Karlsruher Institut für Technologie

Masterarbeit

Cedric Mössner

Entwicklung eines neuronalen Netzes zur Automatisierung von Penetrationstests

März 2018 - August 2018

Erstgutachter:	Prof. Dr. Sebastian Abeck
Zweitgutachter:	Prof. Dr. Bernhard Neumair
Betreuender Mitarbeiter:	Florian Röser

Cooperation & Management (C&M, Prof. Abeck)
Institut für Telematik, Fakultät für Informatik
www.cm.tm.kit.edu

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 6. Juni 2018

_____

Cedric Mössner

Inhaltsverzeichnis

1 Einleitung

Hackerangriffe - ein wiederkehrendes Phänomen in den Medien, vor denen auch Giganten wie Sony [Ne] oder die Regierungsnetze [CD] nicht ausreichend geschützt sind. Um solchen Angriffen zuvorzukommen, muss die IT-Sicherheit des Unternehmens funktionieren - der Code muss untersucht werden, Zugriffs- und Zugangskontrolle muss etabliert werden, Verbindungen müssen verschlüsselt werden.
Doch all diese und weitere Maßnahmen halten häufig bösartige Angreifer nicht davon ab, dennoch Wege zu finden, in die Systeme einzudringen. Aus diesem Grund werden *Penenetrationstester* - Angreifer, die vom Unternehmen selbst engagiert werden - beauftragt, diese Wege ins System zu finden. Dieser Weg ist häufig mühselig und braucht viel Zeit im ohnehin schon arbeitsintensiven Umfeld. Aus dieser Arbeit soll daher eine Methoden der künstliche Intelligenz nutzende Lösung hervorgehen, die dem Pentester - kurz für Penetrationstester - zur Hand geht, um so dem echten Angreifer immer einen Schritt voraus zu sein.

1.1 Einführung in das Themengebiet

Sehr lange Zeit war Penetrationtesting eine Untergrund-Thematik. *Black Hats*, bösartige Angreifer, die Angriffe zu ihrem eigenen Nutzen fahren, tauschten sich in Foren und Chatsystemen über aktuelle Sicherheitslücken aus. Diese wurden dann genutzt, um etwa Nutzerdaten zu erlangen oder nur Zerstörung zu verursachen. Das Internet war zu dieser Zeit noch nicht so weitgehend genutzt und Services wie Onlinebanking oder Onlineshops kaum vertreten. Doch mit zunehmender Schwere der Angriffe formierte sich eine Gruppe, die diese Schwachstellen ebenfalls suchten und an die Unternehmen weiterleiteten. Diese *White Hats* wurden jedoch ebenso missachtet wie die eigentlichen Angreifer und ihre Hinweise wurden ignoriert. Somit eröffneten sie Seiten für *Common Vulnerabilities and Exposures*, kurz *CVEs* [Co]. Dort wurden Schwachstellen für die Öffentlichkeit zugänglich gemacht - auch für bösartige Angreifer. Durch diesen radikalen Schritt wollten die Sicherheitsforscher die Unternehmen zwingen sich um ihre Sicherheit zu kümmern.
Langsam aber stetig wuchs das Bewusstsein für Sicherheit und auch große Firmen wie Google und Amazon etablierten eigene Abteilungen hierfür. In diesen wurde nach Sicherheitslücken gesucht, um diese so schnell wie möglich zu reparieren. Diese Arbeit wurde von IT-Sicherheitsforschern getätigt. Eine spezielle Abteilung bestand aus *Penetrationtestern*. Sie handelten wie die ursprünglichen Black Hats als Angreifer auf das System, das sie eigentlich beschützen sollten. Der Grund hierfür war, den Standpunkt zu wechseln und mit den selben Methoden und Tricks der wahren Bedrohung anzugreifen,

zu analysieren und zu beurteilen, sodass der Dienst bei einem echten Angriff bereits gegen alle Möglichkeiten geschützt war.
Heutzutage sind beide Rollen - IT-Sicherheitsforscher und Pentester - wichtige Faktoren für die Sicherheit eines umfangreichen Systems. Während die Ersteren Schwachstellen anhand des Codes finden, Log-Dateien nach unerlaubtem Zugriff untersuchen und logische Fehler in der Anwendung beheben, greifen Letztere die Anwendung häufig ohne ein vorheriges Wissen an. Hierzu nutzen Pentester jedoch alle Möglichkeiten, die ihnen zur Verfügung stehen. Das geht teilweise bis in das Rückentwickeln der kompilierten Anwendung oder sogar über soziale Kontakte, die Zugang gewähren, welcher eigentlich verboten gehört.
Eine der wichtigsten Tätigkeiten für das Pentesting einer Webapplikation ist jedoch das *Input Fuzzing*. Darunter versteht man das Ausprobieren verschiedener möglicherweise bösartigen Nutzereingaben. Eine Nutzereingabe kann beispielsweise in einem Login-Formular ein auf dem Server ausgeführtes SQL-Statement beenden und ein weiteres ausführen. Hierbei wird beispielsweise der Befehl für den Login erweitert durch ein immer wahres Statement, was die Passwortabfrage aushebelt und den Nutzer immer einloggt, auch bei falschem Passwort. Um solche bösartigen Eingaben zu vermeiden, ist es zunächst nötig, herauszufinden, welche davon eventuell unerwünschtes Verhalten erzeugen. Die Angriffsweisen sind hier jedoch enorm vielzählig. Unerlaubter Login ist nur eines von vielen Szenarios. Unterschieden wird zwischen *Input Fuzzing* und *Input Injection*. Bei Input Injection werden gezielt bösartige Nutzereingaben getestet um ein unerwünschtes Verhalten der App zu provozieren. Input Fuzzing ist weitreichender, da hierbei auch neue, nicht unbedingt bösartige Inputs getestet werden. Das hat den Vorteil, dass man sich nicht auf bekannte Probleme beschränkt, sondern den Raum erweitert und so neue Arten von Schwachstellen feststellen kann.
Auf der anderen Seite fallen selbstverständlich durch eine enorme Anzahl möglicher Angriffsvektoren oder *Payloads* kombiniert für jede mögliche Nutzereingabe eine gewaltige Menge an Daten an: genauer $|Payloads| * |Nutzereingabefelder|$. Hierbei zählen jedoch auch zunächst für den Nutzer versteckte Eingabefelder wie POST-Formulare oder Cookies zu möglichen manipulierbaren Angriffsflächen. Diese Menge an Daten zu analysieren und mögliche Schwachstellen zu entdecken erfordert viel Zeit und Mühe und ist sehr fehleranfällig, da hier bereits eine kleine Veränderung an der Antwort ein Indiz liefern kann.

Um menschliche Tätigkeiten zu automatisieren, werden derzeit vermehrt neuronale Netze genutzt. Sie emulieren eine dem menschlichen Denken ähnliche Struktur durch extreme Parallelisierung und starke Interaktion. Die einzelnen Neuronen gleichen hierbei sehr schwachen Prozessoren, die beispielsweise eine simple *größer als* Relation berechnen. Dies wird von einer Vielzahl an Neuronen berechnet, einer Schicht, deren Ausgabe wird aggregiert und an eine weitere Schicht gegeben. Dies kann beliebig oft wiederholt werden. Bekommt das Netz nun Feedback über richtige und falsche Ausgaben im *überwachten Training*, so können im *Backpropagation*-Prozess die variablen Gewichte angepasst werden, wodurch sich die Ausgaben des Netzes an die Vorgaben anpassen. Als Beispiel könnte man

ein neuronales Netz schreiben, bei welchem ein Bild von entweder einer Katze oder einem Hund als Eingabe dient, welches durch mehrere Schichten von Neuronen propagiert wird und im finalen Schritt ein einzelnes Neuron einen binären Ausgabewert ausgibt - null für Katze, eins für Hund. Dieses Netz wird nun initial eine hohe Fehlerrate haben. Wenn nun allerdings mehrere tausend Bilder von Katzen oder Hunden hineingegeben werden, bei denen die *Ground Truth*, der tatsächliche Wahrheitswert, bekannt ist, so kann das Netz die Parameter so anpassen, dass die Fehlerquote immer niedriger wird.

1.2 Fragestellungen

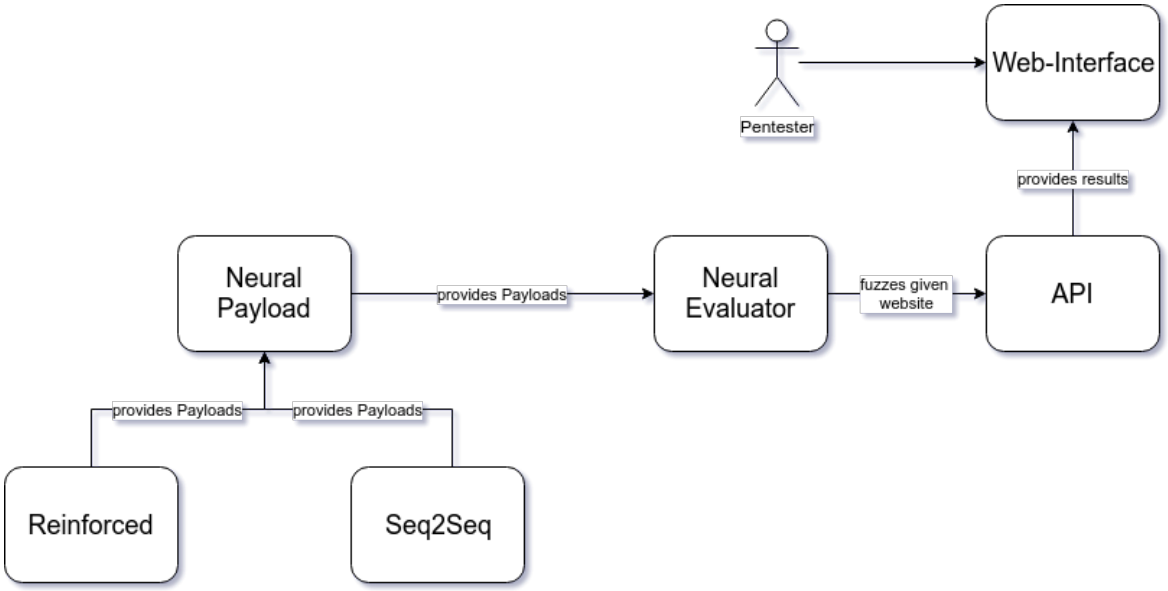

Abbildung 1.1: Übersicht der Bestandteile

An die beiden zuvor genannten Probleme des Pentesting soll diese Arbeit anknüpfen. Sowohl Payloadgenerierung als auch Ergebnisanalyse sind schwer automatisierbare Themen, da sie viel menschliches Eingreifen und Analysieren benötigen. Aus diesem Grund wird jeweils ein neuronaler Ansatz verfolgt. Zunächst gilt es, ein neuronales Netz zu erzeugen, welches ebensolche möglichen Schadcodes generiert. Hierfür müssen zuerst Trainingsdaten aggregiert werden. Die grundlegende Fragestellung hierbei lautet:
Ist ein neuronales Netz fähig, Payloads für Input Fuzzing zu liefern?
Werden eventuell sogar neuartige Payloads geliefert, die bisher unerkannt Schwachstellen aufdecken?
Daraufhin soll ein zweites neuronales Netz die Applikation modular erweitern. Dieses Netz soll die durch den Payload veränderte Website analysieren und einzelnen Zeilen eine prozentuale Schätzung zuweisen, dass hier ein Angriff vorgenommen werden kann. Interessant ist hierfür natürlich nicht nur die Website selbst, sondern auch ihre Header und dort befindliche Parameter. Auch der eingegebene Payload lässt sich dort wiederfinden - als GET oder POST-Parameter oder anderweitiges Nutzerdatum.

Es gilt hier ebenfalls herauszufinden, welche Netzarchitektur die besten Ergebnisse liefert, wie viele Schichten optimal sind und welche Algorithmen im Detail zur genauesten Erkennungsrate führen. Zuletzt sollen die Erkenntnisse in einer für den Nutzer optimalen Weise dargestellt werden. Hierfür wird in Form von Projektteam-Arbeiten, die im Rahmen der Veranstaltung "Web-Anwendungen und Serviceorientierte Architekturen"(WASA) durchgeführt werden, ein Microservice erstellt, welcher für die Ausführung der Netze ein geeignetes Frontend bereit stellt.

