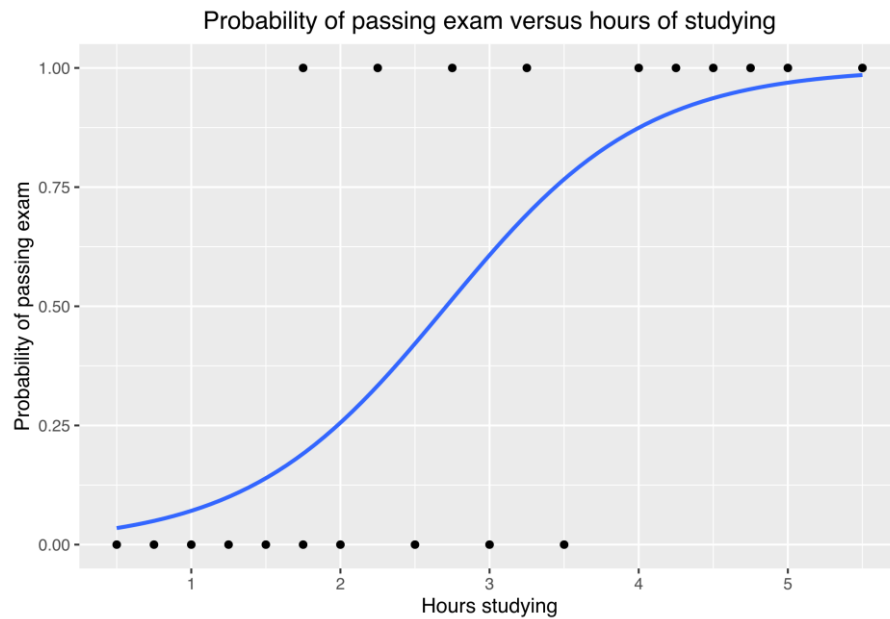


Figure 2.3 Example Curve: How Study Hours Affect Exam Success Rate

2.2.2.2 Support Vector Machine (SVM)

A support vector machine (SVM) is a supervised learning algorithm used for many classification and regression problems, including signal processing, medical applications, natural language processing, and speech and image recognition.

The objective of the SVM algorithm is to find a hyperplane that, to the best degree possible, separates data points of one class from those of another class. “Best” is defined as the hyperplane with the largest margin between the two classes, represented by plus versus minus in the figure below. Margin means the maximal width of the slab parallel to the hyperplane that has no interior data points. Only for linearly separable problems can the algorithm find such a hyperplane; for most practical problems, the algorithm maximizes the soft margin, allowing a small number of misclassifications.

Support vectors are a subset of the training observations that define the position of the separating hyperplane. These are the data points closest to the hyperplane and are crucial for determining the optimal margin. While SVMs are originally designed for binary classification, they can be extended to multiclass problems by combining multiple binary classifiers.

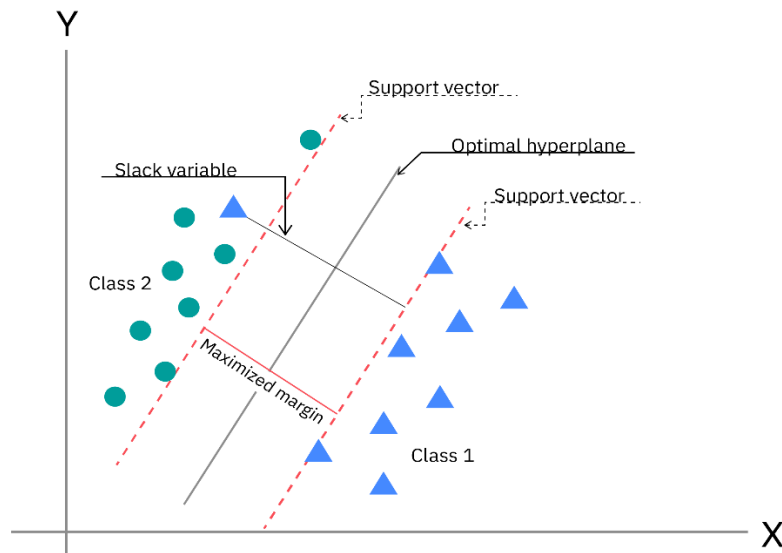


Figure 2.4 SVM: Optimal Hyperplane and Support Vectors

To handle non-linearly separable data, SVMs use kernel methods. Kernel functions transform the data into a higher-dimensional space, where classes may become linearly separable. This transformation is implicit and computationally efficient, thanks to the kernel trick. Common kernel types include:

- Linear: Suitable for linearly separable data.
- Polynomial: Captures polynomial relationships between features.
- Radial Basis Function (RBF): Effective for complex, non-linear boundaries.
- Sigmoid: A Mercer kernel under specific conditions.

Training an SVM involves solving a quadratic optimization problem to find the hyperplane that minimizes the soft margin. The number of transformed features depends on the number of support vectors, making the model compact and efficient once trained.

Key advantages of SVMs include their ability to handle high-dimensional data, robustness to outliers, and effectiveness in solving non-linear problems using kernels. Once trained, only the support vectors are needed to define the decision boundary, making SVMs suitable for automated code generation and real-world applications.[\[18\]](#)

2.2.3 Real-world machine learning use cases

Here are just a few examples of machine learning you might encounter every day:

Speech recognition: It is also known as automatic speech recognition (ASR), computer speech recognition, or speech-to-text, and it is a capability which uses natural language processing (NLP) to translate human speech into a written format. Many mobile devices incorporate speech recognition into their systems to conduct voice search—e.g. Siri—or improve accessibility for texting.

Customer service : Online chatbots are replacing human agents along the customer journey, changing the way we think about customer engagement across websites and social media platforms. Chatbots answer frequently asked questions (FAQs) about topics such as shipping, or provide personalized advice, cross-selling products or suggesting sizes for users. Examples include virtual agents on e-commerce sites, messaging bots, using Slack and Facebook Messenger, and tasks usually done by virtual assistants and voice assistants.

Computer vision : This AI technology enables computers to derive meaningful information from digital images, videos, and other visual inputs, and then take the appropriate action. Powered by convolutional neural networks, computer vision has applications in photo tagging on social media, radiology imaging in healthcare, and self-driving cars in the automotive industry.

Recommendation engines: Using past consumption behavior data, AI algorithms can help to discover data trends that can be used to develop more effective cross-selling strategies. Recommendation engines are used by online retailers to make relevant product recommendations to customers during the checkout process.

Robotic process automation (RPA): Also known as software robotics, RPA uses intelligent automation technologies to perform repetitive manual tasks.

Automated stock trading: Designed to optimize stock portfolios, AI-driven high-frequency trading platforms make thousands or even millions of trades per day without human intervention.

Fraud detection: Banks and other financial institutions can use machine learning to spot suspicious transactions. Supervised learning can train a model using information about known fraudulent transactions. Anomaly detection can identify transactions that look atypical and deserve further investigation.[19]

2.3 Deep Learning

Deep learning is a subfield of machine learning that employs deep neural networks for analyzing and interpreting complex data. Such networks are modeled after the human brain and allow the computer to identify patterns and relationships without human intervention in large amounts of unstructured information. The deep learning model is continuously improving its accuracy by tuning internal parameters with training.

Deep learning models can be trained to perform classification tasks and recognize patterns in images, text, audio, and other types of data. This technology also enables automation of tasks that typically require human intelligence, such as image description and audio transcription. Where human brains have millions of interconnected neurons that work together to learn information, deep learning features neural networks constructed from multiple layers of software nodes that work together.

This technique has achieved astonishing outcomes in image recognition, understanding natural language, and processing speech, making it the foundation of contemporary artificial intelligence systems.[20]

2.3.1 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANN) are inspired by the way biological neural system works, such as the brain process information. The information processing system is composed of a large number of highly interconnected processing elements (neurons) working together to solve specific problems. ANNs, just like people, learn by example. Similar to learning in biological systems, ANN learning involves adjustments to the synaptic connections that exist between the neurons.

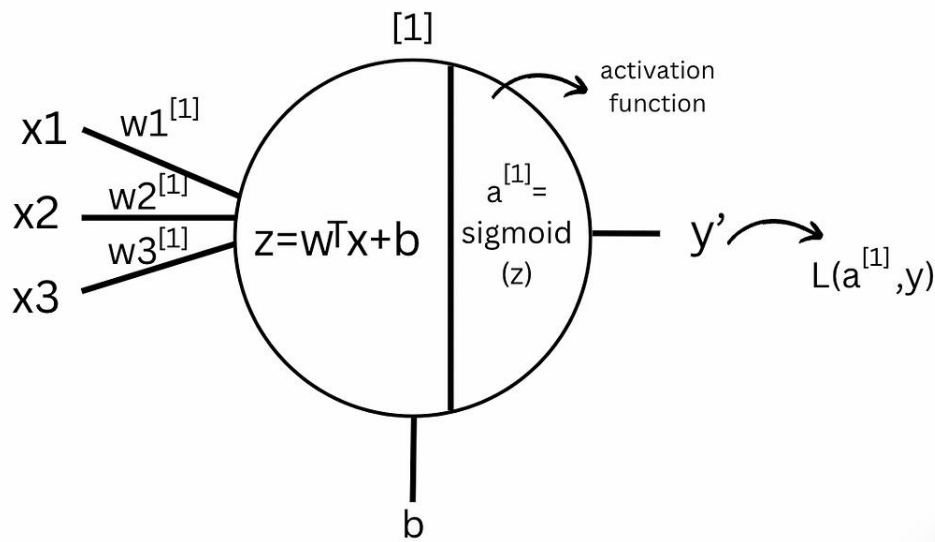


Figure 2.5 Neuron Computation in an Artificial Neural Network

Here $[X_1, X_2, X_3]$ are the input features to the neuron represented as X . Whereas the superscript is used to denote the layer. The weights are denoted by $[w_1, w_2, w_3]$ associated with each connection to the neuron from the input of that particular layer. The bias is represented by b associated with the neuron. “ z ” is the weighted sum of inputs added with the bias which is linear by nature. “ a ” is the activation function that is applied to z to add non-linearity as complex models can't be represented as a line.

n = the number of inputs
from the incoming layer

$$f \left(b + \sum_{i=1}^n x_i w_i \right)$$

Figure 2.6 The Mathematic of Neural Networks

The activation function is applied to the weighted sum of inputs to the neuron, including the bias term, and the resulting value becomes the neuron's output, which is then passed to the next layer. Its primary role is to introduce non-linearity into the model, allowing the network to learn complex patterns and approximate any arbitrary function. In this one-layer neural network architecture, the output of the activation function in layer serves as the final output, denoted as y' . This output is used to compute the loss function, $L(a, y)$, which measures the deviation between the predicted and actual output. This deviation is crucial for backpropagation and optimization, which will be discussed in later sections.[21]

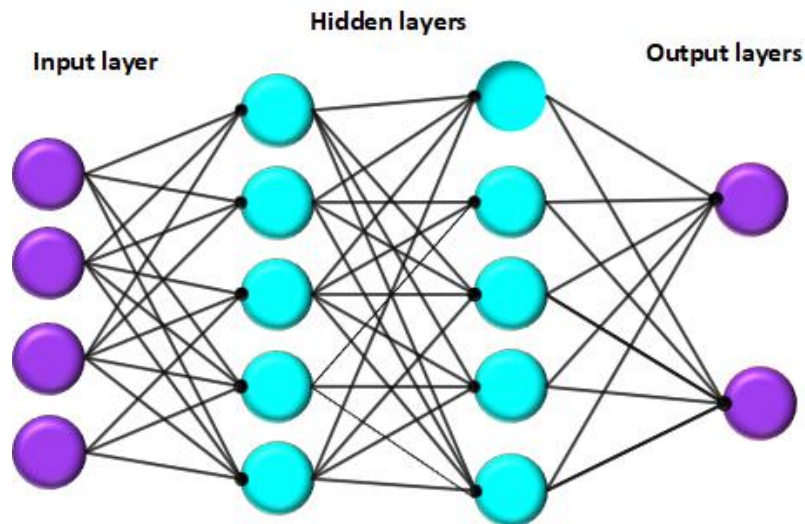


Figure 2.7 Basic Structure of an Artificial Neural Network

An artificial neural network is primarily composed of three “types” of layers: the input layer, one or more hidden layers, and the output layer. These layers collectively work to process the information and yield meaningful predictions.

The input layer serves as the entry point for data into the neural network. Each neuron in this layer corresponds to either a specific feature of the input dataset or an input vector. For instance, in an image classification problem, all input neurons may specify the intensity of each individual pixel. This layer exists purely for the purpose of passing on the raw input values to the next layer without change.

The hidden layers serve to process and transform information passed to them from the input layer. They are called hidden since nothing about their inner workings can be observed directly. In a typical hidden layer, a neuron takes in inputs from the preceding layer, applies a weighted summation, adds a bias, and applies an activation function on the result. Depending on the complexity of the task, a network can have varying numbers of hidden layers with varying numbers of neurons in each layer. DNNs having multiple hidden layers are quite popular in applications such as image recognition, speech processing, and natural language understanding since they can efficiently extract many complex patterns from the data.

The output layer is the final layer of the network that provides the prediction made by the model. The structure of this layer is based on the solved problem. In the case of classification networks, each output neuron corresponds to one class; in regression problems, usually only one output neuron provides a continuous value: an activation function is then used in the output layer according to the type of task, e.g., softmax for multi-class classification and any linear function for regression.

These layers are interconnected by **weighted links** that determine the importance of each input. The weights are adjusted during training.

2.3.4 Deep Learning Training Cycle

1 Data input and preprocessing: First load the data and clean it, removing any missing values or corrupted entries, and outliers. Next is to transform inputs into a uniform format. For example, numerical features should be normalized or scaled to lie in a common range, and any required data augmentation strategies should be applied. The last step is splitting the data into three sets: training, validation, and test sets so that the model is evaluated on new or unseen data.

2 Forward propagation : Pass each training sample through the network layer by layer to produce an output prediction. At each layer, the inputs are combined with the weights and biases of that layer and then passed through an activation function. The output from a "forward pass" through the network shows the final network output based on the current parameters.

3 Loss function calculation: Measure the quality of a prediction by comparing it to the true target using a loss function. The loss function quantifies the error, for example, with regression, mean squared error could be used, whereas with classification, it would be cross-entropy. In other words, the loss measures, in a quantitative manner, the difference between the predicted output and the actual label, a larger difference leads to a larger loss value.

4 Backpropagation: Compute the gradient of the loss with respect to each weight by propagating the error backward through the network. During this backward pass, the chain rule is used to determine how much each weight contributed to the final error. Essentially, backpropagation can create an error signal at each layer so that it is clear how to change the weights in order to reduce the error.

5 Weight updates (optimization): Use an optimization algorithm (stochastic gradient descent or Adam) to update the weights and biases of the network based on the gradients obtained from backpropagation. For example, stochastic gradient descent (SGD) subtracts a fraction (the learning rate) of the gradient from each weight, moving it in the direction that reduces loss. More advanced optimizers like Adam adapt the learning rate for each parameter and include momentum terms (combining ideas from momentum and RMSprop) to speed up convergence.

2.3.2 Activation functions

Activation functions play a fundamental role in neural networks by determining how neurons process input data and transfer information to subsequent layers. The choice of activation function significantly influences the network's performance and learning capability

2.3.2.1 Non-Linear Activation Functions

A network using only a linear activation functions is essentially equivalent to a simple linear model, limiting its ability to capture complex patterns in data. Non-linear activation functions enable deep networks to model intricate relationships between inputs and outputs.

They allow backpropagation by ensuring derivatives depend on input values, facilitating effective weight adjustments. They also enable the creation of deep networks, where transformed outputs from one layer pass non-linearly to the next, improving the model's ability to learn complex representations.

1. Sigmoid (Logistic) Activation Function

This function takes any real value as input and outputs values in the range of 0 to 1 making it useful for probabilistic models and binary classification tasks.

The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.

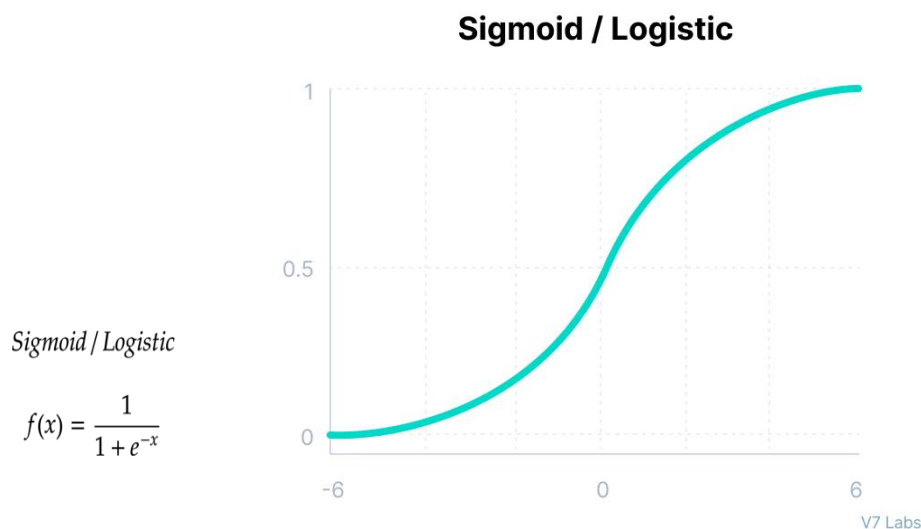


Figure 2.8 Sigmoid Activation Function

It is ideal for probability-based applications due to its constrained output range and is differentiable, ensuring smooth gradient updates during optimization. However, it suffers from the vanishing gradient problem, as extreme values lead to near-zero derivatives, hindering learning. Additionally, it is not zero-centered, which can slow down the training process.

2. Tanh (Hyperbolic Tangent) Function

The Tanh function is similar to the sigmoid function but maps input values to a range between -1 and 1, providing stronger non-linearity.

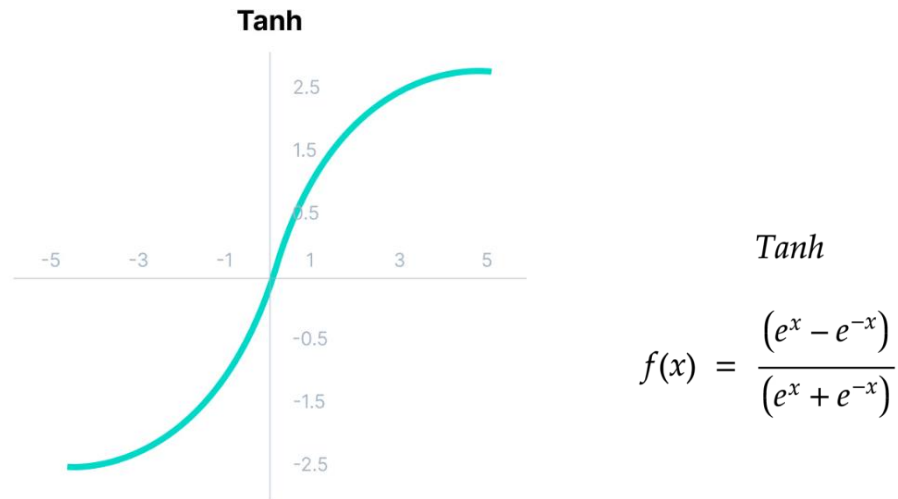


Figure 2.9 Tanh (Hyperbolic Tangent) Function

Its outputs are zero-centered, which improves convergence speed in deep networks, and it is often used in recurrent neural networks (RNNs) and convolutional neural networks (CNNs). However, it still suffers from the vanishing gradient problem, albeit less than the sigmoid function.

3. ReLU (Rectified Linear Unit) Function

ReLU is one of the most commonly used activation functions in deep learning. It introduces non-linearity by returning zero for negative inputs while retaining positive values unchanged.

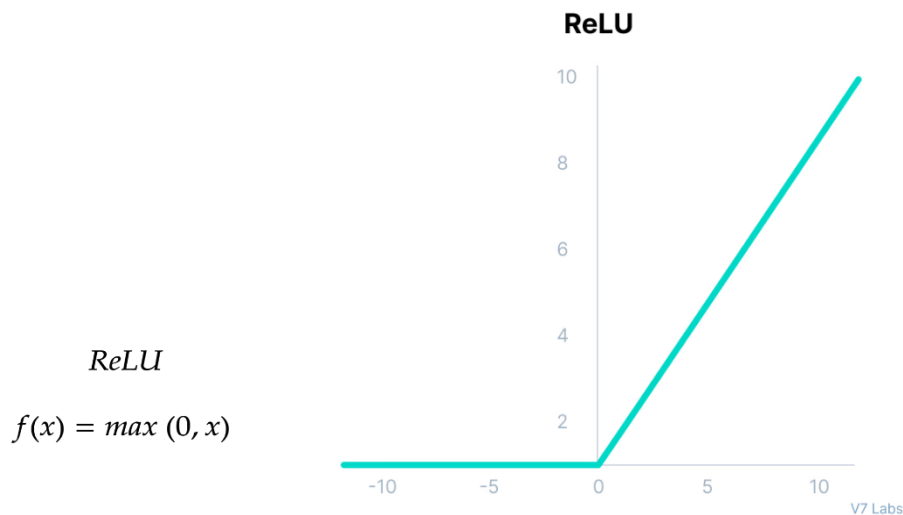


Figure 2.10 ReLU (Rectified Linear Unit) Function

It is computationally efficient, as only a subset of neurons activates at a time, and helps accelerate gradient descent convergence due to its non-saturating nature. However, it suffers

from the Dying ReLU problem, where neurons can become permanently inactive for negative inputs, preventing further updates.[22]

2.3.3 Deep learning architectures

Deep learning has enjoyed tremendous advancement in the last few years, serving as the major pillar for innovation in many different fields. Each architecture is designed for particular problems, with the performance in each case optimized for the specific needs of the task at hand.

Over the years, many deep learning models have been developed, often extending some fundamental designs. Among these, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and long short-term memory networks (LSTMs) are the most commonly known. In their respective areas, these architectures have been very efficient, thereby enabling progress in image recognition, sequence modeling, and time series analysis.

2.3.3.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are neural networks designed to recognize patterns in sequences of data. They're used for identifying patterns such as text, genomes, handwriting, or numerical time series data from stock markets, sensors, and more.

Unlike traditional feedforward neural networks, where inputs are processed only once in a forward direction, RNNs possess a unique feature: They have loops in them, allowing information to persist.

This looping mechanism enables RNNs to remember previous information and use it to influence the processing of current inputs. This is like having a memory that captures information about what has been calculated so far, making RNNs particularly suited for tasks where the context or the sequence is crucial for making predictions or decisions.[23]

Structure of RNNs

RNNs are made of neurons: data-processing nodes that work together to perform complex tasks. The neurons are organized as input, output, and hidden layers. The input layer receives the information to process, and the output layer provides the result. Data processing, analysis, and prediction take place in the hidden layer.

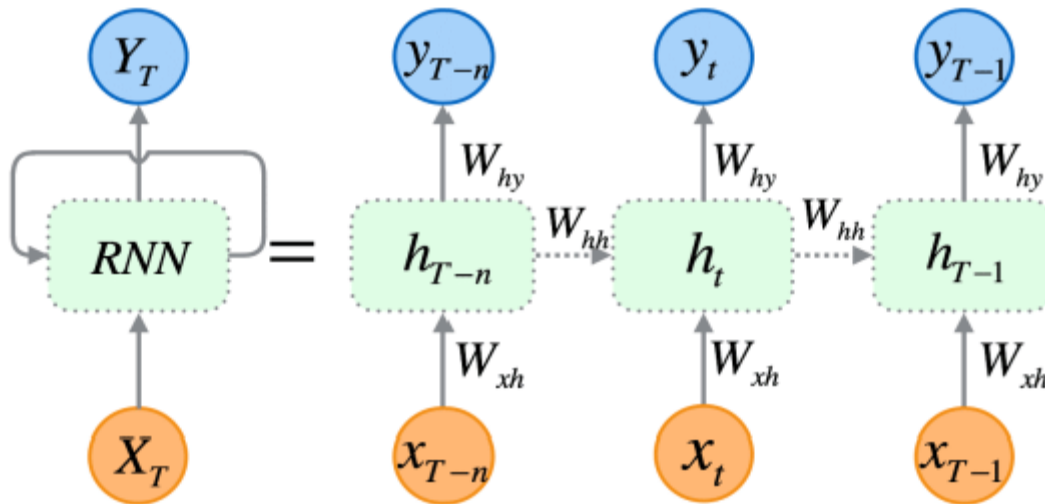


Figure 2.11 Recurrent Neural Network (RNN) Architecture

The diagram illustrates the unrolled RNN over time steps, showing:

- **Inputs (x_t):** Sequential data (e.g., words in a sentence).
- **Hidden states (h_t):** Memory units with recurrent weights (W_{hh}).
- **Outputs (y_t):** Predictions at each step.
- **Weight matrices (W_{xh}, W_{hh}, W_{hy}):** Shared across time steps for efficiency

Hidden layer

RNNs work by passing the sequential data that they receive to the hidden layers one step at a time. However, they also have a self-looping or recurrent workflow: the hidden layer can remember and use previous inputs for future predictions in a short-term memory component. It uses the current input and the stored memory to predict the next sequence.

For example, consider the sequence: Apple is red. You want the RNN to predict red when it receives the input sequence Apple is. When the hidden layer processes the word Apple, it stores a copy in its memory. Next, when it sees the word is, it recalls Apple from its memory and understands the full sequence: Apple is for context. It can then predict red for improved accuracy. This makes RNNs useful in speech recognition, machine translation, and other language modeling tasks.