1.3 Beschreibung der Demonstratoren

Dieser letzte Abschnitt, die Entwicklung eines Microservices, stellt auch zugleich den Demonstrator der Applikation dar. Hierbei soll zunächst eine REST-API entwickelt werden, welche die Benutzung mit einem Frontend erleichtert. Es soll folgende Funktionen umfassen:

```
1  Feature: Visualizing the results
2
3  As a Penetrationtester
4  I want visualized attack results
5  So that I can recognize the vulnerable parts of the website instantly
6
7  Scenario: Launch the attack
8  Given I have opened the website
9    And I have entered a target
10   And I have chosen a default payload
11 When  I click on the Launch-Button
12 Then  the app uses a default configuration to attack the target
13   And I see a progress bar
14
15 Scenario: See the results
16 Given I have launched the attack
17 When  the app has finished the attack
18 Then  I see results assigning a chance of vulnerability to each line of
       HTML-Code
```

Listing 1.1: Visualisierung der Ergebnisse

```
1  Feature: Input Choosing
2
3  As a Penetrationtester
4  I want to choose the inputs to be fuzzed
5  So that I can run the app in the background
6
7  Scenario: Choose Inputs
```

```
8  Given I have opened the website
9    And I have chosen a target
10 When  I click on the Choose-Payloads-Button
11 Then  I can check or uncheck every possible User-Input
12
13 Scenario: The attack
14 Given I have chosen several inputs
15 When  I launch the attack
16 Then  the app fuzzes the chosen inputs in the attack
```

Listing 1.2: Auswahl der Eingabefelder

```
1  Feature: Pretraining
2
3  As a Penetrationtester
4  I want to pretrain the fuzzer with custom payloads
5  So that the app can have success more quickly
6
7  Scenario: Choose Payloads
8  Given I have opened the website
9    And I have chosen to pretrain a model
10 When  I choose custom payloads
11   And I choose a path for the model to save
12   And I click on pretrain
13 Then  the app trains a new model based on my payloads
```

Listing 1.3: Pretraining

1.4 Gliederung der Arbeit

In diesem Kapitel soll eine kurze Übersicht gegeben werden, welche Themen in den folgenden Kapiteln wie präsentiert werden.

Kapitel 2: Grundlagen

Hier findet sich das Wissen, welches für die folgenden Kapitel vorausgesetzt wird. Insbesondere wird ein Fokus auf Penetration Testing, neuronale Netze und die verwendeten Technologien gelegt.

Kapitel 3: Stand der Technik

In diesem Kapitel soll erläutert werden, wie aktuelle Ansätze Penetration Testing betreiben und wie sie neuronale Netze verwenden.

Kapitel 4: Neural Payload

Neural Payload ist das neuronale Netz, welches Payloads für das Input Fuzzing generieren soll. In diesem Kapitel wird die Funktionsweise, der Entwurf, Aggregation der Daten, die Evaluierung und die Verbesserungsiterationen vorgestellt.

Kapitel 5: Neural Evaluator

Neural Evaluator bezeichnet das neuronale Netz zur Analyse des Websitequelltextes, durch welchen Schlüsse über eventuelle Verwundbarkeiten gezogen werden können. Hier werden wie im vorigen Kapitel Funktionsweise, Entwurf, Datengenerierung, Evaluierung und Verbesserungsiterationen vorgestellt.

Kapitel 6: Visualisierung

In diesem Kapitel wird das Teamprojekt im Rahmen des Praktikums genauer erläutert, in dessen Umfeld der umgebende Microservice zu den Netzen erstellt werden soll.

2 Grundlagen

2.1 Grundlagen der IT-Sicherheit

Um IT-Sicherheit durchzusetzen, muss zunächst definiert werden, was vor wem wie geschützt wird. Um hier eine sinnvolle Übersicht zu erhalten, werden dafür Netzpläne erstellt, die die Infrastruktur widerspiegeln, Applikationen gelistet und auch physische Zugangswege überwacht. Da sich diese Arbeit ausschließlich auf Webapplikationen spezialisiert, sollen diese Methoden hier nicht weiter erläutert werden. Auch die potentiellen Bedrohungen seien hier nur reduziert betrachtet. Man unterscheidet gewöhnlich nach Zielsetzung, Fokussierung, Motivation und Ressourcenverfügbarkeit. [PDHH] Ein üblicher Angreifer auf ein Unternehmen kann sehr tiefes Verständnis und hohe Motivation haben, sodass Abschreckung oder einfaches *bessere Sicherheitstechnik als andere zu haben* nicht ausreicht. Auch die Ressourcenverfügbarkeit ist für diese Arbeit nicht relevant, da für die hier betrachteten Angriffsverfahren nur wenige Ressourcen nötig sind. Bei der Zielsetzung werden nun ebenfalls alle möglichen Szenarien betrachtet, seien es Zugriff auf Ressouce oder Informationen, Manipulation der Dienste oder Sabotage. Alle diese Szenarien sind im Kontext eines Webseitenbetreibers und Unternehmens fatal, da sie Gesetze brechen oder Unmengen an Geld kosten.
Um die Gefahren einordnen zu können, ist es dennoch wichtig die Schutzziele festzulegen. Man spricht hier häufig vom *CIA Triad*, da die Schutzziele *Confidentiality* (Vertraulichkeit), *Integrity* (Integrität) und *Availability* (Verfügbarkeit) darunter fallen. Weitere Schutzziele wie Datenschutz, Anonymität und Verbindlichkeit lassen sich ableiten, gehören aber nicht in übliche Definitionen [Ec14].
Generell bezeichnet Vertraulichkeit, dass alle sensitiven Daten nicht in die Hände von Dritten gelangen dürfen. Ein spezielles Bespiel wäre etwa die Kommunikation zwischen Webserver und Nutzer. Hierbei darf keine Information an Außenstehende übertragen werden. Dies wird zwar zunächst durch verschlüsselte Protokolle gewährleistet, kann jedoch umgangen werden, sobald ein Zugriff vom Angreifer auf die Datenbank Erfolg hat oder über die Website Malware verteilt werden kann.
Integrität bezeichnet die Unveränderlichkeit von Daten sowohl auf dem Kommunikationsweg als auch später. Auch hier muss ein abgesicherter Zustand der Datenbank sichergestellt werden ebenso wie clientseitige Abschirmung gegen Angriffe.
Das Ziel der Verfügbarkeit gestaltet sich oft besonders schwierig in der Sicherstellung. Es gilt hier den Server vor bösartiger Überlastung zu schützen und weiterhin authentischen Verkehr zu ermöglichen. Allerdings fallen hierunter auch bösartige Nutzereingaben, die den Server überlasten können oder Dienste zerstören. Speziell diese Angriffe sind im Rahmen dieser Arbeit interessant.

Um diese Ziele durchzusetzen muss nun analysiert werden, wo mögliche Sicherheitsprobleme existieren. Anschließend können gefundene Lücken nach Risiko - einer Kennzahl, die aus Eintrittswahrscheinlichkeit und Schwere des Angriffs generiert wird -bewertet werden um sie anschließend ihrer Reihenfolge nach zu beheben. Dieser Prozess muss kontinuierlich wiederholt werden, um neue Erkenntnisse der Forschung zu betrachten - so wird beispielsweise eine Schlüssellänge nach einer Weile nicht mehr ausreichen. Auch neue Dienste des Unternehmens sowie neue Erkenntnisse durch Nutzernachrichten oder CVEs können relevant sein. Dadurch ergibt sich ein Kreislauf, der dafür sorgt, dass Sicherheit durchgesetzt werden kann (siehe auch Abb. 2.1).

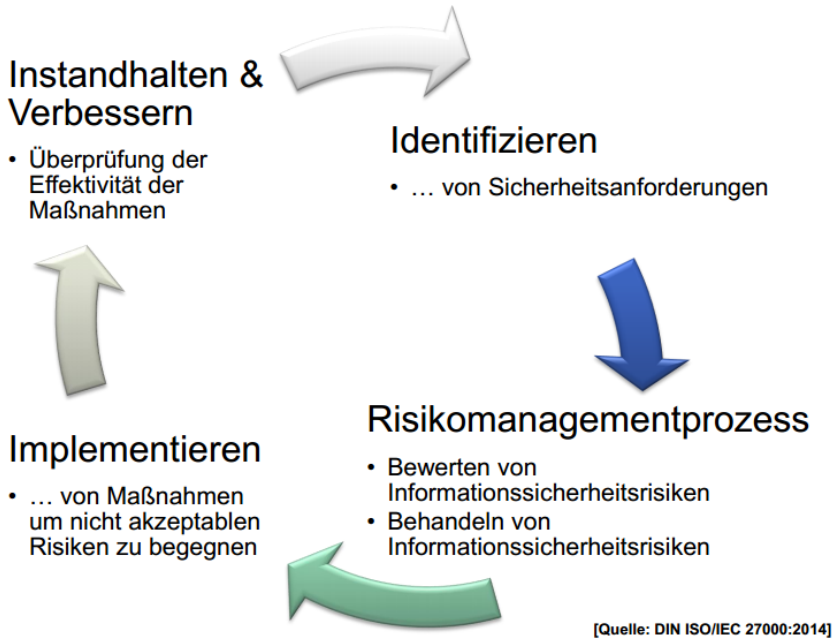

Abbildung 2.1: Prozessmodell der IT-Sicherheit [PDHH]

2.2 Penetration Testing

Ein *Penetration Test* oder kurz *Pentest* wird durchgeführt, um die Sicherheit eines Servers, eines Netzes oder einer Applikation sicher zu stellen. Hierbei wird zu Mitteln gegriffen, die von wirklich bösartigen Angreifern gewählt werden, um in das Netz unbefugt einzudringen. Durch Pentests lässt sich sicherstellen, dass man diesem zuvorkommt und die möglichen Lücken bereits behoben wurden. Es werden vielfältige Verfahren, wie schon zuvor erklärt, verfolgt. Sie alle dienen jedoch der Identifikation von Sicherheitslücken welche durch etwaige Fehler in Software, Hardware oder auch menschlichen Komponenten entstehen.
Das Vorgehen beginnt typischerweise im Pre-Engagement. Hier werden zunächst die Rahmenbedingungen des Pentests mit dem Unternehmen abgeklärt, welches ihre Dienste absichern möchte.

Hierunter fallen vor allem Dauer des Tests und welche Applikationen, Server oder Servernetze betroffen sind. Dennoch sind auch weitere Punkte interessant: Die Durchführung des Pentests kann die Server unerwartet belasten, da viele automatische Eingaben getätigt werden. Aus diesem Grund kann es ratsam sein, die Applikation entweder separat zur Verfügung zu stellen oder eine Uhrzeit für besonders leistungsintensive Tätigkeiten wie etwa nachts auszumachen. Auch die Mittel, mit denen ein Pentest getätigt werden darf, muss abgeklärt sein. Beispielsweise kann Social Engineering ein wichtiger Angriffspunkt für ein Unternehmensnetz sein, ist jedoch wenig aussagekräftig, wenn es um die Sicherheit einer speziellen Webapplikation geht. Besonders wichtig ist hierbei die Unterscheidung zwischen Blackbox Pentesting, Whitebox Pentesting und Graybox Pentesting. Beim Blackbox Testing erfährt der Sicherheitsingenieur keine internen Details über die Applikation. Speziell der Quellcode kann nicht betrachtet werden, aber auch Log-Dateien sind gesperrt. Letztere können bei einem Graybox Test normalerweise angesehen werden und bei einem Whitebox Test darf sogar der Source Code direkt untersucht werden.
Diese Arbeit fokussiert sich auf Blackbox Tests, da sie die Perspektive des echten Angreifers am besten wiedergeben.

2.2.1 Blackbox Penetration Testing

Für Blackbox Tests lassen sich typischerweise 5 Schritte erkennen:

1. Pre-Engagement
2. Footprinting
3. Scanning
4. Enumeration
5. Penetration

Das Pre-Engagement ist selbstverständlich Teil jedes Tests. Die üblichen Schritte werden aber erst nötig, da man keinen Zugriff auf weitere Ressourcen erhält.
Footprinting bezeichnet die Informationsbeschaffung, welche ohne unüblichen Zugriff auf das tatsächliche Ziel durchgeführt werden kann. Diese ist besonders wichtig, da sie nicht erkannt werden kann. Es handelt sich um öffentlich zugängliche Informationen, die man ohne besondere Kenntnisse erlangen kann. Hierunter fallen beispielsweise die IPs von Domains, DNS Einträge oder Informationen, die auf der Website direkt zu finden sind.
Im Scanning werden jedoch allerlei offensive und daher von Detection Systems erkennbaren Angriffe gefahren. Port Scans untersuchen den Server auf unübliche ausgeführte Dienste, die eine potentielle Angriffsfläche darstellen. Schwachstellen Scanner untersuchen speziell Webseiten automatisiert nach

offenkundigen Problemen. Das Problem hierbei ist jedoch, dass häufige False-Positives oder False-Negatives die Arbeit stark erschweren und meist nur mit sehr aufwendiger Konfiguration nützliche Ergebnisse erzielt werden.
Die Enumerationsphase dient dazu, die erzielten Resultate auszuwerten und so mögliche Angriffsvektoren zu identifizieren. Dadurch können spezielle Angriffsprogramme, *Exploits*, geschrieben und ausgeführt werden, welche in der Penetration-Phase angewandt werden. Typischerweise verwendet man diese Exploits nicht mehr, damit kein Schaden entsteht. Sie zu schreiben ist jedoch bei gravierenden, neuen Lücken nicht unüblich, da sie meist einen *Proof of Concept* darstellen und beweisen, wie ein Angriff möglich ist. Das erleichtert oft auch das Reparieren der Software.

2.2.2 Input Fuzzing

Input Fuzzing wird besonders bei Penetrationtests für Web Applikationen durchgeführt, kann jedoch auch bei Anwendungen, Protokollen oder sogar Dateien nützlich sein. Dabei werden für alle möglichen vom Nutzer veränderbaren Variablen potentiell gefährliche Eingabewerte verwendet. Mögliche Eingabewerte finden sich in

- HTTP Protokoll Optionen wie GET und POST, aber auch PUT, DELETE oder PATCH
- Cookies
- weiteren Header-Daten
- reguläre Formulardaten
- manipulierte Formularfelder

Diese lassen sich sehr gut über die Entwickleransichten des Browsers identifizieren, siehe auch Abb. 2.2. Jeglicher Parameter wird bei einer echten HTTP-Anfrage an einen Server mitgeschickt, welcher die Anfrage verarbeitet und mögliche Logins durchführt oder weitere Dinge berechnet. Dadurch entstehen Situationen von Nutzereingaben. Speziell mit Veränderung von Formularwerten oder Cookies rechnen viele Betreiber nicht. Das ist beispielsweise für die *Remember Me*-Funktion vieler Webseiten ein Problem. Hierbei wird nämlich ein Token und Nutzerdaten wie in einem Nutzername-Passwort-Szenario im Cookie hinterlegt. Der Cookie wird manipuliert und im schlimmsten Fall ist eine Injektion von Schadcode möglich. Ein Beispiel in der Programmiersprache PHP sähe wie folgt aus:

```
1 ...
2 $sql = "SELECT id, user, password FROM users WHERE password = '" .
      $_COOKIE["token"] . "';";
3 $result = mysqli_query($conn, $sql);
4
```

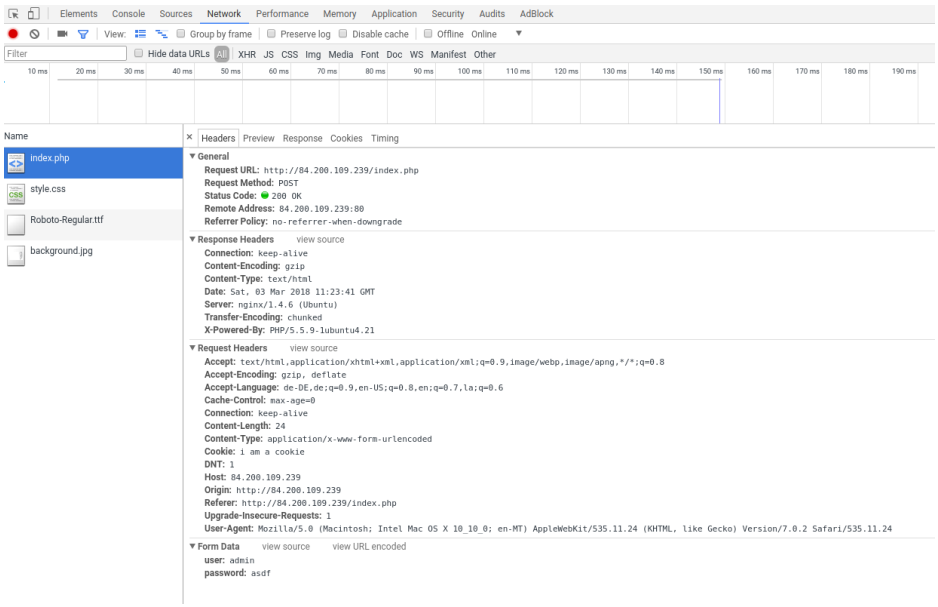

Abbildung 2.2: Identifikation von Nutzereingaben, Screenshot

```
5  if (mysqli_num_rows($result) > 0) {
6    while ($row = mysqli_fetch_assoc($result)) {
7      echo "Adminzugriff erlaubt.";
8    }
9  } else {
10   echo "Nope!";
11 }
```

Listing 2.1: PHP Beispiel für eine Cookie Injektion

Wenn nun hier eine Manipulation des Cookies erfolgt, sodass der Wert von *token* zum Beispiel *1' or '1'='1* ist, so wird das Statement immer zu *true* evaluieren, wodurch ein Nutzer unerlaubt eingeloggt wird. Dies lässt sich durch einfache *prepared statements* umgehen, sodass der Wert der Token-Variable als wirkliche Variable eingesetzt wird und nicht durch String-Operationen.
Es existieren weitere nicht triviale Fälle, die teilweise auch nur zu einer erhöhten Auslastung des Servers führen. Da solche Denial-of-Service-Angriff das Schutzziel der Verfügbarkeit beeinträchtigen, gilt es auch solche Eingaben zu finden. Ein beliebtes Beispiel sind Eingaben in Formularfelder, die schlichtweg zu lang sind. Eingabefelder sind von Natur aus nicht limitiert. Wenn nun Strings mit mehreren Millionen Zeichen eingefügt werden, hat dies bei vielen Servern zur Folge, dass der Dienst abstürzt.
Grundsätzlich werden Angriffsvektoren meist aus einer Kombination von Zahlen (Null, mit Vorzeichen, ohne, Gleitkomma), Strings (Urls, Kommandozeilenbefehle), Metadaten und binären Zeichen

erstellt [OW].
Das Erkennen einer potentiellen Schwachstelle ist meist sehr aufwendig, kann aber von einem Menschen oft gut erkannt werden. Fehlermeldungen sind häufig deutliche Anzeichen von möglichen Angriffen. Sie können Indizien liefern, welche Datenbankstruktur im Hintergrund läuft oder welches Eingabefeld möglicherweise fehleranfällig ist. Ein weiterer möglicher Angriff sind *Cross-Site-Scripting*-Angriffe, kurz XSS. Dabei wird zum Beispiel ein Nutzername so gewählt, dass er HTML-Code enthält, welcher JavaScript ausführt. Das lässt sich, wenn der Name nicht überprüft wird, relativ leicht sowohl durchführen als auch erkennen. Ein beispielhafter Nutzername findet sich in 2.2.

```
1 Cedric <script>alert("XSS-Alarm");</script>
```

Listing 2.2: Cross Site Scripting - Nutzername

Der Name wird als normaler Benutzername angezeigt, sodass nicht auffällt, dass ein Angriff erfolgt ist, wohingegen das JavaScript im Hintergrund ausgeführt wird. Dadurch wird ein Diebstahl von Cookies möglich, indem der Seiten-Cookie - möglicherweise mit einem Login Token - ausgelesen und via JavaScript an den Angreifer weitergegeben wird. Dadurch können vollständige Sessions gestohlen werden. Die Erkennung gestaltet sich hier allerdings relativ einfach. Wenn der Nutzername Sonderzeichen enthält und diese ohne Entfremdung wieder im Antwortcode erscheinen, kann von möglichen XSS-Angriffen ausgegangen werden. Problematisch ist jedoch die Nutzung von internationalen Namen, Kommentar- oder Beschreibungsfeldern. Man kann hier dann die HTML-Felder entweder via *Sanitizing* in normalen Text umwandeln - auch die Sonderzeichen, oder aber man verbietet jegliches gefährliche Zeichen.
Es wäre, je nach Konfiguration des Webservers, auch möglich durch Nutzereingaben direkte Kommandozeilenbefehle auszuführen, einen Overflow zu erzeugen oder weitere gravierende Befehle durchzuführen. Um hierfür geeignete Payloads zu finden, müsste ein endlos großer Suchraum ausprobiert werden, von welchem höchstens ein kleiner Anteil wirkliche Probleme aufzeigt. Hierzu sollen in dieser Arbeit neuronale Netze zum Einsatz kommen.

2.3 Neuronale Netze

Neuronale Netze sind ein Ansatz, das menschliche Gehirn auf einem gewöhnlichen Computer zu imitieren. Das menschliche Gehirn nutzt eine Vielzahl an sehr schwachen Prozessoren (Neuronen). Dadurch entsteht ein starker Parallelismus bei der Berechnung. Anders als bei üblicher Parallelität, wo die Interaktion zwischen Prozessen möglichst gering gehalten wird, wird hier eine hohe Kohäsion genutzt. Die Neuronen werden in Schichten zusammengefasst. In der ersten Schicht wird das Inputdatum angelegt, etwa ein Bild oder die kodierte Version eines Buchstaben. Hierüber wird dann eine Berechnung in jedem Neuron gestartet. Die Neuronen der weiteren Schichten bekommen

dann den Ausgabewert jedes Neurons der Vorgängerschicht, berechnen jedoch anhand ihrer internen Gewichte unterschiedliche Ergebnisse. Diese Berechnung nutzt eine festgelegte Ausgabefunktion. Zu diesen später mehr. Daraufhin wird die Ausgabe jedes Neurons an jedes Neuron der nächsten Schicht weitergeleitet. So entsteht eine stark parallelisierbare Berechnung, die aus diesem Grund in modernen Implementierungen meist auf der Grafikkarte ausgeführt werden, welche deutlich mehr Rechenkerne als ein gewöhnlicher Prozessor enthält. Diese Architektur nennt sich *Feed-Forward Netz*. Es existieren komplexere Alternativen, die später erläutert werden.
Die Gewichte der Neuronen können aufgrund ihrer Zahl unmöglich manuell implementiert werden.