Training

Recurrent Neural Networks (RNNs) are trained by feeding them sequences of data and adjusting their internal parameters specifically the weights to minimize prediction errors. In an RNN, the same set of weights is shared across all time steps, which allows the network to process sequential information efficiently and retain temporal dependencies.

The training process involves computing a loss between the predicted output and the actual target, then updating the weights to reduce this loss using gradient descent. To compute

gradients in an RNN, a specialized form of backpropagation called Backpropagation Through Time (BPTT) is used.

BPTT works by unfolding the RNN over time, treating it as a feedforward network across the sequence of time steps. It computes the error at each time step and then propagates gradients backward through the entire sequence. These gradients are accumulated and used to adjust the weights, allowing the model to learn dependencies across time and improve its prediction accuracy.

Types of recurrent neural networks:

RNNs are often characterized by one-to-one architecture: one input sequence is associated with one output. However, you can flexibly adjust them into various configurations for specific purposes. The following are several common RNN types.

One-to-One

A One-to-One RNN is the simplest architecture where a single input maps to a single output. It's typically used for tasks like classification, where the entire input sequence is processed to produce one prediction.

One-to-many

This RNN type channels one input to several outputs. It enables linguistic applications like image captioning by generating a sentence from a single keyword.

Many-to-one

The model uses multiple inputs to predict multiple outputs. For example, you can create a language translator with an RNN, which analyzes a sentence and correctly structures the words in a different language.

Many-to-many

Several inputs are mapped to an output. This is helpful in applications like sentiment analysis, where the model predicts customers' sentiments like positive, negative, and neutral from input testimonials.[\[24\]](#)

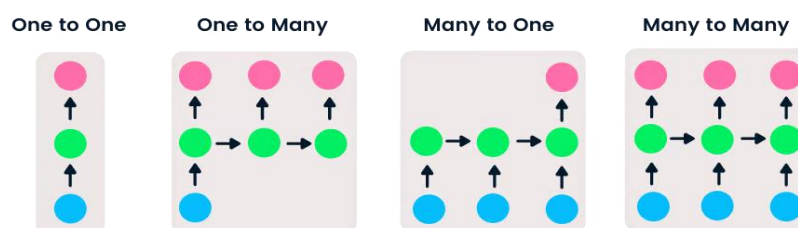


Figure 2.12 Types of RNN's

Limitations of RNNs

Despite their effectiveness, RNNs suffer from several limitations that impact their performance, especially when dealing with long sequences:

Vanishing gradient problem: One of the significant drawbacks of basic RNNs is the vanishing gradient problem. It occurs when gradients during training become extremely small as they are backpropagated through time. This limits the network's ability to capture long-range dependencies.

Exploding gradient problem: RNNs can also suffer from the exploding gradient problem, where gradients become exceptionally large during training, causing numerical instability. Exploding gradient is easier to detect and manage.

Limited memory: Traditional RNNs have a limited memory capacity, and they struggle to carry information across many time steps. This can be problematic when dealing with long sequences where the network may "forget" important information from earlier time steps.[\[25\]](#)

2.3.3.2 Long Short-Term Memory Networks

Traditional RNNs struggle with long-term dependencies due to the vanishing and exploding gradient problem. To address this, **Long Short-Term Memory (LSTM) networks** were introduced.

A long short-term memory (LSTM) network is a type of recurrent neural network (RNN). LSTMs are predominantly used to learn, process, and classify sequential data because they can learn long-term dependencies between time steps of data.

Architecture and Functioning

LSTM layers use additional gates to control what information in the hidden state is exported as output and to the next hidden state. These additional gates overcome the common issue with RNNs in learning long-term dependencies. In addition to the hidden state in traditional RNNs, the architecture for an LSTM block typically has a memory cell, input gate, output gate, and forget gate. The additional gates enable the network to learn long-term relationships in the data more effectively. Lower sensitivity to the time gap makes LSTM networks better for analyzing sequential data than simple RNNs. In the figure below, you can see the LSTM architecture and data flow at time step t .

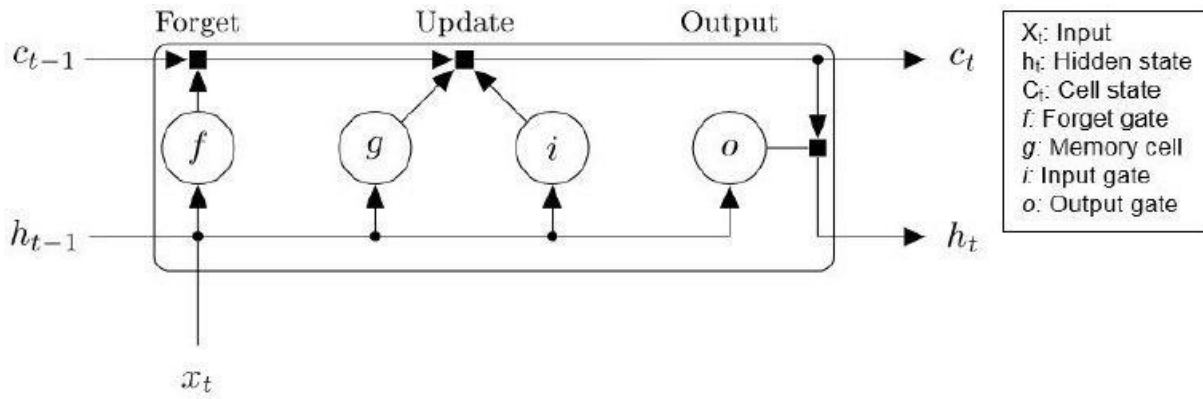


Figure 2.13 LSTM Gate Mechanisms

Data flow at time step t for an LSTM unit. The forget gate and memory cell prevent the vanishing and exploding gradient problems.

These formulas define the internal mechanisms of the LSTM cell and correspond to the diagram above.

As illustrated in Figure 2.13, the diagram shows the equations used to compute the forget gate f_t , input gate i_t , candidate memory C_t , updated cell state c_t , output gate o_t , and hidden state h_t .

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
 c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
 h_t &= o_t \circ \sigma_h(c_t)
 \end{aligned}$$

Figure 2.14 Mathematical Formulation of LSTM Gates and States.

Gate Mechanisms and Information Flow

The weights and biases to the input gate control the extent to which a new value flows into the LSTM unit. Similarly, the weights and biases to the forget gate and output gate control the extent to which a value remains in the unit and the extent to which the value in the unit is used to compute the output activation of the LSTM block, respectively.

The following diagram illustrates the data flow through an LSTM layer with multiple time steps. The number of channels in the output matches the number of hidden units in the LSTM layer

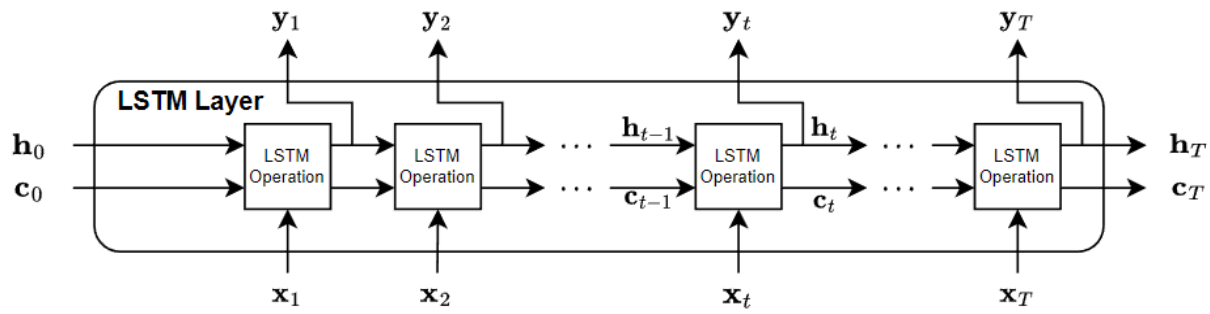


Figure 2.15 LSTM Layer Architecture and Operations

Data flow for an LSTM with multiple time steps. Each LSTM operation receives the hidden state and cell state from the previous operation and passes an updated state and cell state to the next operation.

Applications of LSTM Networks:

LSTMs work well with sequence and time-series data for classification and regression tasks. LSTMs also work well on videos because videos are essentially a sequence of images. Similar to working with signals, it helps to perform feature extraction before feeding the sequence of images into the LSTM layer. Leverage convolutional neural networks (CNNs) (e.g., GoogLeNet) for feature extraction on each frame. The following figure shows how to design an LSTM network for different tasks.[\[26\]](#)

2.3.3.4 Transformers

Recurrent neural networks (RNNs), including LSTMs and GRUs, are state-of-the-art for sequence modeling and transduction tasks like machine translation. However, their sequential computation limits parallelization, especially for long sequences. Attention mechanisms improve dependency modeling but are typically used with RNNs.

The Transformer eliminates recurrence, relying entirely on attention for global dependencies, enabling greater parallelization and state-of-the-art results. Unlike convolutional models (e.g., ByteNet, ConvS2S), the Transformer reduces operations for distant dependencies to a constant. It is the first model to use only self-attention, avoiding RNNs or convolutions.

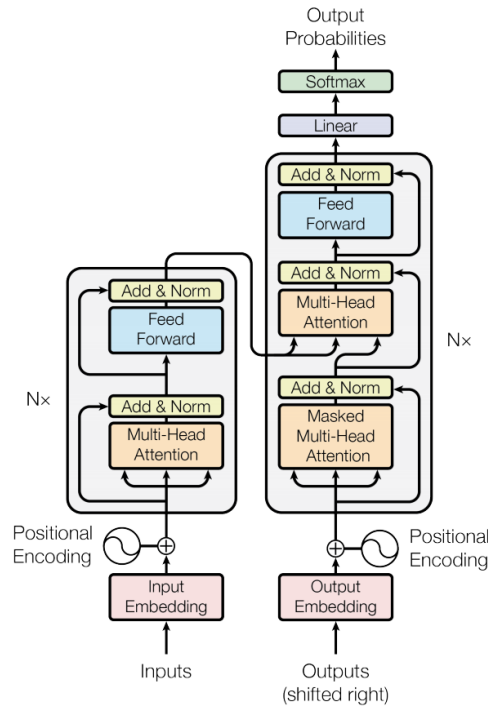


Figure 2.16 Transformer Model Architecture with Multi-Head Attention

1. Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical encoders. Each encoder has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical Decoders. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i .

2. Embeddings and Softmax

Similarly to other sequence transduction models, learned embeddings are used to convert the input tokens and output tokens to vectors of dimension d_{model} . The usual learned linear transformation and softmax function are used to convert the decoder output to predicted next-token probabilities. In this model, the same weight matrix is shared between the two embedding

layers and the pre-softmax linear transformation, similar to. In the embedding layers, those weights are multiplied by $\sqrt{d_{\text{model}}}$.

3. Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.[27]

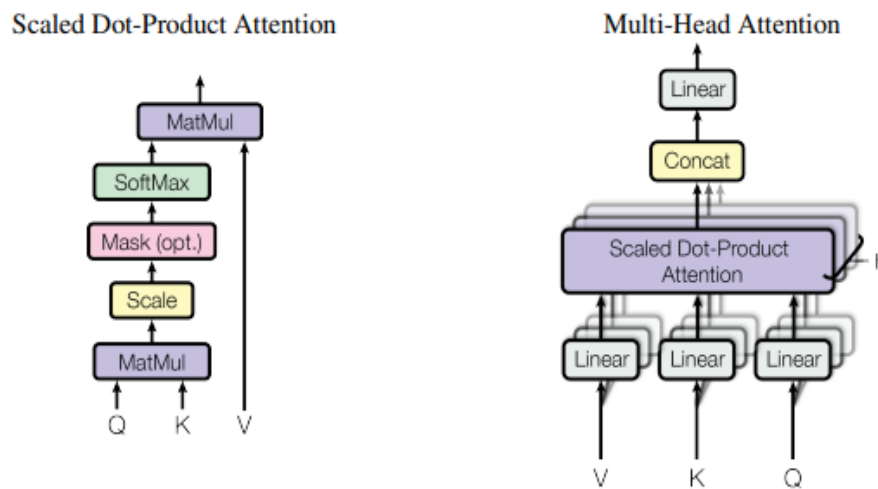


Figure 2.17 Scaled Dot-Product Attention Mechanisms

3.1 Scaled Dot-Product Attention

The scaled dot-product attention mechanism is an important part of Transformer architecture and works with queries, keys, and values as the input. First, now, the model computes a dot product over the similarity of each query with all the keys. To avoid any training instability due to very large similarity scores, the values of these similarity scores are scaled down by dividing them by the square root of the key dimension.