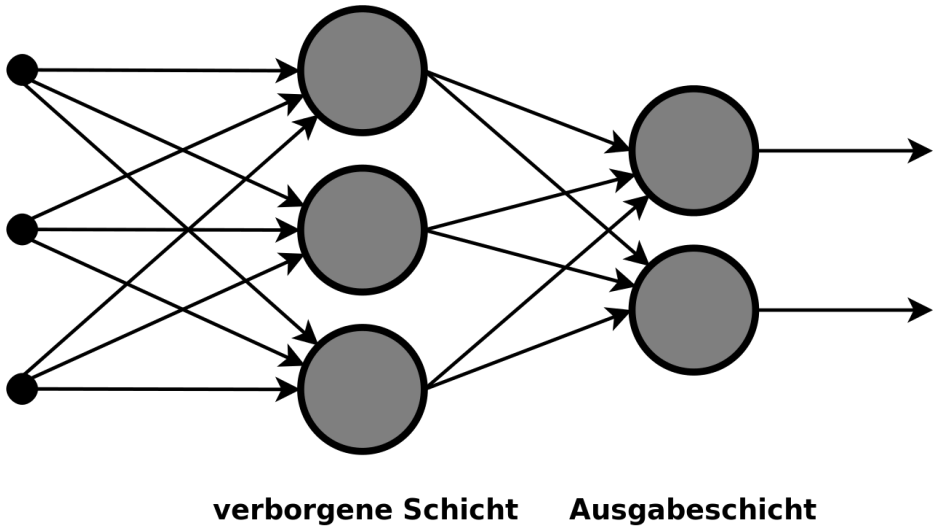

Abbildung 2.3: Ein mehrschichtiges neuronales Netz [Of]

Stattdessen werden sie trainiert. Dies geschieht in den meisten Fällen überwacht. Das bedeutet, man nutzt eine gewisse Anzahl an Trainingsdaten, um dem Netz zu zeigen, welche finale Ausgabe bei welchem Eingabewert erwartet wird. Durch den *Backpropagation*-Algorithmus werden dadurch die Gewichte angepasst. Er nutzt eine Fehlerfunktion, um die Distanz des tatsächlichen Ausgabewertes zum erwünschten Ausgabewert zu errechnen. Anhand dieses Fehlers können dann Gradienten berechnet werden und die Gewichte werden angepasst. Hier soll diese oberflächliche Beschreibung genügen, da der Algorithmus von modernen Frameworks bereits implementiert ist. Das Aktualisieren der Gewichte geschieht jedoch in kleinen Schritten, da auch in den Trainingsdaten einzelne fehlerhafte Angaben sein könnten. Es wird eine Lernrate α verwendet, sodass die Gewichte nach der Formel

$$w_{neu} = w_{alt} + \alpha * gradient$$

berechnet werden.
Dadurch werden die Gewichte anhand der Trainingsdaten angepasst und es entsteht eine Lernkurve. Je mehr Trainingsdaten vorhanden sind, desto länger lässt sich das Netz trainieren, jedoch können

dieselben Daten auch mehrfach zum Training genutzt werden. Dies nennt sich Epoche.
Um die Güte des Netzes zu Evaluieren werden die verfügbaren Daten gewöhnlicherweise in zwei Sätze aufgeteilt: Die Trainingsdaten und die Validierungsdaten. Letztere enthalten zwar auch Eingabedaten und erwünschte Ausgaben, jedoch wird hier kein Training durchgeführt. Es wird ausschließlich dem Netz das Eingabedatum gegeben und der Fehler berechnet. Dadurch kann man feststellen, wie gut das Netz generalisiert, also auf bislang ungesehenen Daten funktioniert. Der Fehler hier ist deutlich aussagekräftiger als auf den Trainingsdaten. Es könnte ja sein, dass die Trainingsdaten nicht repräsentativ für die eigentlichen Aufgaben des Netzes sind. Zum Beispiel könnte man an die Klassifikation von Fotografien denken. Bei den Trainingsdaten würden immer ausschließlich sonnige Tage genutzt werden, während die eigentliche Aufgabe häufig auch im Regen stattfindet. Dann würde das Netz nicht gut generalisieren. Ein weiteres Problem stellt *Overfitting* dar. Overfitting bezeichnet grob gesagt das Auswendiglernen des Netzes. Das Netz wird danach also auf den Trainingsdaten eine enorm gute Fehlerrate haben, jedoch auf echten Daten, wie denen im Validierungsset, sehr schlecht abschneiden. Es lässt sich allerdings sehr gut an der Trainingskurve erkennen, wie in Fig. 2.4 dargestellt.

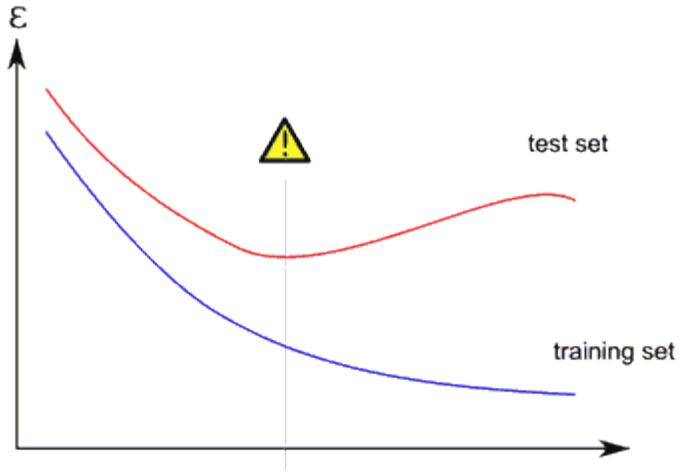

Abbildung 2.4: Beispielhafte Kurven bei Overfitting [Gr]

2.3.1 Rekurrente neuronale Netze und Linear Short Term Memory

Das Feed-Forward Netz ist zwar eine interessante Struktur, jedoch hat sie einen entscheidenden Nachteil: Es kann kein Verhältnis zwischen verschiedenen Eingabedaten hergestellt werden. Das ist für viele Problemstellungen kein Problem, für Sequenzen von Daten hingegen kann es hilfreich sein, ein solches aufbauen zu können. Das beste Beispiel bietet die Spracherkennung. Ein einzelner Laut von 30ms lässt sich auch kaum von einem Menschen klassifizieren schon gar nicht, wenn er alleine steht, allerdings ein vollständiges Wort erkannt werden soll. Wenn man hingegen die 30ms-Stücke

hintereinander - nicht auf einmal - abspielt, erkennen wir Worte mit Leichtigkeit.
Hier setzen rekurrente neuronale Netze an. Sie verknüpfen die Ausgabe einer Schicht mit der Eingabe einer Schicht. Es ist hierbei möglich, die finale Ausgabe mit der Eingabeschicht zu verknüpfen oder die Zwischenschichten zu verwenden, alle Ansätze sind möglich. Diese Verknüpfung führt dazu, dass Informationen zwischen mehreren Klassifikationen übertragen werden. Zurück bei dem Beispiel der Spracherkennung bedeutet das, dass die Informationen des vorhergehenden 30ms Stück mit klassifiziert werden. Wie oder welche dieser Informationen übertragen werden, wird wieder trainiert. Der Algorithmus hierzu ist ebenfalls wieder Backpropagation, jedoch wird das Netz ausgerollt. Das ist möglich, da die Eingabe nur eine endliche Länge hat. So entsteht *Backpropagation Through Time*. Das Problem hierbei ist allerdings, dass die Netze häufig entweder das *Vanishing Gradient*

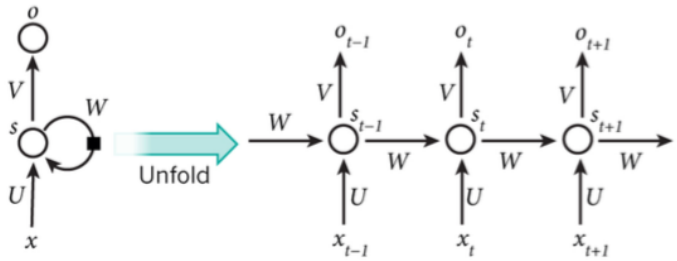

Abbildung 2.5: Rekurrente Netze können ausgerollt werden, um Backpropagation Through Time zu ermöglichen. [KK]

oder *Exploding Gradient* Problem aufweisen. Hierbei werden bei langen Zeitreihen entweder die Informationen vom ersten Aufruf völlig vernachlässigt oder ausschließlich relevant. Das hat eine schlechtere Klassifikation zur Folge.
Als Lösung hierzu wurde eine spezielle Aktivierungsfunktion vorgestellt, welche die Rekursion bereits in sich einschließt. *Long Short Term Memories*, kurz LSTMs, nutzen einen komplexen Mechanismus mit mehreren lernbaren *Gates*, die angeben, wie viel vergessen oder übertragen werden soll. Sie sind der De-Facto Standard für lange Reihen, nutzen allerdings viele Gewichte, die alle trainiert werden müssen. Es müssen also viele Trainingsdaten vorhanden sein um gute Ergebnisse zu erzielen.

2.4 Reinforcement Learning

Reinforcement Learning an sich ist zunächst kein neuronales Konzept, wird aber vermehrt in diesem Szenario verwendet. Die Technik selbst geht zurück auf den Pawlowschen Hund [Mc]. Kurz zusammengefasst, geht es darum, über Belohnung und Bestrafung dem Hund - oder, übertragen, der künstlichen Intelligenz - zu signalisieren, welche Handlung erwünscht und welche unerwünscht ist. Im empirischen Experiment wurde dem Hund das Futter immer mit dem Klingeln einer Glocke gegeben. Das Essen an sich löste vermehrte Speichelproduktion aus. Wurde dies lange genug gemacht, konnte ein einfaches Klingeln der Glocke ohne die Gabe von Futter Speichelproduktion auslösen. Durch

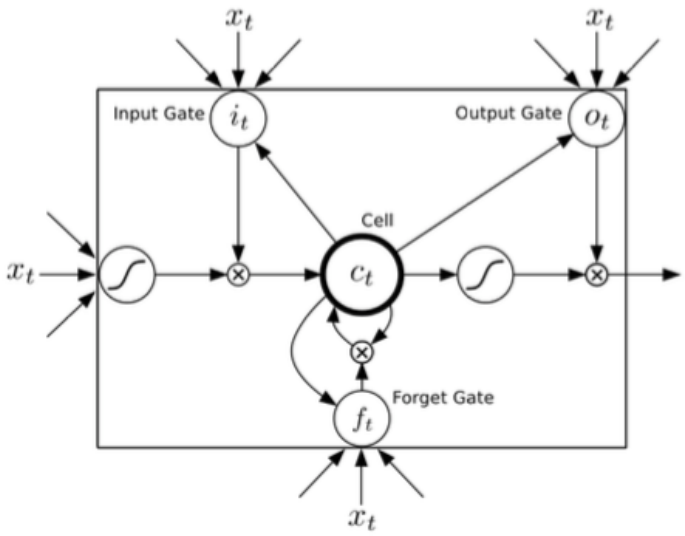

Abbildung 2.6: Eine Einheit eines Linear Short Term Memories. [KK]

diesen Ansatz lassen sich komplexere Handlungen wie Spiele oder Robotik modellieren, jedoch kann es auch Vorteile für den Ansatz in dieser Arbeit bieten.
In der Praxis lässt sich dieses Konzept auf maschinelles Lernen gut übertragen. Im Folgenden wird dazu ein Spiel als Beispiel genutzt, welches gewonnen werden soll. Als der *Agent* wird die künstliche Intelligenz bezeichnet. Es sind vier Variablen für den Algorithmus wichtig:

- Der aktuelle Zustand des Spiels
- Die Aktion, die der Agent durchführt.
- Eine Funktion $P(s|a)$, die angibt, wie hoch die Wahrscheinlichkeit ist, dass der Agent im Zustand s die Aktion a durchführt
- Eine Belohnungsfuntion $R(s,s')$, die bestimmt, welche Belohnung der Agent nach Zustandsübergang von s auf s' erhält

Es ist klar: Der Agent soll immer die Zustandsänderung wählen, die die maximale Belohnung verspricht. Meist ist auch klar, welcher Zustand erreicht wird, wenn der Agent eine Aktion durchführt. Jedoch ist die Belohnung selten eindeutig. Im Fall eines Spiels, in welchem der Agent in vier Richtungen gehen kann (vier Aktionen) und ein bestimmtes Feld erreichen soll, welches ihm nicht zuvor bewusst ist, wird das Problem deutlich. Er muss also ausprobieren, welches Feld für ihn eine Belohnung enthält und welches eine Bestrafung. Als deutliches Beispiel sei Abb. 2.7 genannt. Es ist leicht, die optimale Route zu erkennen, wenn man bereits das gesamte Spielfeld kennt. Jedoch ist es eher schwierig, wenn man nicht weiß, wo die Belohnung liegt. Es bleibt nichts, als jedes Feld auszuprobieren. Erhält man eine Belohnung oder eine Bestrafung, so kann das Ergebnis rückwirkend

festgelegt werden. Läuft der Agent also zunächst zwei Schritte nach links und schließend drei nach oben, kann jedem dieser Schritte eine Belohnung zugewiesen werden, da das Spiel erfolgreich abgeschlossen wird. Läuft der Agent schließend jedoch zwei Schritte nach links, zwei nach oben, eins nach rechts und eins nach oben, so wird eine negative Belohnung auf Felder ausgelegt, die ursprünglich eine Belohnung enthielten und die ja auch den kürzesten Weg zum Ziel darstellen. Um diesem Verhalten entgegenzuwirken werden mehrere Maßnahmen eingeleitet: Es wird *Q-Learning* genutzt.

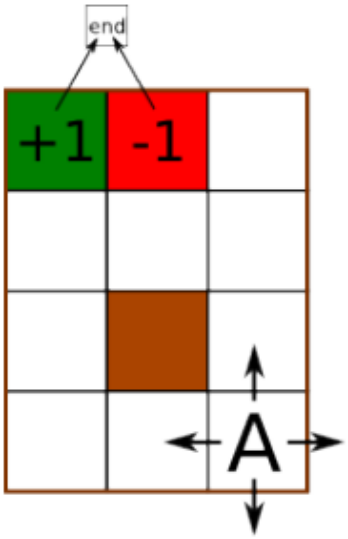

Abbildung 2.7: Ein beispielhaftes Szenario, in dem ein Agent lernen soll, welche Felder Belohnung und welche Bestrafung bringen. [KK]

2.4.1 Q-Learning

Die *Q*-Funktion verhält sich ähnlich wie die Belohnungsfunktion *R*, jedoch ist die Funktion nicht abhängig von zwei Zuständen - also einem Zustandsübergang - sondern modelliert die erwartete Belohnung bei Wahl einer Aktion a in Zustand s. Auf das Beispiel bezogen wird nun nicht mehr eine Belohnung vergeben, wenn der Agent vom Anfangszustand auf das untere mittlere Feld läuft, sondern wenn er im Anfangszustand die Aktion *LINKS* ausführt. Der Vorteil hieran ist, dass der Agent diese Aktion vollständig beeinflussen kann, der neue Zustand jedoch verborgen ist. Nun kann in jedem Zustand das deterministisch berechenbare Maximum der Q-Funktion aus allen möglichen Aktionen a berechnet werden.

$$\pi^*(s) = argmax_a Q(s,a)$$

Die Q-Funktion ist nun das Objekt, das erlernt werden soll. In nicht-neuronalen Ansätzen wird hierzu bei Beendigung eines Spiels das Ergebnis auf die durchlaufenen Zustände und Aktionen übertragen,

jedoch mit einem Faktor γ, der frühere Zustände schwächer beeinflusst als spätere, da von dort ja deutlich mehr Aktionen möglich sind:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \cdot argmax_{a'} Q(s',a') - Q(s,a))$$

Weitere Optimierungen existieren, wie etwa ein *Living Punishment*, bei dem der Agent in jedem Zug automatisch bestraft wird, wenn er das Ziel noch nicht erreicht hat. Das sorgt etwa dafür, dass er nicht ständig gegen eine Wand läuft auf einem Feld, welches mit einer Belohnung versehen ist.

2.4.2 Deep-Q-Learning

Aufgrund hoher Komplexität von Zuständen, einer Vielzahl von Aktionen, die durchgeführt werden können und sehr langer Spieldauer, ist es häufig schwer, traditionelles Reinforcement Learning durchzuführen. Aus diesem Grund wird die Q-Funktion als neuronales Netz modelliert. Das Netz bekommt den aktuellen Zustand als Eingabe und soll, je nach Ansatz die Aktion mit der höchsten Belohnung zurück geben oder die erwartete Belohnung für eine ebenfalls gegebene Aktion ausgeben. Dieser Ansatz hat bereits häufig zum Erfolg geführt, wie etwa in Google's AlphaGo [Li]. Das Netz wurde dazu zunächst auf Spielerzügen von Experten trainiert, da diese meist eine hohe Belohnung versprachen, anschließend ließen die Entwickler die künstliche Intelligenz gegen sich selbst und gegen andere Spielen.
Um eine Variation bei den Trainingsmustern zu erlangen, wird häufig ein Gedächtnis eingeführt, in welchem die letzten x Aktionen gespeichert werden. Aus diesem werden nach einer Weile des Anwendens einige zufällige Aktionen herausgewählt und das Netz kann trainiert werden [Pa].

2.5 PyTorch

PyTorch ist Framework für Python, welches die Erstellung von neuronalen Netzen stark erleichtert. Es müssen weder Backpropagation, noch häufig genutzte Aktivierungsfunktionen oder sonstige Interna selbst programmiert werden. Darüber hinaus ist es möglich, die Ausführung des Modells auf die Grafikkarte zu verlagern. Durch die enorme Parallelität bringt dies einen Geschwindigkeitsschub von circa Faktor 13 (eigene, empirische Beobachtung). Das sind jedoch Merkmale, die von allen aktuellen Frameworks unterstützt werden. Hier soll kurz elaboriert werden, weshalb dieses Framework gewählt wurde.
Die größten Frameworks sind aktuell Tensorflow und PyTorch. Letzteres ist eine modernere Variante von Torch, einem Lua-Framework. Tensorflow ist auf einer sehr tiefen Ebene, welche den Programmierer zwingt, vieles selbst in die Hand zu nehmen. Aus diesem Grund wird Tensorflow meist mit einem abstrakteren Framework wie Keras genutzt. PyTorch hingegen ist ebenso abstrakt wie Keras.

Beide sind daher intuitiv nutzbar und leicht verständlich. Keras wird allerdings - im Gegensatz zu Tensorflow, welches von Google produziert wird - von wenigen Hobbyprogrammieren verwaltet und daher sind Updateänderungen nicht so detailliert, wie man es sich wünschen würde und auch die Dokumentation ist teilweise eher grob. PyTorch hingegen wird von Facebook entwickelt und hat eine detaillierte Dokumentation, die auch gut gepflegt ist.
Diese Unterschiede sind jedoch sehr marginal, wenn man den *Computational Graph* betrachtet. Tensorflow nutzt einen *Static Computation Graph*, PyTorch hingegen einen *Dynamic Computation Graph* [TLH]. Bei Ersterem wird das Netz als solches kompiliert und hat dadurch eine feste, statische Struktur, die sich im Nachhinein nicht mehr variieren lässt. Das ist problematisch, falls man für bestimmte Eingabewerte andere Schichten nutzen möchte. PyTorch bietet hier also einfach mehr Möglichkeiten. Nun könnte man vermuten, dass die Leistung bei Static Computation Graphs deutlich höher ist. Dass dem nicht so ist, wie mehrere Nutzer in Foren bestätigten [Vea] [Veb]. Grund dafür sind einige Optimierungen, die PyTorch im Hintergrund durchführt [TLH].
Da PyTorch jedoch sehr jung ist, ist plattformübergreifendes Arbeiten noch nicht ohne weiteres möglich. So wird zum Beispiel Windows nicht als Entwicklungsplattform unterstützt.
Da aber PyTorch mehr Funktionalität bietet und keine relevanten Einbußen für den Entwickler gemacht werden, wird diese Arbeit PyTorch als Framework nutzen.
Das Framework selbst ist installierbar via *pip* für alle gängigen Python-Versionen. Es werden zusätzlich gleich die Bibliotheken für die Verwendung von CUDA in PyTorch installiert. Cuda ist ein gängiges Framework für *General Purpose Computation on Graphics Processing Unit*, also dem Programmieren von Grafikkarten für andere Zwecke als Grafik. Es wird von der Firma Nvidia angeboten für ihre hauseigenen Grafikkarten.

2.6 Integration in REST-Architektur

Um die Applikation für den Benutzer leichter zugänglich zu machen, soll im Rahmen des WASA-Praktikums von Cooperation & Management (C&M) eine REST-Architektur entwickelt werden. Dadurch sollen die Funktionen gekapselt werden und somit nicht mehr über Konfigurationsdateien und die Kommandozeile, sondern über ein intuitives Frontend verwendet werden.

2.6.1 REST

Representational State Transfer, kurz REST, bezeichnet einen Architekturstil, der häufig für Webservices verwendet wird. Wie der Name bereits suggeriert, ist die Grundlage hierfür ein zustandsloses Backend. Das erspart dem Backend - also dem Server - einiges an Rechenaufwand, da der Zustand bei jeder Anfrage vom Frontend oder anderen Nutzern vollständig mitgeliefert werden muss. Für diese Kommunikation wird meist das HTTP-Protokoll verwendet. Es liefert bereits alle wichtigen Statuscodes mit und muss nur wenig angepasst werden. Als Austauschformat eignet sich JSON gut.

Die einfache Struktur ermöglicht eine schnelle und unproblematische Überführen in ein assoziatives Dictionary, welches etwa in Python gut verwendet werden kann. [PSA]

2.6.2 Frontend

Als Frontend bezeichnet man den clientseitig ausgeführten Code, der von dem Browser interpretiert und angezeigt wird. Er besteht meist aus HTML für den Inhalt, CSS für die Definition von Stylekaskaden und JavaScript für die dynamischen Elemente.

2.6.2.1 Angular

Angular ist ein von *Google* entworfenes Framework, welches die Kombination der erwähnten Technologien stark erleichtert. Es nutzt TypeScript, eine Java-ähnliche typisierte Sprache, die zu JavaScript kompiliert und somit vom Browser interpretiert werden kann. Zusätzlich bietet es allerdings eine Vielzahl an automatisch generiertem Code und eine umfangreiche Library, welche den Einsatz von REST vereinfacht.

2.6.3 Backend

Das Backend wird definiert durch Code, der auf dem Server ausgeführt wird und Ressourcen bereit stellt. Diese können durch eine Anfrage an die API vom Backend abgefragt werden. Getreu den Prinzipien von REST, muss das Backend keinen internen Zustand für einzelne Nutzer halten. Dies minimiert nicht nur die Angriffsfläche für Denial-of-Service Angriffe, sondern ermöglicht auch eine größere Nutzerzahl.

2.6.3.1 Flask

Da die neuronalen Netze in Python geschrieben sind, ist es sinnvoll, auch das Backend und die API in Python anzubieten. Hierzu bietet sich die Bibliothek Flask an. Diese ist beabsichtigt minimalistisch gehalten und liefert nur für REST nötige Dienste mit. Dadurch lassen sich die Prinzipien einfacher umsetzen, ohne dass zu viel umgebender Code geschrieben werden muss. Dadurch kann auch eine eigene Struktur einer Applikation leichter genutzt werden.

2.7 Evaluation

Um die Güte sowohl des Payload generierenden Netzes als auch des erkennenden Netzes zu erkennen, werden diverse Webapplikationen genutzt, die beabsichtigte Schwachstellen enthalten. Für klassifizierende neuronale Netze wären einige Validierungsdaten, die das erwünschte Ergebnis enthalten genug, jedoch ist dies im Blickfeld des Fuzzings nicht ausreichen. Es sollen neue Daten generiert werden, die sogar neue, unbekannte Schwachstellen finden. Hierzu kann es noch keine Ground-Truth-Daten geben. Aus diesem Grund müssen die Angriffe durchgeführt und ihr Ergebnis evaluiert werden. Hierzu dienen besagte Webapplikationen. Sie waren ursprünglich dafür gedacht, Pentestern eine legale Angriffsfläche, an der Fähigkeiten getestet und verbessert werden können. Sie werden lokal betrieben, sodass Angriffe mit verhältnismäßig geringer Latenz ausgeführt werden können. Auch der lokale Server kann dann konfiguriert werden, sodass zum Beispiel PHP Fehlermeldungen ausgibt, die helfen können bei der Erkennung von Schwachstellen. Es werden verschiedene Testumgebungen genutzt, um ein breit gefächertes Bild eines Erfolgs oder Misserfolgs zu erlangen. Dadurch soll das neuronale Netz besser generalisieren und weniger mit Overfitting zu kämpfen haben.