Each word in a sequence is mapped to three vectors:

- Q = Query
- K = Key
- V = Value

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Figure 2.18 Self-Attention Equation

These scaled scores are subsequently put through a softmax, which returns the normalized weights. The weights tell the model how much importance to attach to each value whilst combining them into a final output. This mechanism enables an efficient focus on only the relevant parts of the input. Scaled dot-product attention, being much faster and memory-efficient compared to additive attention that involves computing similarities with a small neural network, perfectly fits the bill in case of large-scale applications.

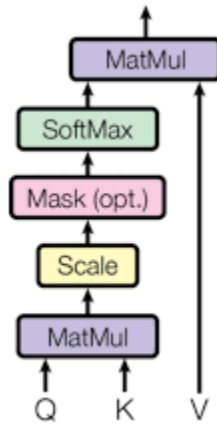


Figure 2.19 Scaled Dot-Product Attention Mechanisms

3.2 Multi-Head Attention

Multi-head attention is extension of the basic attention mechanism in that it enables the model to jointly attend to different regions of the input simultaneously. Instead of computing one attention function, it computes multiple projections of the queries, keys, and values using different learned projections. They are computed in parallel and concatenated, projected one last time to produce the final output.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Figure 2.20 Multi-Head Attention Equation

This enables the model to capture diverse information from different representation subspaces, thus it is better suited to handle complex patterns. For example, Transformers typically consist of eight attention heads that run on reduced dimensions in an attempt to provide computational efficiency.

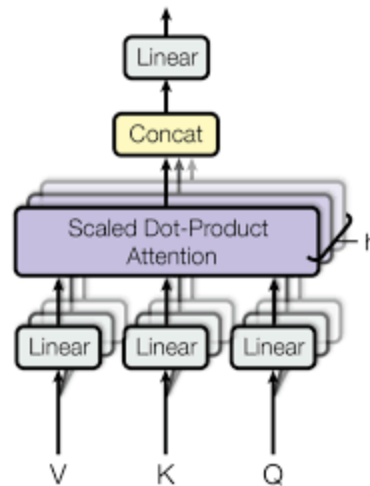


Figure 2.21 Multi-Head Attention Mechanisms

4. Positional Encoding

Transformers lack inherent information about the order of the input sequence due to their parallel processing nature. Positional encoding is introduced to provide the model with information about the position of each token in the sequence.

Positional encodings are added to the input embeddings to give the model a sense of token order. These encodings can be either learned or fixed. [28]

In the original Transformer architecture, fixed sinusoidal functions are used for encoding positions.

5. Point-wise Feedforward Network

Point-wise feedforward networks are used extensively in transformer architectures. These networks consist of two linear transformations with a ReLU activation in between. The first linear transformation expands the dimensionality of each input position and the second reduces it back to the original dimension. This layer operates independently on each position within the input sequence, which allows it to efficiently process long sequences in parallel. Its significance lies in its ability to introduce non-linearity and increase the model's capacity to capture complex features, improving the performance of tasks like language modeling, translation, and more.

The mathematical formulation is:

$$\text{FFN}(x) = W_2 (\text{ReLU}(W_1 x + b_1)) + b_2$$

Figure 2.22 Feedforward Network Equation

where W_1 expands dimensions (e.g., from d_{model} to d_{ff}), and W_2 compresses back to d_{model} .

The point-wise feedforward network concept gained prominence with the introduction of the Transformer model in the seminal paper "Attention Is All You Need" by Vaswani et al. in 2017. The model's innovative architecture, including this type of feedforward network, revolutionized the field of natural language processing (NLP).[29]

2.3.3.5 BERT

1.Introduction to BERT

In recent years, deep learning has brought tremendous changes in natural language processing. Among such great innovations is BERT, by Google, that introduced this powerful bidirectional training mechanism within the Transformer construct. BERT's construction is able to capture deeper contextual relationships in text and set new state-of-the-art benchmarks across the different NLP tasks. The general-purpose nature of BERT allows it to be easily fine-tuned, making it well-suited for SQL injection detection, where subtle semantic patterns are crucial.

2.BERT Architecture Overview

The BERT model architecture is a multilayer bidirectional Transformer encoder, based on the encoder component of the original Transformer model. While the original Transformer use an encoder-decoder structure where the encoder processes input sequences using self-attention mechanisms and the decoder combines self-attention with encoder-decoder attention, BERT focuses exclusively on the encoder to learn deep contextual representations.

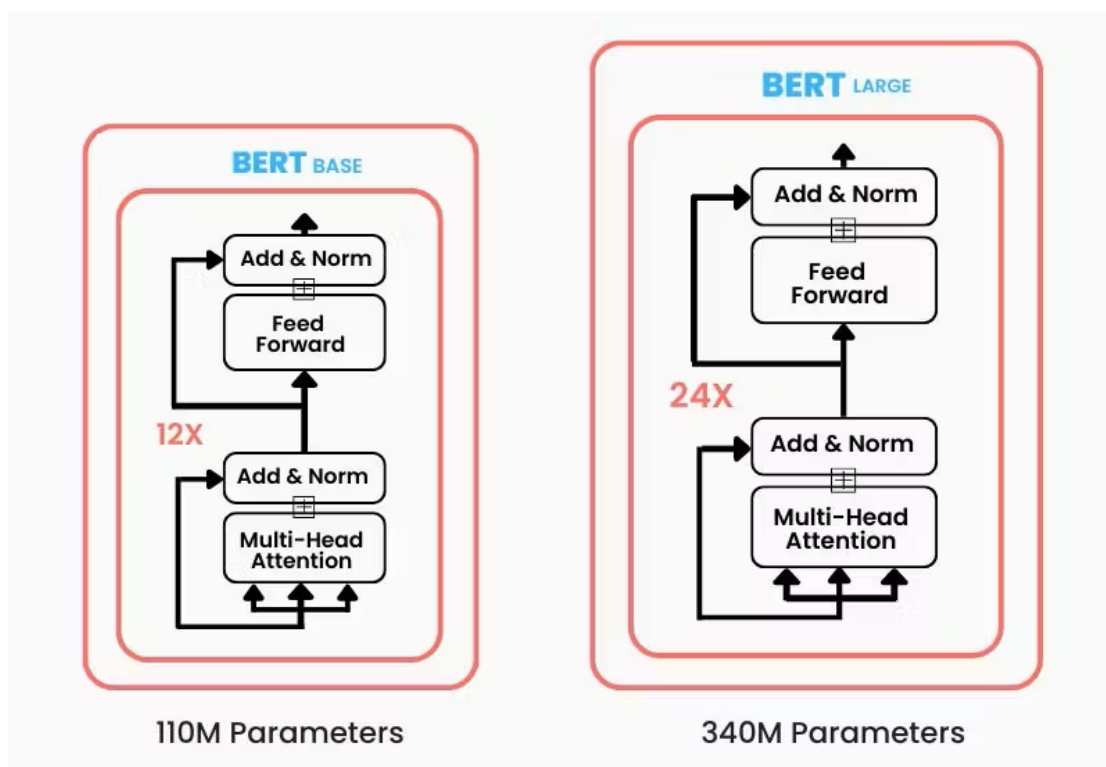


Figure 2.23 Main BERT Models

The BERT architecture is available in two main configurations: BERTBASE and BERTLARGE. BERTBASE consists of 12 encoder layers(Transformer blocks), 768 hidden units per layer, and 12 attentions heads, resulting in approximately 110 million parameters. In contrast, BERTLARGE extends the architecture to 24 encoder layers(Transformer blocks), 1024 hidden units, and 16 attentions heads, totaling around 340 million parameters.[30]

BERT-Base's encoder uses the standard Transformer encoder design. It is a stack of 12 encoder layers. Each layer contains two sub-layers, first a multi-head self-attention module and second a position-wise feed-forward network. In line with the original Transformer, each sub-layer is wrapped with a residual (add) connection and then followed by layer normalization. Precisely, the output of each sub-layer is computed as

LayerNorm($x + \text{SubLayer}(x)$), where $\text{SubLayer}(x)$ is either the attention or feed-forward function. This architecture ensures that the output dimensionality in every layer matches the input dimensionality, so the representation size remains unchanged across all 12 layers.

As a result, BERT models are capable of learning more complex contextual representations from input sequences.

3 Pre-training BERT

In the pre-training stage, BERT is trained on a gigantic corpus of unlabeled text with two unsupervised tasks:

Masked Language Modeling (MLM):

Instead of predicting the next word in a sequence (as with typical language models), BERT sifts through and randomly masks some of the input tokens and subsequently learns to predict the masked tokens from the context in which they appear. For example, given the sentence "The SQL [MASK] is malicious," the model will learn the missing word is likely "query."

Next Sentence Prediction (NSP):

This task is forecasting whether a specific sentence B has a tendency to follow sentence A naturally in the original text. This helps in providing the model with an appreciation of correspondence between sets of sentences, and this is useful for question answering and sentence classification.

By learning from these two tasks, BERT learns a rich bidirectional language understanding, which allows it to learn nuanced semantic and syntactic relationships between words in text.

4 Fine-tuning

After the pre-training was accomplished, BERT-instance could be fine-tuned to perform specific downstream tasks, such as text classification, sentiment analysis, named entity recognition, or in this piece of work, SQL injection detection.

During fine-tuning:

- . BERT with weights from pre-training is loaded.
- . Then, a simple classification layer is added on top of the output of BERT (e.g., a single dense layer followed by a softmax or sigmoid activation).
- . The entire network, including the pre-trained layers, is then trained on the target dataset with labeled examples.

This step requires far less time and fewer labeled examples than would normally be required to train a deep model from scratch, because BERT provides a strong linguistic knowledge base to begin with.

5 Advantages and Limitations of BERT

BERT has revolutionized natural language understanding by enabling deep bidirectional context learning. It has set state-of-the-art performance on a wide range of tasks. However, it is computationally expensive, and fine-tuning may require careful hyperparameter tuning and preprocessing strategies. Furthermore, its effectiveness may decrease on tasks involving very short texts or domain-specific language if not properly adapted.

2.4 Related works

2.4.1 Introduction

SQL Injection remains one of the most persistent and potential threats to web application security. Over the last decade, researchers have exploited distinct and varying machine-learning and deep-learning approaches to automate the detection and mitigation of SQL injection attacks. This chapter is a synthesis of important works in the area, with special emphasis on advanced techniques that use artificial intelligence to enhance accuracy in detection, reduce false alarms, and, in some cases, help with prioritization and prevention of attacks in real-time.

2.4.2 Detection Approaches Using Machine Learning

Early works on SQLi detection focused on the classical machine learning paradigm, such as Support Vector Machines (SVM), Decision Trees, Naïve Bayes, and Logistic Regression. These methods generally required engaged features obtained from HTTP request content,

URL, or even SQL query structure. They showed decent performances but were constrained by the quality and view of the features extracted. Sometimes, a minor change in the attack pattern would deter these methods from detecting them.

With the appearance of a few hybrid methods merging rules with machine learning classifiers to improve detection, more flexibility and accuracy were introduced but confronted again with false positive and false negative alarms, especially concerning large-scale data and obfuscated attacks.

2.4.3 Deep Learning for SQL Injection Detection

Recent research has increasingly turned to deep learning to overcome the limitations of traditional models. Three notable contributions illustrate the evolution and diversification of deep learning-based detection approaches.

2.4.3.1 NLP and BERT-Based Detection [31]

Lakhani et al. proposed an NLP approach to SQLi detection. Their model uses BERT or Bidirectional Encoder Representations from Transformers for contextual feature extraction. BERT was fine-tuned on a labeled dataset of SQL queries and indicated good performance with 97% accuracy and 0.8% false positives, and 5.8% false negatives.

The study argues that traditional code-level prevention mechanisms are insufficient and suggests that NLP-based approaches, such as BERT, offer adaptable detection capabilities for various SQLi variants.

2.4.3.2 MLP vs. LSTM Comparative Study [32]

Tang et al. did a successful study by comparing Multi-Layer Perceptron and Long Short-Term Memory networks for exploiting SQL injections in real ISP traffic data. The study involved feature extraction of eight handcrafted features from URL payloads: number of keywords, number of special characters, length of the payload, and so on.

The MLP with three hidden layers gave excellent results, with an accuracy of 99.67%, precision of 100%, and recall of 99.41%. On the other hand, LSTM, on its promise in sequential learning, gave accuracy results of 97.68% with heavy training time, thus being inefficient in this task. It was concluded that the feature-rich MLPs were much more efficient in this task, but the LSTM had potential in more complex scenarios.

2.4.3.3 CNN-LSTM Hybrid with Risk Prioritization [33]

Alan Paul et al. proposed an all-encompassing framework, i.e., SQLR34P3R, which puts the problem of SQL injection detection into a multi-class classification setting. The system detects the SQLi, prioritizes the attacks, and aids the prevention strategies. The system consists of a CNN-LSTM hybrid model, which is trained on a massive dataset of over 520,000 samples collected from the web and network traffic, attaining an average f-score of 97%.