2.7.1 Hacking Challenges

Der Autor dieser Arbeit geht in seiner Freizeit einem YouTube-Projekt nach. Dort bietet er Interessierten die Möglichkeit Penetrationtesting und Weiteres zu lernen. Um einen legalen Ort zu bieten, an dem geübt werden kann, stellt er den Zuschauern *Hacking Challenges* zur Verfügung. Diese sind in PHP geschrieben und nutzen ein MySQL Backend. Jede der Challenges enthält exakt eine Schwachstelle, durch die ein Passwortfeld ausgefüllt werden kann um so die Challenge zu lösen. Es gibt dabei keine Hinweise auf die Lösung, sodass die Teilnehmer auch selbst recherchieren müssen. Im Laufe dieser Arbeit werden weitere Challenges entstehen müssen, sodass andere Lücken getestet werden können.

2.7.2 Damn Vulnerable Web Application

Damn Vulnerable Web Application, kurz *DVWA*, ist eine “Webapplikation, die verdammt angreifbar ist“[te]. Sie wurde ebenfalls entworfen, um Pentestern eine Fläche zu geben, um ihre Fähigkeiten zu testen und zu trainieren. Über das Team hinter DVWA ist leider nichts bekannt. Auch diese Applikation ist in PHP entwickelt, nutzt eine MySQL-Datenbank und der vollständige Code ist Open-Source. Der Nachteil hierbei ist, dass mehrere Schwachstellen auf einmal existieren können, die die Evaluation erschweren können.

2.7.3 Webgoat

Webgoat ist ebenfalls ein Open-Source Projekt, entwickelt von den Mitgliedern der *The Open Web Application Security Project* Foundation. Auch hier sind mehrere Webseiten enthalten, welche gezielte

Schwachstellen beinhalten. Die Applikation ist jedoch in Java programmiert und kann entweder als Docker-Container oder als normale Java-App gestartet werden [Pr].

2.7.4 Juice Shop

Das Juice Shop Projekt wird ebenfalls von *The Open Web Application Security Project* angeboten und ist eine AngularJS-Applikation mit einem node.js-Backend und nutzt eine SQLite-Datenbank. Es kann ebenfalls als Docker-Container oder über node.js direkt gestartet werden [Ki].

3 Stand der Technik

Bislang sind neuronale Ansätze im Penetration Testing eher selten vertreten. Meist wird für Input Fuzzing eine Grammatik von Hand erstellt, was sehr zeitraubend ist, oder es werden alle interessanten Payloads getestet. Letzteres ist ein sehr rechenintensiver Prozess, der im gründlichsten Fall niemals fertiggestellt ist. Das liegt am unbeschränkten Raum für Eingaben und dem dadurch entstehenden nicht-Determinismus. Der Nachteil bei beiden Ansätzen ist jedoch, dass sie entweder wirklich alle möglichen Eingaben testen müssen, worunter mehr als 90% nutzlos sind, oder aber die Eingaben von Menschen generiert werden, wodurch leicht Fehler entstehen und die nötige Exploration zum Finden neuer, bislang unentdeckter Schwachstellen fehlt.

3.1 Neuronale Netze zur Inputgenerierung

3.1.1 Learn&Fuzz:Machine Learning for Input Fuzzing

Aus diesem Grund haben sich die Forscher des Microsoft Research Teams dazu entschlossen, für den PDF-Viewer ihres Browsers *Edge* ein neuronales Netz zu erstellen, welches interessante PDF-Dateien generiert, um so neue Schwachstellen zu finden. Sie erhoffen sich so, Fehler in der Grammatik, die ebenfalls von ihnen erstellt wurde auszumerzen und bessere Ergebnisse zu erzielen. Die Fehler können leicht entstehen, da die Spezifikation von PDF-Dateien etwa 1300 Seiten umfasst [PG17].
Hierzu werden rekurrente neuronale Netze genutzt, die PDF-Objects erstellen. PDF-Objects sind,

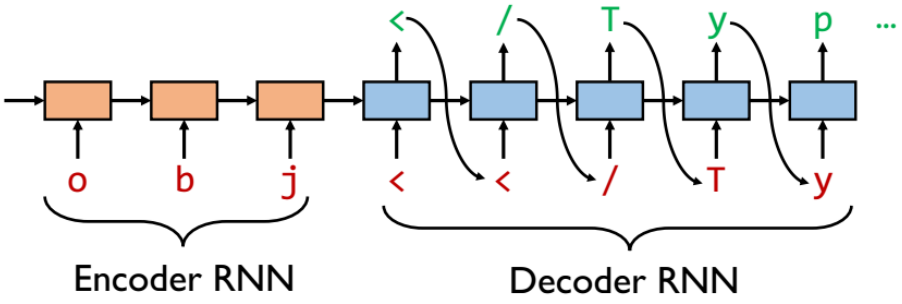

Abbildung 3.1: Das Sequence-to-Sequence Modell von [PG17].

kurz gesagt, die Bausteine einer PDF-Datei. Als Metrik wird die Code-Coverage ihres PDF-Parsers verwendet. Der konkrete Ansatz sieht ein *sequence-to-sequence Modell* vor. Siehe hierzu auch Abb. 3.1

So wird es möglich, Eingaben beliebiger Länge ebenso wie Ausgaben beliebiger Länge zu ermöglichen. Hierzu wird rekurrent die Eingabe verarbeitet, welche im Gesamten ein Zwischenergebnis produziert. Dieses besteht nur aus einem Vektor. Dieser dient als Initialisierungsvektor für den generierenden Teil des Netzes. Es nutzt dieses Zwischenergebnis um wiederum rekurrent einzelne Zeichen auszugeben. Gemeinsam ergeben Eingabe und Ausgabe dann das PDF-Object.
Es wird zwischen 3 Ansätzen unterschieden:

- NoSample, einem greedy-Algorithmus, der immer das Zeichen mit der höchsten Wahrscheinlichkeit für die aktuelle Position ausgibt.
- Sample, einem Ansatz, welcher mehrere Buchstaben ausprobiert und
- SampleSpace, welcher Sample nach Whitespaces nutzt und ansonsten NoSample

Während NoSample die höchste Quote an ordnungsgemäß formatierten und somit ohne Fehler angezeigten PDF-Objects generiert, produziert Sample die größte Varianz, was positiv für das Fuzzing ist. SampleSpace soll hier einen vernünftigen Tradeoff darstellen.
Das Netz selbst besteht aus einem drei-Schichten Modell mit jeweils 128 Linear Short Term Memory Einheiten. Nach einem Training über 50 Epochen auf 63000 PDF-Objects aus insgesamt 534 PDF-Dateien wurde mit SampleSpace eine Code-Abdeckung des Parsers von 97% erreicht. Zudem wurde eine stackoverflow Schwachstelle gefunden, die bislang unentdeckt war.

3.1.2 DeepHack

Das DeepHack-Projekt der Firma BishopFox wurde im Rahmen der DefCon 2017 vorgestellt [Bib]. Es existiert als Open-Source Projekt auf Github [Bia]. Leider existieren keine offiziellen Forschungspaper dazu. Die Informationen im Vortrag sind sehr grundlegend und das Projekt selbst ist spärlich dokumentiert. Da jedoch nur wenige Ansätze existieren, soll auch dieser hier erwähnt sein. Sie nutzen einen andersartigen Ansatz: Deep Reinforcement Learning. Anders als bei einem Spiel, bekommt das Netz hier eine Belohnung, wenn es einen erfolgreichen Angriff durchführt. Leider existiert auch hier keine genauere Beschreibung. Die Autoren erzählten ebenfalls von vielen Trainingsdaten, die sie genutzt haben, welche bei reinem Reinforcement Learning so nicht nötig sind.
Aus der Analyse des Quellcodes geht hervor, dass ein Netz genommen wurde mit folgender Struktur:

- Für die Eingabe eines Kontextes:
  - LSTM-Schicht mit 32 Neuronen
  - Schicht mit 128 Neuronen
- Für die Eingabe eines Anfrage-Strings:

  - LSTM-Schicht mit 128 Neuronen
  - Schicht mit 128 Neuronen
- Für die Eingabe des Versuchs:
  - Schicht mit 128 Neuronen
- Schicht zur Konkatenation von Kontext und Anfrage
- Schicht mit 64 Neuronen
- Schicht zur Konkatenation der bisherigen Ausgabe und der Versuchsbeschreibung

Eine Belohnung wird vergeben, wenn das Netz keinen Fehler in der Anfrage verursacht, der Server also einen Statuscode von 200 zurückgibt. Als Kodierung für Buchstaben wird ein *One-Hot Encoding* genutzt. Dabei wird ein Zeichen als Vektor dargestellt, in dem alle Einträge null sind außer demjenigen, der dem Zeichen entspricht, welcher auf eins gesetzt wird. Als Beispiel eines One-Hot Encodings für kleine Buchstaben von a bis z würde b als Vektor

$$b = [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

dargestellt werden. Als Fehlerfunktion wurde kategorische Crossentropy gewählt, welche für Einteilung in Klassen meist genutzt wird. Als Ergebnis gibt das Netz dann immer die prozentuale Wahrscheinlichkeit für das nächste Zeichen aus. Mittels einer *temperature* kann nun entschieden werden, wie hoch die Varianz sein soll.
Das Modell stellt zudem noch eine Möglichkeit für überwachtes Training zur Verfügung, welche dem Netz Rewards verspricht, wenn es die in der Datei angegebenen Strings bei ebenfalls angegebenen Query-Strings und Zuständen erhält. Im Vortrag wurde erwähnt, dass die Entwickler Daten aus der Anwendung *sqlmap* extrahiert haben und für diese Art des Trainings nutzten.

3.1.3 Vergleich bestehender Ansätze

DeepHack und Learn&Fuzz gehen sehr verschiedene Wege mit teils auch unterschiedlichen Zielen. DeepHack nutzt Reinforcement Learning während Learn&Fuzz ein Sequence-to-Sequence Modell verwendet. Reinforcement Learning scheint hier der exotischere Ansatz zu sein, da er üblicherweise für Spiele verwendet wird und nicht für generative neuronale Netze. Dennoch konnte das Netz - so die Autoren - einige Erfolge verbuchen. Der Ansatz des Microsoft Research Teams wird häufig verwendet, zum Beispiel für maschinelle Übersetzung oder Spracherkennung. Dennoch ist das Ziel hier ein völlig anderes: Es sollen PDF-Objekte erstellt werden. Das sind nicht im direkten Sinne Eingaben, wie sie für Input Fuzzing verwendet werden. Auf den Testservern hat kein Nutzer Rechte, um PDF-Dateien zu teilen. Hier ist der allgemeine Ansatz von DeepHack sinnvoller, da er die Möglichkeit bietet das

Netz unproblematisch neu zu trainieren, wenn auf eine spezielle Webapplikation Angriffe gefahren werden sollen. Dafür werden dann keine Änderungen an der Architektur selbst nötig, es kann lediglich helfen, die Payloads für das überwachte Lernen anzupassen.
Gleichheit weisen sie allerdings in einem Punkt auf: Beide Netze nutzen LSTMs, um eine Verbindung innerhalb der Strings zu etablieren. Wie in den Grundlagen bereits erläutert, ist dieser Ansatz nur verständlich.

3.1.4 Relevanz für diese Arbeit

Da diese Arbeit Fuzzing-Payloads erstellen soll, sind beide Ansätze interessante Versuche. Sie können allerdings nicht direkt übernommen werden, da nur bei DeepHack ein offener Quellcode existiert. Das Verzeichnis enthält allerdings keine Trainingsdaten und nutzt das Tensorflow-Framework. Zudem bedarf es weiterer Optimierungen, da etwa das *End-of-Sentence-token* als |-Zeichen festgelegt wurde. Es soll jedoch für den Fuzzer auch möglich sein, dieses Zeichen als Eingabe zu verwenden.
Ziel ist ebenfalls ein Vergleich beider Techniken, was dazu führt, dass beide Ansätze implementiert, trainiert und optimiert werden, um die Qualität der Sequence-to-Sequence Payloads mit denen von Reinforcement Learning zu vergleichen.
Beide Ansätze sollen zunächst überwacht mit bekannten Payloads trainiert und anschließend anhand der verwundbaren Webseiten verbessert werden.

4 Neural Payload

Neural Payload ist der Name des Projektes, welches mit der Hilfe von Deep Learning Payloads erzeugen soll. Diese sollen zum Input Fuzzing von Nutzereingaben dienen und somit im Penetrationtest mögliche Schwachstellen aufzeigen. Es werden die beiden Ansätze aus der Literaturrecherche implementiert, angepasst, verbessert und verglichen: DeepHack, welches Reinforcement Learning zur Stringgenerierung verwendet und Learn&Fuzz, welches den Sequence to Sequence Ansatz verfolgt. Für die Implementierung von DeepHack stand zwar der Sourcecode zur Verfügung, war jedoch in Tensorflow und konnte somit nicht direkt für den Vergleich verwendet werden. Auch einige Designentscheidungen wurden überarbeitet. Beispielsweise steht in DeepHack das Zeichen | für End-of-Sequence, das heißt, es markiert das Ende des Payloads. In vielen Payloads ist dieses Zeichen jedoch essentieller Bestandteil und könnte nicht verwendet werden. Zudem sind in DeepHack alle Payloads, die eine HTTP-Antwort mit Statuscode *200* hervorrufen als Erfolg. Da aber auch valide, nicht bösartige Nutzerangaben immer einen Code *200* hervorrufen, finden sich unter den Payloads zwangsläufig auch gewöhnliche Nutzereingaben, die die große Mehrheit an Eingaben darstellen. Eine Möglichkeit zum Pretraining existiert nicht.
Learn&Fuzz ist ausschließlich als Paper verfügbar. Darum ist hier eine Implementierung durchaus interessant. Zudem werden als Payloads im Paper ausschließlich PDF-Objekte betrachtet. Gewöhnliche Angriffe auf Webseiten gilt es für diesen Ansatz zu untersuchen.

4.1 Reinforcement Learning Neural Payload (RLNP)

4.1.1 Funktionsweise

Das Reinforcement Learning bei RLNP wird wie üblich als eine Art Spiel modelliert. Es wird unterteilt in zwei Abschnitte:

- Überwachtes Pretraining
- Unüberwachter echter Angriff

4.1.1.1 Überwachtes Pretraining

Beim überwachten Pretraining geht es darum, initiale Gewichte für das neuronale Netz zu definieren. Es soll die Struktur von üblichen Payloads erkennen und lernen, sodass später generierte Angriffsstrings auf einer initialen Liste basieren. Dies ist hilfreich, da die künstliche Intelligenz sonst zumindest zu Beginn blind raten muss und die Struktur von Payloads erst dann erschlossen wird. Dieser Prozess kostet bei jeder Anwendung Zeit, die gespart werden könnte.
Das Pretraining erfolgt anhand einer Liste gegebener Payloads. Im Fall dieser Arbeit wurde eine Liste basierend auf Payloads von *IC-Consult* erstellt. Das Netz wird darauf trainiert Payloads zu generieren, die ähnlich den Gegebenen sind. Um dem Netz keine Regelmäßigkeit zu liefern und so Auswendiglernen zu verhindern, muss eine Varianz eingeführt werden. Im Trainingsverlauf wird daher zunächst eine Münze geworfen. Bei Kopfwurf wird ein Payload aus der mitgelieferten Liste gewählt, bei Zahl wird ein gewöhnliches *Spiel* durchgeführt. Beides führt jedoch dazu, dass ein nächster Buchstabe basierend auf dem aktuellen Zustand gewählt wird. Der aktuelle Zustand ist der String aus den in diesem Durchlauf bereits gewählten Buchstaben. Wenn nun also zu Beginn entschieden wurde, dass ein String aus der Liste genutzt werden soll, wird einfach der nächstfolgende Buchstabe für den Zustand gewählt, der zu einem der mitgelieferten Payloads passt. Anderweitig wird - auf der Grundlage eines weiteren Zufalls - entweder ein zufälliger Buchstabe gewählt oder das Netz befragt. Hier sind einige Tuningparameter zu finden. In der Konfiguration kann der Anwender angeben, wie hoch die Wahrscheinlichkeit ist, dass zu Beginn und gegen Ende das Modell befragt wird und selbstverständlich wie lang diese Phase ist. Dies wird berechnet nach

$$\varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end}) * e^{-\frac{steps_i}{steps_{max}}}.$$

Hiermit kann angegeben werden, wie stark der Wert selbst am Ende des Trainings verschmiert wird, sprich ein zufälliger Wert gewählt wird. Dadurch kann eine dauerhafte Exploration gewährleistet werden und es werden neue Payloads ausprobiert. Der neue Zustand, der durch die Konkatenation des bisherigen Zustands und des neuen Buchstaben entsteht, wird nun in eine Liste eingetragen. Anschließend wird auf Spielsieg oder Niederlage geprüft. Ein Sieg besteht daraus, exakt einen String aus der Liste abzubilden und das letzte Zeichen des Strings mit dem End-of-Sequence Zeichen abzuschließen. Eine Niederlage entsteht, wenn es mit dem aktuellen Zustand unmöglich wird einen der Payloads abzubilden, sprich der Zustand kein Präfix für einen String mehr ist. Bei beiden Szenarien wird der Zustand zurückgesetzt und der Gewinn mittels Q-Learning ins Gedächtnis eingetragen. Das bedeutet, der letzte gewählte Zustand und die letzte Aktion bekommen die Sieges- bzw. Niederlagenbelohnung. Der vorletzte Zustand und die Aktion, die zum letzten Zustand führte bekommen $\gamma * Belohnungdesletzten Zustandes$ und so weiter. All diese Tupel aus (Zustand, Aktion, nächster Zustand, Belohnung) werden nun in das Gedächtnis eingetragen. Es ist eine zyklische Liste, welche immer die *length* neuesten Trainingsdaten enthält. Aus diesem Gedächtnis werden bei jedem Zustand $batch_size$-viele zufällige Einheiten gewählt und anhand ihrer Daten trainiert. Dazu später mehr.

Die Entscheidung, hier einen Münzwurf einzuführen mag ungewöhnlich klingen, führt jedoch überhaupt erst zu einem Lerneffekt. Ohne das aktive Nutzen von garantierten Siegen ist die Wahrscheinlichkeit, einen Sieg zu erlangen

$$P(x) = \frac{1}{\frac{n^l}{m}}$$

mit n als Anzahl der Zeichen, welche auf 96 geschätzt werden können, l als durchschnittliche Länge der Payloads und m als Anzahl der Payloads. Bei gegebenen Trainingsdaten führt das zu

$$P(x) \approx \frac{1}{\frac{96^{24}}{356}} < 10^{-45}.$$

Das bedeutet, dass nach einer vernünftigen Trainingszeit von maximal 24 Stunden mit hoher Wahrscheinlichkeit alle Trainingsbeispiele Niederlagen sind. Damit sind ausschließlich Negativbeispiele für das Training vorhanden gewesen und das Netz wird immer bestraft. Es versucht also diese Bestrafung zu minimieren, was exakt dann geschieht, wenn es als erstes Zeichen End-of-Sequence ausgibt, da dann nur ein Zeichen bestraft werden kann. Um dies zu vermeiden, müssen Positivbeispiele vorhanden sein, was durch den Münzwurf erreicht wird. Eine probabilistische Wahl anstelle einer deterministischen ist dennoch von Vorteil, da sich so das Netz nicht daran gewöhnt, in einem Zug zu verlieren, anschließend z

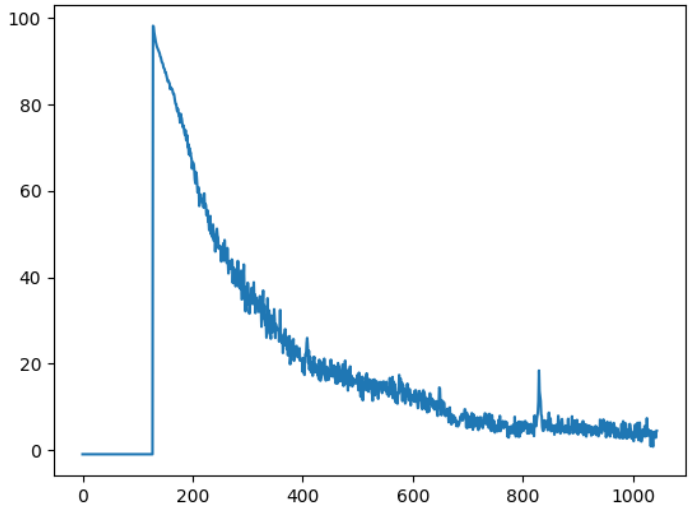

Abbildung 4.1: Fehlerkurve des Netzes ohne Gleichverteilung der Siege und Niederlagen - nur leere Strings werden produziert

4.1.1.2 Unüberwachtes Training

Das unüberwachte Training erfordert nun keine Liste mit gegebenen Payloads mehr, sondern Zugriff auf eine Website. Auf sie soll ein Angriff durchgeführt werden. Dazu wird ebenfalls ein Angriffsdictionary benötigt, in welchem für einen POST-Befehl alle Daten enthalten sind. Ein Feld der Daten wird durch *ZAP* markiert. Dies ist der Teil, an dem der Payload eingesetzt werden soll. Wenn nun also beispielsweise die Website *http://localhost/index.php* mit dem Wörterbuch *{user:admin, password:ZAP}* angegriffen werden soll, so wird bei *password* das *ZAP* in jedem Durchgang durch den Payload - zB *||6* ersetzt und an die Website gesendet. Um nun eine fundierte Entscheidung treffen zu können, ob der Angriff ein Erfolg oder eine Niederlage war, muss ein Evaluator zur Verfügung stehen. Dazu soll Kapitel 5 dienen. Für die Entwicklung hier war allerdings auch ein einfacherer Evaluator ausreichend, welcher Status-Codes anders als 200 als verloren wertet. Antworten, deren Inhalt sich von der ersten Anfrage mit einer normalen Nutzereingabe (also etwa dem grundlegenden, unveränderten Wörterbuch *{user:admin, password:ZAP}*) unterscheidet werden als Gewinn gewertet und Antworten, die gleich sind werden als noch nicht beendet gewertet. Dies ist möglich, da das Projekt hauptsächlich an den Challenges des Autors ausgewertet wurde, bei welchem dies immer auch die Wahrheit abbildet. So wird etwa bei der hier verwendeten Challenge eine Schwachstelle im POST-Parameter *password* erwartet. Gibt man ein gewöhnliches Passwort ein, wird nur ein String, der das falsche Passwort anzeigt, ausgegeben. Das heißt, der Evaluator kann nicht einfach die Antwort aus der normalen Websiteanfrage ohne die POST-Parameter als Standardantwort werten, da sonst der String, der den Fehlschlag anzeigt, nicht ausgegeben wird. Es muss also eine Anfrage mit POST-Parameter gestartet werden, die vermutlich keinen Gewinn darstellen gesendet werden. Hierzu eignet sich die grundlegende Liste, wie oben erwähnt. Durch dieses Verfahren werden jedoch auch möglich Fehlermeldungen als Gewinn gewertet. Das ist aber auch gewollt, da solche Fehlermeldungen meist viele Informationen enthalten, die für einen Angriff wertvoll sind. Ansonsten läuft das Verfahren wie im überwachten Training ab.

4.1.2 Architektur

Das Netz selbst ist rekurrent, da der Zustand aus mehreren Buchstaben besteht, von denen jeder Informationen enthält, welcher Buchstabe als nächstes gewählt werden soll. Die Anzahl an Schichten sowie die Anzahl an Ausgabeneuronen der rekurrenten Schichten sind variabel und in der Konfigurationsdatei einstellbar. Nach jeder aus der letzten rekurrenten Schicht ist zudem eine *Dropout*-Schicht mit konfigurierbarem Dropout. Diese lässt $100 * x\%$ der Informationen einfach verfallen. Das führt dazu, dass Informationen im Netz redundant gelernt werden und im Gesamten zu einer besseren Generalisierung. Auch die Verwendung der rekurrenten Technologie ist konfigurierbar. Sowohl gewöhnliche rekurrente Schichten als auch LSTMs sind verfügbar (siehe auch 4.3).