Unlike previous approaches, SQLR34P3R performs contextual risk assessment based on known CVE vulnerabilities and operates its detection engine in real time, catching traffic from platforms such as DVWA and Vulntrado. The work is unique in that it combines threat intelligence with deep learning for a comprehensive and usable solution.

2.4.4 Comparative Discussion

These three contributions reflect the maturation of SQL injection detection research:

- **Lakhani et al.** demonstrated how pre-trained language models can improve detection with minimal feature engineering.
- **Tang et al.** emphasized the efficiency of classical MLPs combined with statistical features from real traffic.
- **Alan Paul et al.** introduced a real-time system combining deep learning with vulnerability assessment.

While MLPs are computationally light and achieve high accuracy, models like BERT or CNN-LSTM provide deeper semantic understanding and adaptability to novel threats. The choice between these methods depends on the deployment context: lightweight detection at the edge (MLP), semantic understanding (BERT), or full-stack risk management (CNN-LSTM).

2.4.5 Summary

In sum, these works provide promising evidence of how deep learning has been effectively used to detect SQL injection attacks with high precision and recall. In addition, they show that the combination of data-driven approaches and domain knowledge (CVE, traffic patterns) can help maximize performance and applicability. The following chapter details the implementation and experimental evaluation of our own SQLi detection models, including both traditional and transformer-based models.

2.5 Conclusion

This chapter explored the fundamentals of machine learning and deep learning, examining their core concepts, key algorithms, and architectural models. These technologies are the cornerstone of modern-day artificial intelligence, in the sense that AI systems learn from data and take decisions based on need. An understanding of these frameworks will thus allow a reader to understand how advanced models such as BERT are operationalized for complex tasks such as SQL injection detection. We shall now use this understanding to study deep learning model implementation in practical cybersecurity scenarios.

Chapter 3

Conception and Implementation

3.1 Introduction

This chapter describes the models' design and implementation for SQL injection detection. Different architectures evaluated in our work include the traditional machine learning and deep learning approaches. The rationale behind this approach is to assess and compare the models' performances on the same dataset in order to find the one most suitable for accurately detecting SQL injection attempts. The next sections describe the dataset employed, as well as the preprocessing done on it, followed by the design for each of the models and the evaluation strategy considered for training and testing.

3.2. Dataset

To train an effective deep learning model successfully, we need to make sure the dataset is properly chosen. For our project, we needed a dataset that includes samples classified into two categories: queries containing SQL injection and those that do not. Using this dataset, the trained model will be able to detect whether a query is a SQL injection or a normal query.

After conducting research on Kaggle, we found a collection of "SQL Injection Datasets" on Kaggle. Among these datasets, we used a collection of datasets that put together by a person named "Syed Saqlain Hussain Shah" and it contain three (3) datasets.

SQLi.csv contains 3951 samples with 78% classified as normal queries and 28% as malicious queries.

SQLiV2.csv contains 33726 samples with 66% classified as normal queries and 34% as malicious queries.

SQLiV3.csv contains 30873 samples with 62% classified as normal queries and 37% as malicious queries and 1% as other.

After training and testing multiple machine learning and deep learning models on the three datasets, we found SQLiV3.CSV to be the most suitable dataset for our project. Models trained on this dataset were better able to detect SQL injection attacks and benign queries compared to other datasets, SQLiV3.csv consistently led to a higher accuracy and better generalization across different models, making it the optimal choice for our final implementation.

DATA preprocessing

Before training the models, the SQLiV3.csv dataset needs to be preprocessed as follows:

-We found two trailing commas (,,) at the end of all the lines in the CSV file and removed these unnecessary characters using a Python script.

- Filters the rows to keep only those that contain exactly 2 columns and removes empty rows.

By applying the above preprocessing steps, we ultimately created a partitioned dataset containing:

- **30,614 SQL queries.**
- Each entry consists of a **Sentence** (the SQL query) and a **Label** (0 for benign, 1 for SQL Injection).

Label distribution :

- **Normal (Label = 0) :** 19,268 queries
- **Malicious (Label = 1) :** 11,346 queries

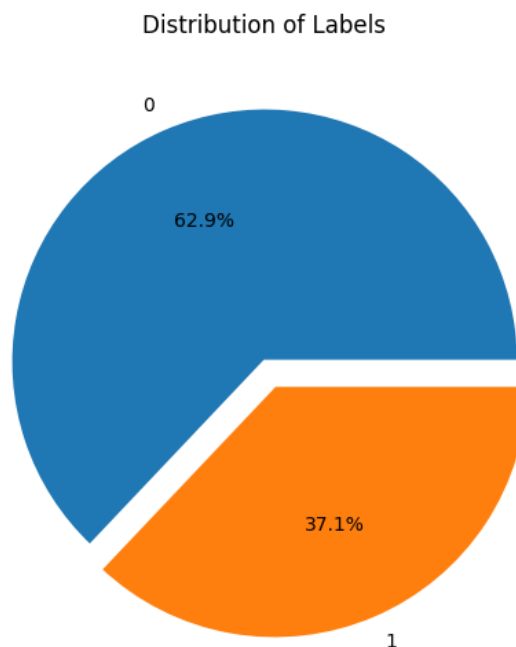


Figure 3.1 Label Distribution of The Dataset

3.3 Development Environment Overview

3.3.1 Programming language

Python

Python is widely used in machine learning and deep learning, and for good reason. Its clean and simple syntax makes it easy to learn and work with, which is especially helpful when building and testing complex models. A key strong point of Python is its huge selection of

libraries, which include TensorFlow, PyTorch, Scikit-learn, and Keras. These libraries offer pre-written code and functions that save time and make the development process easier. Python also has a huge community of developers and researchers, which means there's plenty of documentation, tutorials, and support available online. Python offers simplicity, flexibility, and strong tools, all of which render it an optimal language for machine learning and deep learning projects.

3.3.2 Libraries Used

Numpy: The fundamental library for numerical computation in Python, NumPy, handles arrays, mathematical operations, and prediction outputs such as probability arrays and class labels.

Pandas: The pandas library is efficient for data manipulation and data analysis. we used it in this study to import the CSV dataset, remove duplicate rows, handle missing data, and cast data into appropriate formats for training models

Chardet: Chardet library is a character encoding detection library, and we have used it in order to detect an appropriate encoding for our CSV file which could be read by pandas successfully.

Ktrain : A high-level TensorFlow/Keras library that facilitates easy training, testing, and interpretation of deep learning models. Consistent with this, we used it to transform data into BERT-compatible format, train a text classification model, and evaluate the performance of the model.

Matplotlib: matplotlib is a graph and chart creation library. We leveraged it to plot class distributions, training loss and validation loss, training accuracy and validation accuracy, and the confusion matrix.

3.3.3 Development setup

3.3.3.1 Visual Studio Code

Visual Studio Code (VS Code) is free, open-source software meant for coding, developed by Microsoft. Its support many programming languages such as Python, C++, JavaScript, etc. Equipped with syntax highlighting, code completion (IntelliSense), a debugging tools, and a customizable user interface, and an integrated terminal.

VS Code is very extensible and its capability can, therefore, be enhanced by any user with thousands of extensions that support frameworks, debugger.

It has support for Markdown to style text, allowing users to document alongside their code. With its lightweight might combined with powerful development features, Visual Studio Code is among the most popular editors for web developers, data scientists, DevOps engineers, and others.

3.3.3.2 Jupyter Notebook

The Jupyter Notebook is an open-source, browser-based interactive environment that offers users the ability to create and share documents that contain live code, equations, visualizations, and narrative text. Data scientists, machine learning specialists and scientific researchers are massive user bases of Jupyter Notebooks. They are used for easy experimentation, reproducibility, and collaborative purposes. A notebook is divided into cells, cells can be individually run. Having such a structure makes the Jupyter Notebook very handy for stepwise analysis, prototyping, and reporting. The format enhances understanding by combining code with explanations and visual output.

3.3.3.3 Google Colab

Google Colaboratory (or Colab) is a free cloud service that allows users to write and execute Python in Jupyter Notebook environment without the necessity of local setup. Supporting machine learning, deep learning, data analysis, and research workflows. Colab offers access to computing resources, i.e., CPUs, GPUs, and TPUs, right within the browser. It is majorly used for prototyping, or collaborative development, built-in support for Google Drive and support for major Python libraries such as TensorFlow, PyTorch, and Pandas.

For our work, we used The NVIDIA Tesla T4. It is a powerful energy-efficient GPU designed for a wide range of AI workloads. It was built over the Turing architecture and includes Tensor Cores to accelerate deep learning compute and RT Cores for ray tracing. The T4 GPU comes with 16 GB of GDDR6 memory and supports mixed precision computing. It also delivers up to 260 TOPS (Tera Operations per Second) for INT8 inference. It is mostly used in cloud environments (like Google Colab) for efficiently accelerated machine learning workflows.

3.4 Models Implemented

This section on models presents the various models that have been developed for detecting SQL injections. A number of machine learning models and deep learning models were trained and evaluated with varying architecture and hyperparameters. The aim is to see how traditional techniques and modern, state-of-the-art techniques differ in input to achieve results on the same dataset.

For each model, we describe the structure, the main hyperparameters used during training, and the results obtained. These models are as follows:

Support Vector Machine (SVM)

Logistic Regression (LR)

Multilayer Perceptron (MLP)

Recurrent Neural Network (RNN)

Long Short-Term Memory (LSTM)

Bidirectional Encoder Representations from Transformers (BERT)

3.4.1 Support Vector Machine (SVM)

We used a Support Vector Machine with a linear kernel (`kernel='linear'`) and regularization parameter $C = 0.1$. The input queries were transformed into vectors by applying TF-IDF with 3000 features as the maximum. We dropped duplicates and split the dataset into 80% training and 20% testing sets prior to model training.



```
model = SVC(kernel='linear', C=0.1 ,probability= True)
model.fit(X_train,y_train)
y_pred = model.predict(X_test)
```

3.4.2 Logistic Regression (LR)

We trained a Logistic Regression classifier to detect SQL injection payloads using TF-IDF features (maximum of 3000 features). The classifier was initialized with `solver='liblinear'` and `penalty='l1'`. After removing duplicate queries, the dataset was divided into 80% for training and 20% for testing.



```
lrc = LogisticRegression(solver='liblinear', penalty='l1')
lrc.fit(X_train,y_train)
y_test_pred = lrc.predict(X_test)
```

3.4.3 Multilayer Perceptron (MLP)

We trained a Multi-Layer Perceptron (MLP) neural network Using ReLU activation functions with three hidden layers comprising 512, 256, and 128 units respectively. For binary classification, the output layer employed a sigmoid activation function. Using TF-IDF with a maximum of 3000 characteristics, input queries were vectorized.

Using the binary cross-entropy loss function and the SGD optimizer with a learning rate of 0.01, the model was compiled. Using 20% of the training set for validation, it was trained for 18 epochs on 80% of the data with early stopping activated (`patience = 3`). To guarantee clean input data, duplicates were deleted from the test set prior to evaluation.

Build and Compile the Model


```
model = Sequential([
    Dense(512, input_dim=3000, activation='relu'),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])
sgd = optimizers.SGD(learning_rate=0.01)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=
['accuracy'])
```

Train with Early Stopping

```
early_stop = EarlyStopping(patience=3, restore_best_weights=True)
history = model.fit(X_train, y_train,
                    epochs=27,
                    batch_size=32,
                    verbose=2,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stop])
```

3.4.4 Recurrent Neural Network (RNN)

We trained a Recurrent Neural Network to detect sequential patterns in SQL injection attempts by feeding in tokenized and padded input queries. Character filters were disabled, and case sensitivity was preserved during preprocessing to maintain the SQL syntax and structure.

The first layer of the model architecture was an Embedding layer of 128 dimensions, followed by a SimpleRNN layer of 128 units. To avoid overfitting, we put in a Dropout layer at 0.5. Then a Dense layer of 64 units was added with ReLU activation and L2 regularization ($\lambda = 0.01$). The output layer has a sigmoid activation for binary classification.

The model was compiled using binary crossentropy as the loss function and the Adam optimizer with a learning rate of 0.0001. The model was trained for 20 epochs with early stopping at 3 epochs' patience, and the best weights from validation were restored.

Build and Compile the Model

```
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),
    SimpleRNN(128, return_sequences=False),
    Dropout(0.5),
    Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    Dense(1, activation='sigmoid')
])
optimizer = optimizers.Adam(learning_rate=0.0001)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=
['accuracy'])
```

Train with Early Stopping

```
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=[early_stop]
)
```

3.4.5 Long Short-Term Memory (LSTM)

We have trained an LSTM network to recognize sequential patterns in SQL injection attempts by feeding in tokenized and padded sequences. Character filters were disabled and case sensitivity was preserved during preprocessing to preserve the integrity of the SQL syntax.

The model architecture consisted of a 256-dimensional embedding layer, followed by two stacked LSTM layers with 256 and 128 units respectively. This was followed by a fully connected Dense layer of 64 units with ReLU activation, and a sigmoid-activated output layer for binary classification.

Training was carried out for 30 epochs using the Adam optimizer, a learning rate of 0.001, and the binary cross-entropy loss function to minimize. Early stopping with patience of 3 epochs was employed to prevent overfitting, and the best model weights were restored based on validation performance.

Build and Compile the Model

```
embedding_dim = 128
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_len),
    LSTM(256, return_sequences=True, kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.3),
    LSTM(128, kernel_regularizer=regularizers.l2(0.001)),
    Dropout(0.3),
    Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

Train with Early Stopping

```
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=[early_stop]
)
```

While several classical and deep learning models were implemented (see Sections 3.4.1 to 3.4.5), particular emphasis is placed on the BERT model due to its recent state-of-the-art performance and its central role in our contribution. Therefore, Section 3.4.6 provides a more detailed explanation of its implementation.

3.4.6 BERT

3.4.6.1 Why BERT for SQL Injection Detection

We selected BERT as our SQL injection detection model due to its high capacity to comprehend both meaning and context of textual information, particularly in structured input such as SQL queries. What makes BERT distinctively effective is the fact that it processes the whole input bidirectionally which helps it pick up on subtle patterns that might indicate an attack. Unlike older models that generally rely on basic keyword matching or predefined rules, BERT learns from the actual composition and intent of the query. After fine-tuning on a database consisting of normal and malicious SQL queries, the model was able to accurately identify suspicious inputs that could potentially represent injection attempts. Thus, BERT proves to be a reliable and effective tool for web application security improvement.

We decided to use the BERTBASE model for our implementation mainly because our dataset contains 30,614 labeled queries. Since BERTBASE is less complex than BERTLARGE, it trains faster and uses fewer resources, while still performing well for classification tasks like detecting SQL injection attacks.

3.4.6.2 BERT Code and Implementation

In this section, we present the code and detailed implementation of our SQL injection detection model, which is based on the BERT architecture.

Reading and displaying the dataset

```
with open('SQLiV3_cleaned2.csv', 'rb') as f:
    result = chardet.detect(f.read())
df = pd.read_csv('SQLiV3_cleaned2.csv', sep=',', encoding=result['encoding'])
print(df)
```

This code detects the character encoding of the CSV file SQLiV3_cleaned2.csv using the chardet library and then reads the file into a pandas DataFrame (df) with the correct encoding. This ensures the file is read without encoding errors, especially if it's not in UTF-8 format.

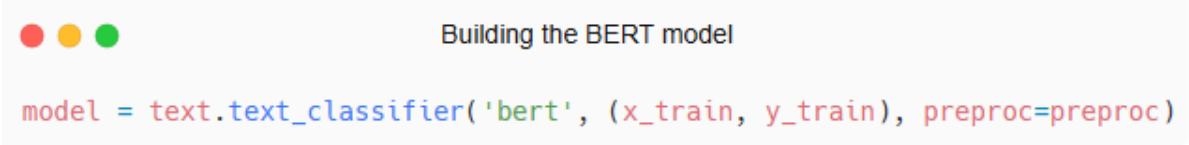
Creating the training and test sets

```
sentences = df['Sentence'].tolist()
labels = df['Label'].tolist()
(x_train, y_train), (x_test, y_test), preproc = text.texts_from_array(sentences, labels,
                                                                      preprocess_mode='bert',
                                                                      maxlen=100,
                                                                      val_pct=0.2,
                                                                      class_names=list(set(labels))
                                                                      )
```

In this code, the text data (Sentence column) and the corresponding label data (Label column) are extracted from the DataFrame and converted to lists. Afterward, the `texts_from_array` function of `ktrain` is used to do the preprocessing of these texts so that they can be used for fine-tuning the model. With `preprocess_mode='bert'`, the word text is passed through a BERT tokenizer that lowercases, tokenizes into word pieces, pads, and then sequences exceeding 100 tokens are automatically shortened. It is intentionally split into training and validation (with 20% of samples used for validation) sets (`val_pct=0.2`), the `class_names` parameter specifies the two label classes allowing the function to handle the binary classification task of SQL

injection detection. Then, the function returns the preprocessed train and validation sets (`x_train`, `y_train`, `x_test`, `y_test`), while the `preproc` object contains the tokenizer and configurations used during the preprocessing stage.

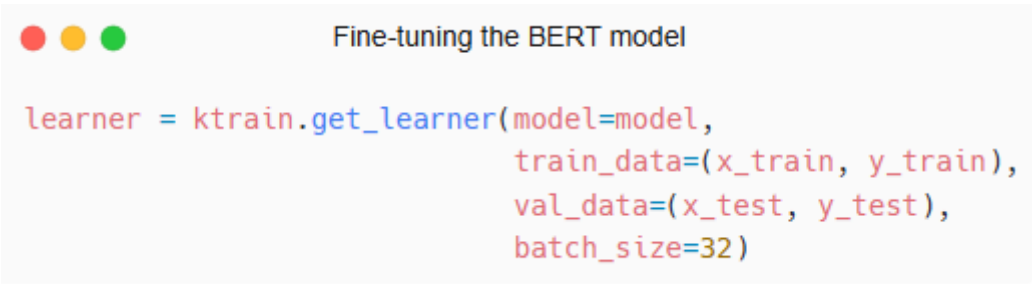
Building the model



```
model = text.text_classifier('bert', (x_train, y_train), preproc=preproc)
```

This code initializes a text classification model using the BERT architecture with the preprocessed training data (`x_train`, `y_train`) and the associated preprocessing configuration `preproc`. It sets up the model ready for fine-tuning on our dataset.

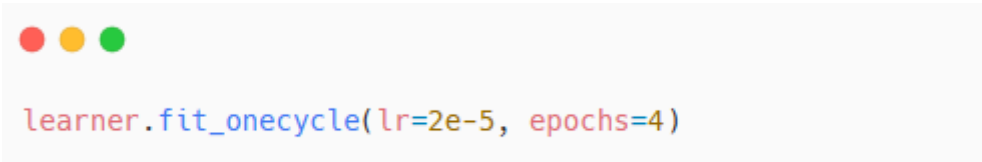
Fine-tuning the BERT model



```
learner = ktrain.get_learner(model=model,
                             train_data=(x_train, y_train),
                             val_data=(x_test, y_test),
                             batch_size=32)
```

The `ktrain.get_learner` function creates a learner object that combines the model and training procedure. The `model` argument takes the pre-trained BERT model that we want to fine-tune. The training data (`train_data`) is passed as a tuple containing the input features (`x_train`) and their corresponding labels (`y_train`). Validation data (`val_data`), given as a tuple, then gets their inputs and labels (`x_test` and `y_test`) used to measure performance of the model during training.

The `batch_size=32` means that the training data will be divided into mini-batches of 32 samples, which helps with efficient gradient updates and memory usage. Returned as the learner object, it can now hold the model and data so it is ready to be fine-tuned and validated and later used for prediction purposes.



```
learner.fit_onecycle(lr=2e-5, epochs=4)
```

This method of `fit_onecycle` is used to fine-tune the pretrained model. The learning rate (`lr=2e-5`) controls the update of the model weights and has been made very low for stable fine-tuning. The number of epochs is 4, meaning that the training dataset is passed through the model four times.

The One Cycle learning rate policy applied by this method slowly increases the learning rate

from a very low value to max value and then brings it down to its lowest rate, helping to improve training speed and overall model performance.

3.5 Summary of Model Architectures

To summarize the architecture and training settings of all implemented models, including BERT, we present in the following tables a comparative overview of the key hyperparameters and configurations.

Model	Preprocessing	Architecture	Hyperparameters
SVM	- TF-IDF (max 3000 features) - Deduplication	- Linear kernel	- C=0.1
LR	- TF-IDF (max 3000 features) - Deduplication	- Linear kernel	- penalty='l1' - solver='liblinear'
MLP	- TF-IDF (max 3000 features) - Deduplication	- 512-256-128 (ReLU/Sigmoid)	- learning_rate=0.01
RNN	- Tokenization + padding - Lowercasing - Deduplication	- SimpleRNN(128) + Embedding(128)	- learning_rate=0.0001
LSTM	- Tokenization + padding - Lowercasing - Deduplication	- Embedding(256) + LSTM(256 → 128)	- learning_rate=0.001
BERT	- BERT tokenizer - Lowercasing - Deduplication	- BERTBASE (12 layers)	- lr=2e-5 (OneCycle) - maxlen =100 - batch_size = 32

Table 3.1 Overview of HyperParameters and Architectures Used in Model Implementation

As shown in Table 3.1, traditional models like SVM and Logistic Regression were paired with TF-IDF vectorization and simple linear kernels, allowing for efficient training and interpretability. The regularization parameters (C for SVM, L1 penalty for LR) were chosen to balance generalization and sparsity.

For neural networks, the MLP was designed with three dense layers that progressively shrink to capture nonlinear patterns in TF-IDF features, whereas RNN and LSTM models processed tokenized input sequences that preserve the syntactic structure of queries. RNN architecture remains relatively shallow so as to prevent overfitting, whereas LSTM is stacked with memory units that capture long-term dependencies.

BERT, on the other hand, is the language model under the transformers trained on big corpora. Fine-tuning with a small learning rate of $2e-5$ and an OneCycleLR scheduler guarantees the achievement of stable convergence. The preprocessing step—truncation and lowercasing—keeps in line with the pretraining settings of BERTBASE.

The adjustments on architecture and preprocessing were made so that a fair comparison can be effected across the various learning paradigms, adapting each model to its strong points in text classification.

Model	Optimization	Regularization	Epochs
SVM	- N/A (solver)	- Implicit L2	N/A
LR	-N/A (solver)	-L1 penalty	N/A
MLP	- SGD + BCE	-Early Stopping (patience=3)	27
RNN	- Adam + BCE	-Dropout(0.5) + L2($\lambda=0.01$) -Early Stopping (patience=3)	20
LSTM	-Adam + BCE	-Dropout(0.3) + L2($\lambda=0.001$) -Early Stopping (patience=3)	30
BERT	- AdamW + BCE	- Built-in dropout	4

Table 3.2 Optimization Techniques, Regularization Methods, and Training Epochs Used per Model

Each model was configured and hyper parametrized by several configurations and hyperparameters to ensure they reached their maximal performance. The general settings outlined in the above table are essentially the best combinations available for the experiments carried out. Different learning rates, optimizers, regularizers, and epochs were observed with respect to their behavior during training. For example, with recurrent models like RNN and LSTM, there were issues with vanishing gradient; hence, hidden units were tuned, dropout layers employed, and early stopping used. Also, the epochs needed control not to lead to overfitting, which was more critical for deeper architectures. This final configuration, including using AdamW for BERT, dropout for RNNs, and early stopping for MLP and LSTM, consistently showed better performance than alternatives. They provide a compromise between the model's complexity, its rate of training, and its capacity to generalize, thus standing as the best of all configurations we tried.

3.6 Conclusion

In this chapter, we described the building and preparing processes for the various SQL injection detection models. We gave the dataset, how it was cleaned and processed, the model structure, and the parameters used in training. The work also offered some code examples and comparison tables to keep things clear.

Testing of the models and analyzing results follow in this; therefore, this work was laying the foundation for the next step. In Chapter 4, we will assess the performance of these models with test data and on fresh unseen data as well: normal and attack. We will compare the results to know which one is the best.

Chapter 4

Performance Evaluation and Results

4.1 Introduction

In this chapter, we present the evaluation of our SQL injection detection models. After building, training and fine-tuning the models in the previous chapter, we now assess their performance using different metrics. The evaluation is done on both the test dataset and unseen data to check how well each model generalizes. We also compare our results with other existing works and provide explanations for the performance differences.

4.2 Performance Evaluation Metrics

The classification performance of the fine-tuned BERT model in detecting SQL injection attacks was evaluated with the help of multiple performance evaluation metrics. They include accuracy, precision, recall, and F1-score, which help to analyze the effectiveness of the model detecting malicious and benign SQL queries.

Confusion Marix

A confusion matrix is a table that summarizes the performance of a classification model by comparing its predicted labels to the true labels. It displays the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) of the model's predictions.[34]

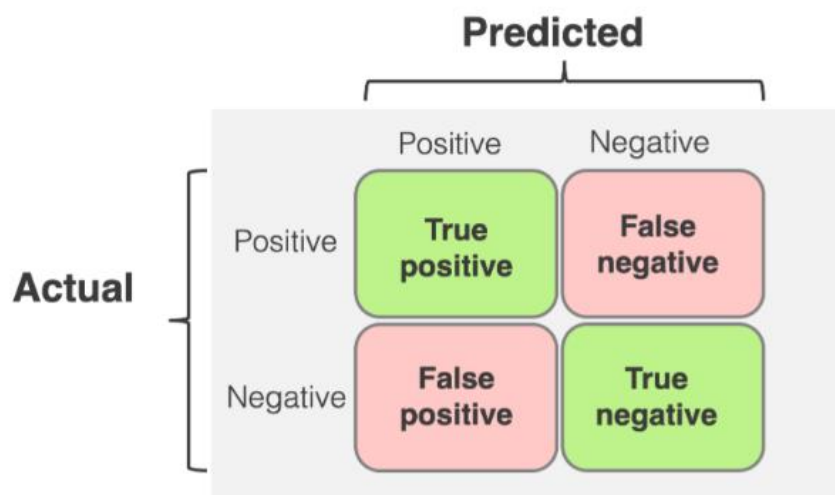


Figure 4.1 Confusion Marix

True Positive (TP) : It shows the number of correctly identified positive cases.

True Negative (TN) : It shows the number of correctly identified negative cases.

False Positive (FP) : It shows the number of incorrectly predicted positive cases.

False Negative (FN) : It shows the number of incorrectly predicted negative cases.

Accuracy:

Accuracy is a fundamental metric for evaluating the performance of a classification model, providing a quick snapshot of how well the model is performing in terms of correct predictions.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

Precision:

Precision is a measure of a model's performance that tells you how many of the positive predictions made by the model are actually correct. This metric reflects the model's ability to avoid false positives, high precision means that when the model predicts "positive," it is usually correct.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall:

It measures how well the model captures all relevant positive cases. This measures the model's ability to capture positive instances, high recall means most true positives are detected by the classifier.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-score:

The F1-score is metric that balances both precision and recall. This metric usually tells us how precise (correctly classifies true positives) and robust (minimizes false negatives) our classifier is. The more the F1 score better will be performance.

$$\text{F1 Score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.3 Evaluation on Test Set (20%)

4.3.1 Results Presentation

In this section, we present the performance results of the proposed models evaluated on the test dataset, which represents 20% of the original data. We report common classification metrics such as Accuracy, Precision, Recall, and F1-score to assess each model's ability to correctly detect SQL injection attempts.

Model	Accuracy	Precision	Recall	F1 Score
SVM	98.56%	99.86 %	96.25%	99.02%
LR	98.12%	99.72%	95.25%	97.44%
MLP	99.44%	99.51%	98.98%	99.25%
RNN	99.06%	100%	97.51%	98.74%
LSTM	99.62%	99.73%	99.25%	99.49%
BERT	99.92%	100%	99.78%	99.89%

Table 4.1 Performance Comparison of Models on SQL Injection Dataset

The experimental results demonstrate outstanding performance across all evaluation metrics. Traditional machine learning models such as Support Vector Machine (SVM) and Logistic Regression are strong contenders, for instance, the SVM model got an accuracy of 98.56% and an F1-score of 99.02%. Contrary to popular belief, which states that SVM accuracy tends to saturate on complex tasks, our model indeed showed fine generalization ability along with confirmation of training and validation curves.

The deep learning models; the MLP and RNN also enhanced the system. LSTM had slightly higher performance: 99.62% accuracy and 99.49% F1-score. But BERT was the strongest performer, with 99.92% accuracy, 100% Precision and 99.78% recall-an indication that it is unbeatable when it comes to detecting SQL injections from very complex types of input patterns.

Beyond the ability of any model, the dataset was a big factor for such astounding performances, having been cleaned and preprocessed to minimize noise and ensure higher consistency of training data. Extensive hyperparameter tuning was done for every model to extract the maximum out of the training, including trying out numerous configurations before finally settling on the best.

These results demonstrate the effectiveness of using deep learning, especially Transformer-based models, for SQL injection detection. The next section provides additional visualizations

(learning curves and confusion matrices) to support these findings and illustrate the training behavior of selected models.

Comparative Analysis : Proposed Models vs. Existing Work

To evaluate our models, we explored similar works on Kaggle, GitHub, and in published articles. We compared our results with existing machine learning and deep learning models applied to the same dataset. This comparative analysis highlights how our models outperform previous approaches in terms of accuracy, precision, and F1-score, especially with a more complete use of the dataset.

We found several works on Kaggle that applied both machine learning and deep learning models to similar problems using the same dataset.

[35] : Created by **Hassan Bechara**, this Kaggle notebook applies Support Vector Machine (SVM) and Logistic Regression (LR) models on the same dataset we used. However, it only uses 3,941 samples, which we consider insufficient for a reliable evaluation.

[36] : Created by **Aman Rajput**, this Kaggle notebook also implements SVM and Logistic Regression models using nearly the same dataset and over 30,000 samples, closely matching our experimental setup. It served as a strong comparative baseline for our work.

[37]: Created by Syed Saqlain Hussain Shah, the same author who originally published the dataset we used. In this notebook, he applied a Multi-Layer Perceptron (MLP) model but used only 4,200 samples from the dataset, which we consider too limited to fully explore its potential.

[38] : Created by Saumya Verma, this Kaggle notebook also implements a Multi-Layer Perceptron (MLP) model using the same dataset. It use only 4,200 samples, which limits the depth of evaluation and generalization.

[39] : Created by Derara Duba, this GitHub project implements an LSTM model using the same dataset and the same number of samples as our work (30,907). It provides a relevant deep learning baseline for performance comparison.

Source	Model	Samples Used	Accuracy	Precision	F1 Score
Our Work	SVM	30,602	98.56%	99.86%	99.02%
[35]	SVM	3,941	90.87%	94.5%	86%
[36]	SVM	30,920	80.83%	N/A	36.86%
Our Work	LR	30,602	98.12%	99.72%	97.44%
[35]	LR	3,941	90.74%	93%	86%
[36]	LR	30,920	93.69%	N/A	85.55%
Our Work	MLP	30,602	99.44%	99.51%	99.25%
[37]	MLP	4,200	97.74%	92.99%	N/A
[38]	MLP	4,200	97.98%	94.01%	N/A
Our Work	LSTM	30,602	99.62%	99.73%	99.49%
[39]	LSTM	30,907	96.42%	99.62%	96.34%

Table 4.2 Proposed Models vs. Existing Work

4.3.2 Visual Performance Analysis

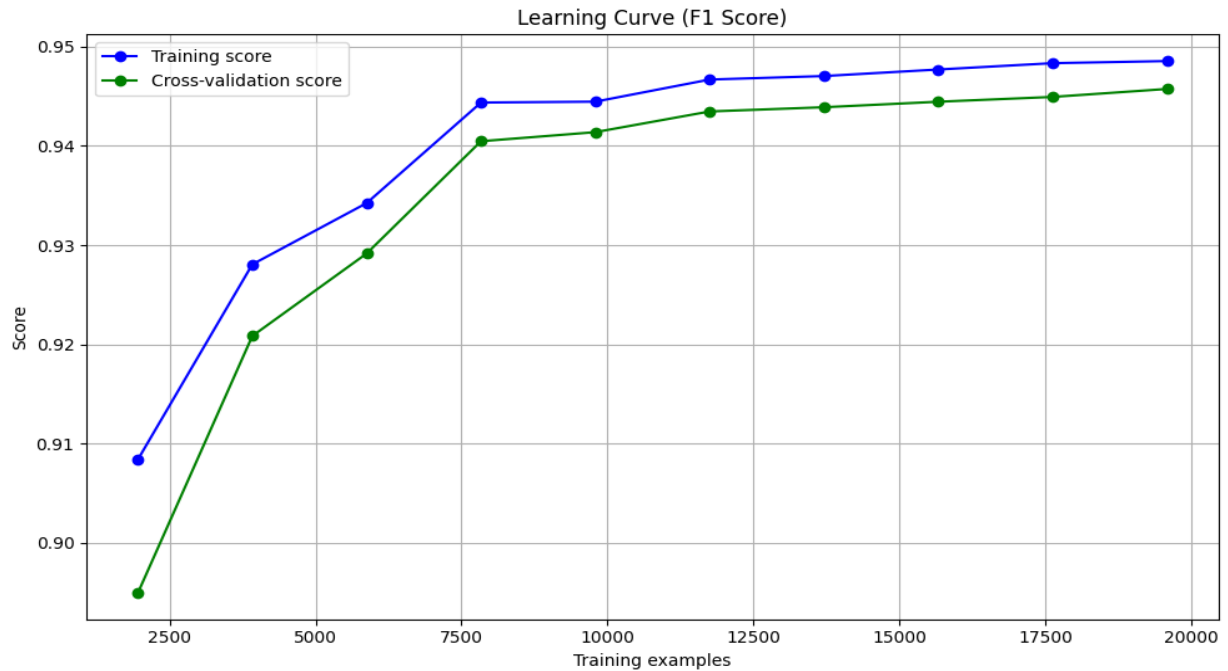


Figure 4.2 F1-Score Across Cross-Validation Folds for the SVM Model

For the SVM model, additional cross-validation was performed. The F1-score remained consistently high across folds, and the training/cross-validation curve shows a stable gap, confirming that the model does not overfit and generalizes well.

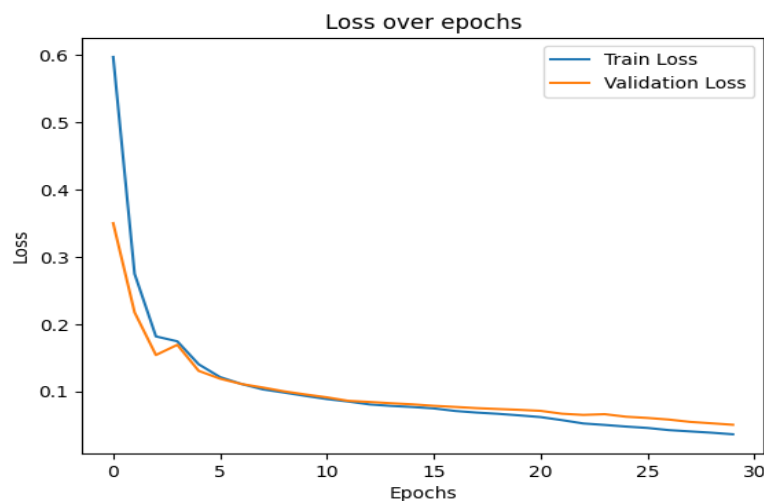


Figure 4.3 LSTM Training and Validation Loss over Epochs

This figure illustrates the training and validation loss progression for the LSTM model. The close alignment of both curves and the continuous decrease in loss demonstrate stable learning and the absence of overfitting.

4.3.3 Performance Analysis of BERT

The results obtained reflect the model's ability to correctly classify instances and demonstrate its effectiveness in detecting SQL injection queries benign queries.

4.4.3.1 Training Loss Curve

We show the training loss curve to better understand how the BERT model learned during training. The loss keeps going down steadily, which means the model is learning well without problems like overfitting. This shows that the training went smoothly and the model works well.

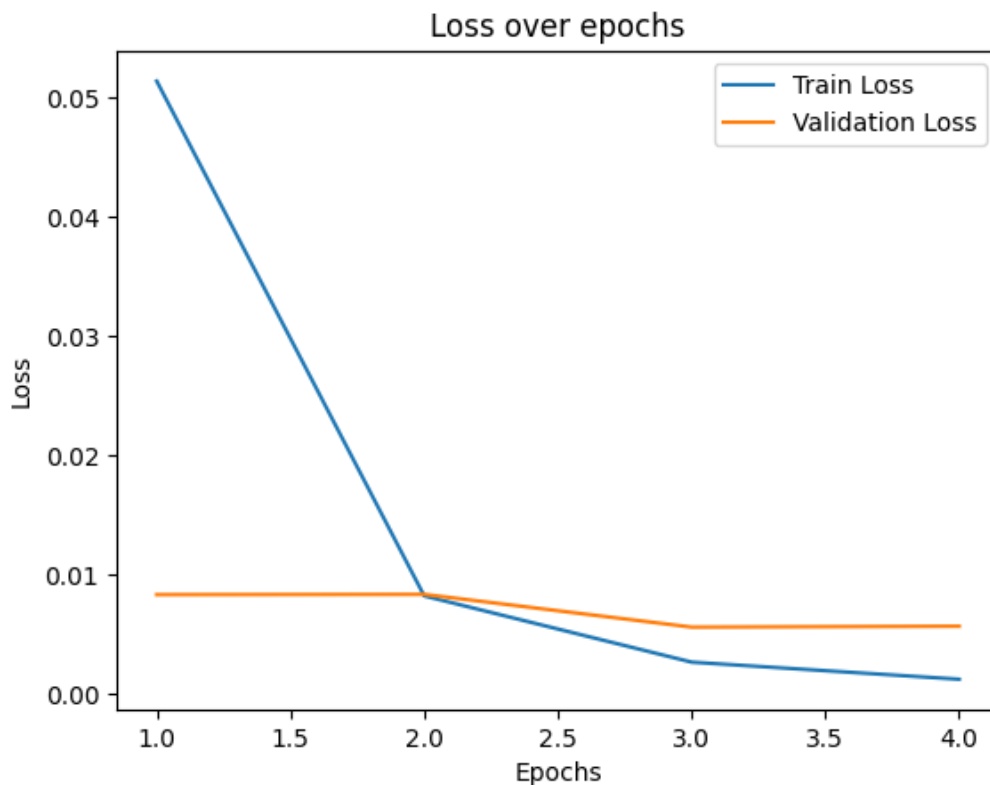


Figure 4.4 Training Loss Curve for the Fine-Tuned BERT Model

4.3.3.2 Model Evaluation on New Unseen Data

First, we created a predictor object that use the fine-tuned model and preprocessing pipeline to make predictions on new data:



```
predictor = ktrain.get_predictor(learner.model, preproc)
```

We then took a test dataset named `sqliv2_utf8.csv`. From this dataset, 500 examples were sampled for each class (label 0, label 1), a total of 1000 examples. The prediction results are summarized below:

For Label = 1:

The model correctly predicted all **500 samples as SQL injections**, reaching an **accuracy of 100%** for this class. This means that no positive instances were missed by the model.

For Label = 0:

Of the 500 Label 0 samples, the model predicted **499 were predicted correctly as benign queries**, and 1 was predicted incorrectly as a SQL injection. An accuracy of **99.8%** for the benign queries.

Overall Performance :

Below is the confusion matrix showing these results:

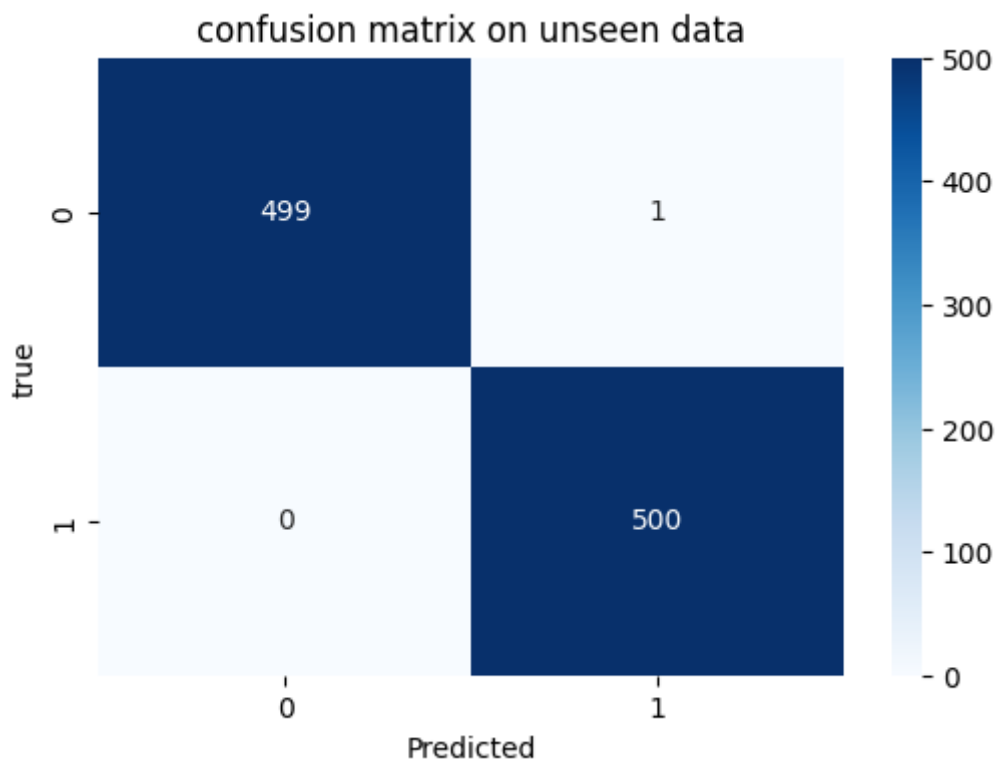


Figure 4.5 Confusion Matrix on Unseen Data

The confusion matrix offers a comprehensive overview of the model's ability to correctly classify each category on unseen data, demonstrating its ability to accurately distinguish between Class 0 (benign queries) and Class 1 (SQL injection) instances. The analysis shows exceptional performance across all standard evaluation metrics:

1. Accuracy (99.90%) :

With almost perfect accuracy, the model made 999 out of 1000 correct predictions. This exceptional performance suggests that the model has learned the underlying patterns in the data extremely well. That one misclassification accounts for just 0.1% error rate.

2. Precision (99.80%)

With a single false positive, this implies the precision value of 99.8%, which means that whenever the model predicted Class 1, it was almost surely correct. High precision is particularly valuable in scenarios where false positives carry significant costs.

3. Recall (100%)

The perfect recall score indicates the model successfully identifies all actual positive cases without any false negatives. This is crucial for applications where missing positive cases (Type II errors) could have serious consequences, such as detecting SQL injection attacks.

4. F1-Score (99.90%)

Almost perfectly balanced between precision and recall, the F1 indicates that there is a clear balance of these metrics with no trade-offs. This means that in terms of the model, one metric is not sacrificed for the other: achieving high precision and perfect recall at the same time.

Model	Accuracy	Precision	Recall	F1 Score
SVM	99,08%	99.76%	99,08%	99.02%
LR	97,44%	98.41%	97,44%	98,70%
MLP	98,99%	99.06%	98,99%	99,49%
RNN	98,85%	98.73%	98,85%	99,42%
LSTM	98,35%	98.96%	97.85%	99,17%
BERT	99.90%	99.80%	100%	99.90%

Table 4.3 Performance Comparison of Models on Unseen SQL Injection Dataset

Among all the models evaluated, BERT stands out with the highest accuracy of 99.90%, alongside with excellent precision and perfect recall, showing that BERT has a clear advantage over some traditional and neural approaches, however, with F1 scores of SVM, MLP, and RNN all above 99%, these three approaches also perform very well. Logistic Regression holds up surprisingly well, given its simplicity, LSTM has very decent results, but lags a little behind the others. Overall, while all models demonstrate strong performance, BERT has a slight edge, offering a more refined and consistent detection capability.

4.4 Conclusion

This chapter presented a detailed evaluation of various models for SQL injection detection. Traditional approaches such as SVM and Logistic Regression achieved strong results. However, deep learning methods, particularly LSTM and BERT, performed even better. BERT achieved outstanding performance, with an accuracy of 99.90% and a recall of 100% on the test set. It also showed excellent generalization on unseen data. These results confirm that transformer-based architectures are highly effective in accurately and reliably detecting SQL injection attacks.

General Conclusion

In this thesis, we studied the problem of SQL Injection attacks, which are among the most common and dangerous threats to web applications. We explored different approaches to detect these attacks using both traditional machine learning models and deep learning techniques.

Our experiments showed that classical models such as Logistic Regression and SVM gave good results when using proper text preprocessing and vectorization. However, deep learning models performed better. Among them, BERT, a pre-trained language model based on the Transformer architecture, achieved the best performance. It reached very high accuracy and recall, even when tested on new and unseen data.

These results confirm the strong potential of deep learning in the field of cybersecurity. BERT's ability to understand the context and meaning of queries makes it especially suited for detecting malicious SQL statements. Its success in this task shows that language models can play a key role in building more secure web applications.

Even if the results are promising, future work could involve testing the model in real-world environments, increasing the size and diversity of the dataset, and adapting the system to detect other types of attacks. This would make the solution more robust and useful in practice.

To conclude, this work proves that modern deep learning models like BERT are powerful tools for improving web security. They offer a high level of precision and can help in building smarter and more adaptive defense systems against SQL Injection attacks

References

- [1] OWASP, “SQL Injection (SQLI)”
https://owasp.org/www-community/attacks/SQL_Injection
- [2] Bright security, “SQL Injection Attack”
<https://brightsec.com/blog/sql-injection-attack/>
- [3] Radware, “SQL Injection: Examples, Real Life Attacks & 9 Defensive Measures”
<https://www.radware.com/cyberpedia/application-security/sql-injection/>
- [4] Beagle Security “Error based SQL Injection”
<https://beaglesecurity.com/blog/vulnerability/error-based-sqli.html>
- [5] Dafydd Stittard and Marcos Pinto, The web Application Hacker’s Handbook: Finding and Exploiting Security Flaws, 2nd Edition, Wiley Publishing Inc, 2011.
- [6] OWASP, “Blind SQL injection”
https://owasp.org/www-community/attacks/Blind_SQL_Injection
- [7] Moxso, “SQL Tautology”
<https://moxso.com/blog/glossary/tautology>
- [8] Justin Clarke, SQL Injection Attacks and Deffense
- [9] OWASP Cheat Sheet Series, “SQL Injection Prevention”
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- [10] OWASP Cheat Sheet Series, “Query Parameterization”
https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html
- [11] PHP “Prepared statements and stored procedures”
<https://www.php.net/manual/en/pdo.prepared-statements.php>
- [12] PHP , “Validation”
<https://www.php.net/manual/fr/filter.examples.validation.php>
- [13] Mitchell, T. M, Machine Learning. McGraw-Hill, 1997.
- [14] Amer F.A.H. ALNUAIMI and Tasnim H.K. ALBALDAWI, An overview of machine learning classification techniques.
- [15] IBM , “Machine Learning”
<https://www.ibm.com/think/topics/machine-learning>
- [16] Reinforcement Learning: An Introduction Second edition, in progress Richard S. Sutton and Andrew G. Barto c 2014, 2015
- [17] GeeksforGeeks , “ Logistic Regression in Machine Learning”
<https://www.geeksforgeeks.org/understanding-logistic-regression/>

- [18] MathsWorks, “Support Vector Machine”
<https://www.mathworks.com/discovery/support-vector-machine.html>
- [19] IBM ,“Machine Learning”
<https://www.ibm.com/think/topics/machine-learning>
- [20] TeckTarget, “What is deep learning and how does it work?”
<https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network>
- [21] Medium “Basic Notations and Representation: Neural Networks”
<https://medium.com/@anushruthikae/basic-notations-and-representation-neural-networks-d46a1be97471>
- [22] V7 Labs, “Activation Functions in Neural Networks [12 Types & Use Cases]”
<https://www.v7labs.com/blog/neural-networks-activation-functions#3-types-of-neural-networks-activation-functions>
- [23] Shelf, “Why Recurrent Neural Networks (RNNs) Dominate Sequential Data Analysis”
<https://shelf.io/blog/recurrent-neural-networks/>
- [24] Amazon Web Services, “What is RNN (Recurrent Neural Network)?”
https://aws.amazon.com/what-is/recurrent-neural-network/?nc1=h_ls
- [25] AIML, “What are the advantages and disadvantages of a Recurrent Neural Network (RNN)? ”
<https://aiml.com/what-are-the-advantages-and-disadvantages-of-a-recurrent-neural-network-rnn/>
- [26] MathsWorks, “Long Short-Term Memory (LSTM)”
<https://www.mathworks.com/discovery/lstm.html>
- [27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, Attention is all you need, version 5, 2017.
- [28] GeeksforGeeks, “Architecture and Working of Transformers in Deep Learning”
<https://www.geeksforgeeks.org/architecture-and-working-of-transformers-in-deep-learning/>
- [29] Envisioning, “Point-wise Feedforward Network”
<https://www.envisioning.io/vocab/point-wise-feedforward-network>
- [30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,”
- [31] S. Lakhani, A. Yadav, and V. Singh, “Detecting SQL Injection Attack using Natural Language Processing,”
- [32] P. Tang, W. Qiu, Z. Huang, H. Lian, and G. Liu, “Detection of SQL injection based on artificial neural network,”

- [33] A. Paul, V. Sharma, and O. Olukoya, “SQL injection attack: Detection, prioritization & prevention,”
- [34] EVIDENTLY AI, “How to interpret a confusion matrix for a machine learning model”
<https://www.evidentlyai.com/classification-metrics/confusion-matrix>
- [35] Kaggle, “hassan bechara, sql-injection-detection”
<https://www.kaggle.com/code/hassanbechara/sql-injection-detection>
- [36] Kaggle, “Aman Rajput, PPAI Project: Comparison with other ML algorithms”
<https://www.kaggle.com/code/amanrajput27/ppai-project-comparison-with-other-ml-algorithms>
- [37] Kaggle, “Syed Saqlain Hussain Shah, SQL Injection dectection using neural network”
<https://www.kaggle.com/code/syedsaqlainhussain/sql-injection-dectection-using-neural-network>
- [38] Kaggle, “cmdrsam, Information security project”
<https://www.kaggle.com/code/cmdrsam/information-security-project>
- [39] Kaggle, “Derara Duba, SQL-Injection-Detection-via-RNN-autencoder-and-LSTM-classifier”
<https://github.com/DeraraD/SQL-Injection-Detection-via-RNN-autencoder-and-LSTM-classifier>