Als Eingabe wird ein Tensor erwartet, der den aktuellen Zustand, also den bisher generierten String, enthält. Die Ausgabe besteht ebenfalls aus einem Tensor, der die erwartete Belohnung für jede Aktion

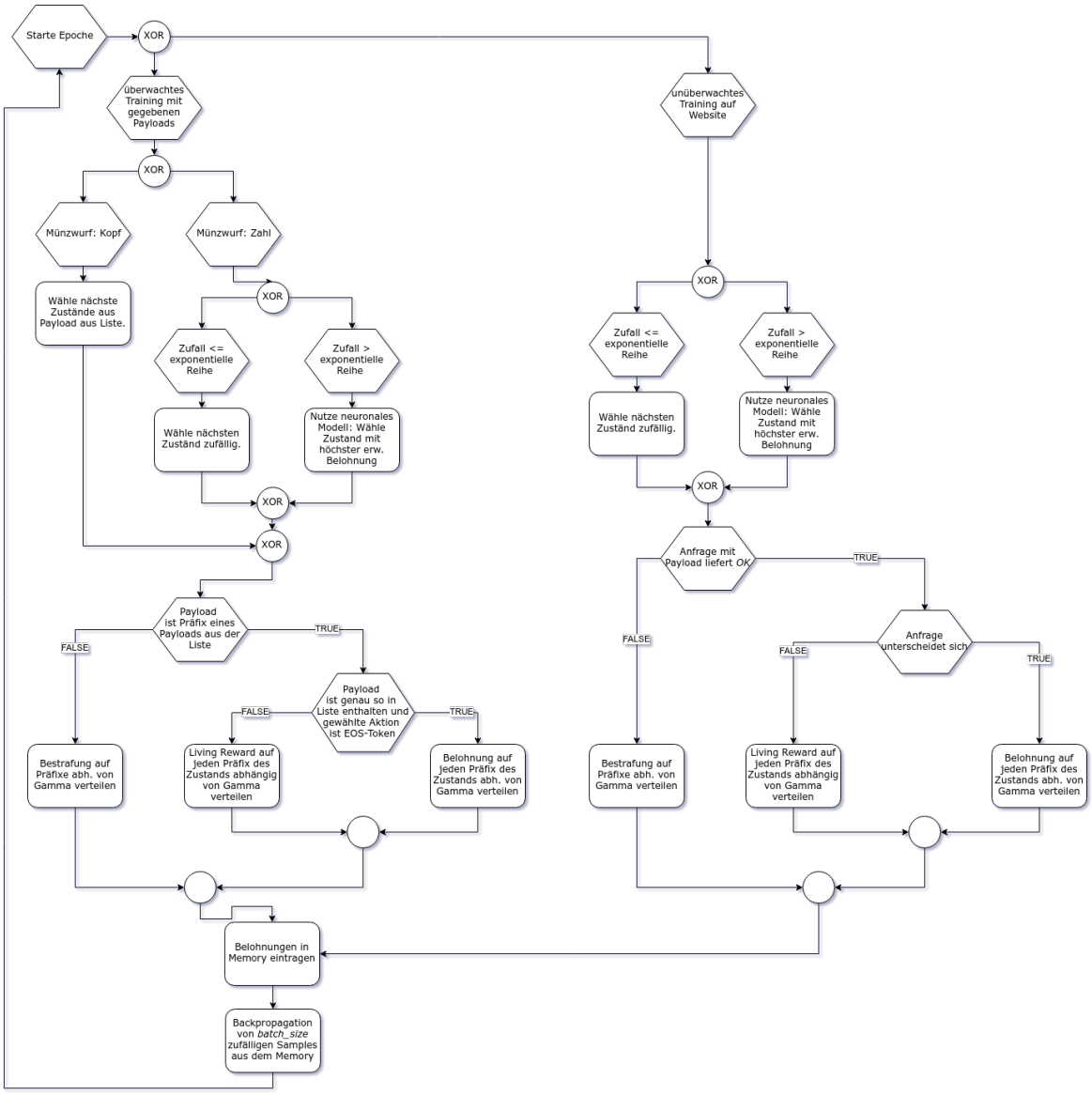

Abbildung 4.2: Flussdiagramm zu RLNP

enthält. Aktionen stellen dabei den nächsten zu wählenden Buchstaben dar. Nun kann einfach das Maximum aus diesem Tensor gewählt werden und so wird der Buchstabe gewählt, welcher die höchste Belohnung verspricht, wie in Abb. 4.4.

4.1.3 Lernen

Backpropagation wird innerhalb der Hauptschleife des Trainingsprozesses mit den Daten aus dem Gedächtnis ausgeführt. Jedes der Beispiele besteht aus (Zustand s, Aktion a, nächster Zustand n, Belohnung r). Nun kann das Modell für jedes Trainingsbeispiel mit Index *i* befragt werden, welche

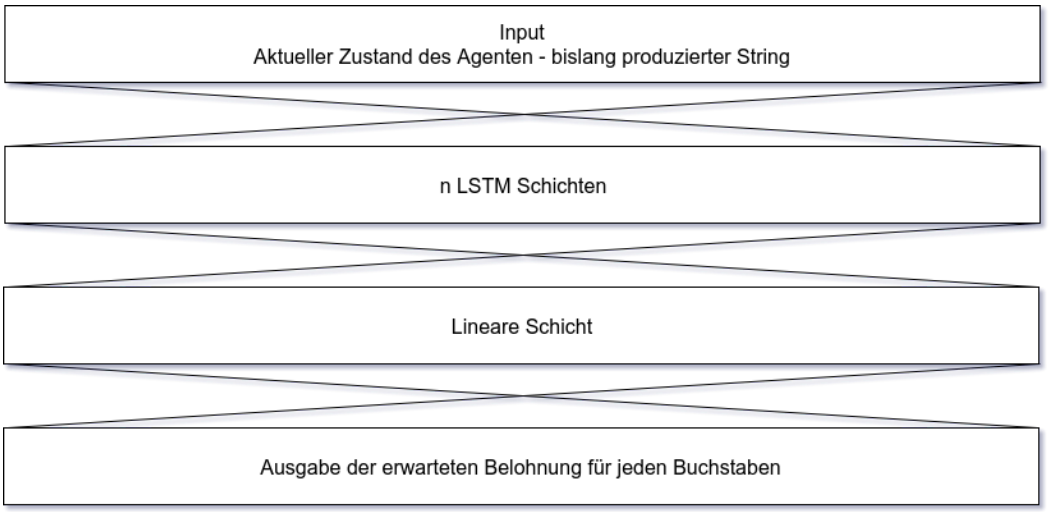

Abbildung 4.3: Die Architektur des Netzes von RLNP

Belohnung r_i für den Zustand a_i bei Eingabe von s_i erwartet wird. Der Fehler kann anhand der tatsächlichen Belohnung im Trainingsbeispiel berechnet und mit dem Backpropagation-Algorithmus propagiert werden. Als Fehlerberechnungfunktion wurde der Mean-Squared Error oder die euklidische Distanzfunktion gewählt, da die Belohnungen nicht zueinander in Beziehung stehen.

4.1.4 Weitere Optimierungen

Eine adaptive Lernrate nach der Berechnung von RMSProp [Hi] aus der Vorlesung von Geoffrey Hinton wurde verwendet, um den Lerneffekt am Anfang zu verstärken und gegen Ende zu verhindern, dass gar nichts mehr gelernt wird und alte Informationen komplett überschrieben werden.
Außerdem wurde ein Living Reward eingeführt. Dieser führt dazu, dass Strings, die weder zu Niederlage noch zu Sieg führen, belohnt oder bestraft werden können. Technisch wird einfach in jedem Durchlauf der aktuelle Zustand und der Living Reward nach Q-Learning im Gedächtnis eingetragen. Sinnvoll ist dies in beiden Szenarien - sowohl beim überwachten als auch beim unüberwachten Lernen. Beim überwachten Training erwies es sich als nützlich, eine tatsächliche Belohnung zu vergeben. Dadurch werden alle Aktionen belohnt, die nicht zu einem Verlust führen. Da aber anhand einer Liste von Payloads verglichen wird, wird somit jede Ausgabe belohnt, die zu einem Präfix eines Elements der Liste führt. Beim unüberwachten Lernen war es effektiver, eine Bestrafung als Living Reward zu vergeben. Dadurch werden Eingaben, die zu keiner Veränderung auf der Website führen leicht bestraft. Die Strings werden kürzer und gewöhnliche Nutzereingaben wie Namen oder übliche Passwörter werden vermieden.

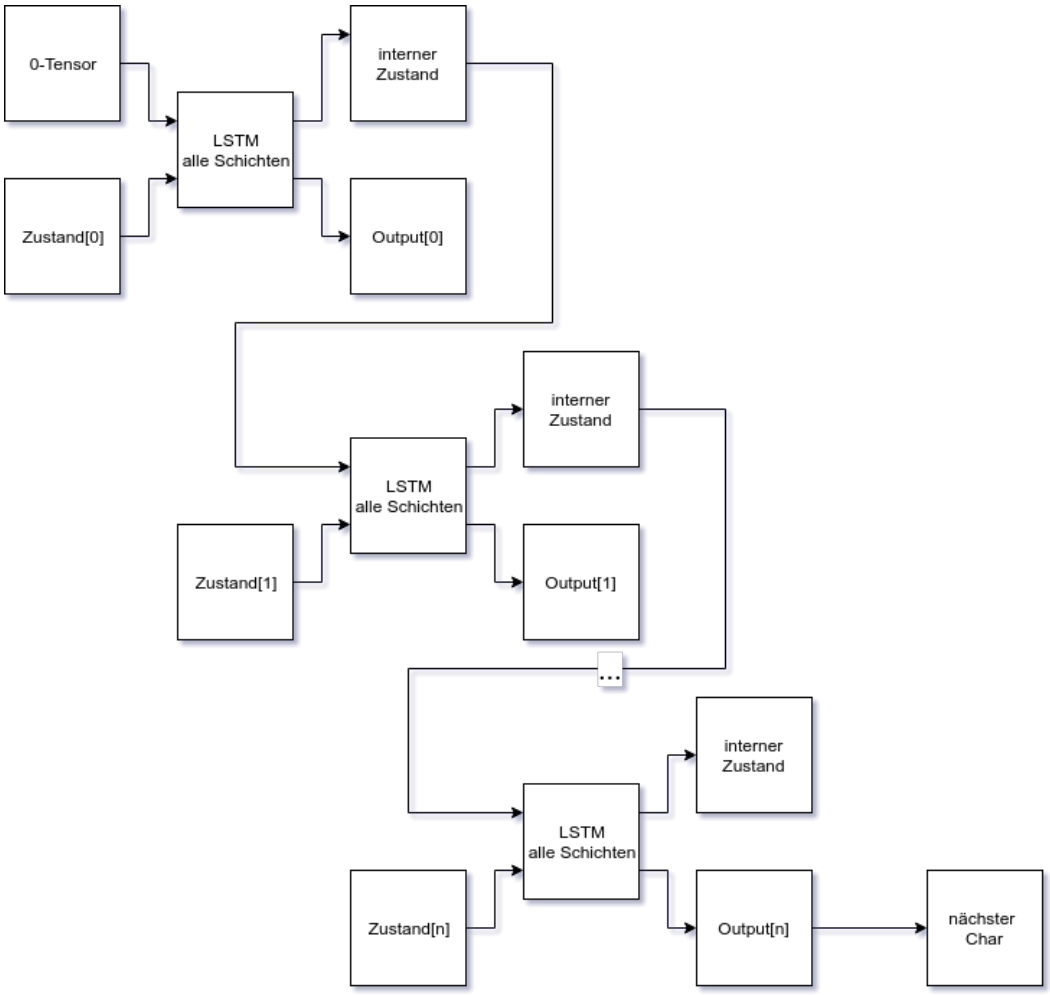

Abbildung 4.4: Funktion der Stringgenerierung mit LSTMs

4.1.5 Evaluation

Das Netz wurde hier anhand einer Hacking Challenge des Autors für SQL-Injection getestet. Mit drei LSTM-Schichten mit je 256 Neuronen, keinem Dropout und einem γ-Faktor von 0.99 wurde das Netz anhand der Payloadliste von IC-Consult vortrainiert. Dieser Vorgang lief etwa zwölf Stunden auf einer Nvidia Geforce GTX1080 mit 2560 Cuda-Kernen und acht Gigabyte Grafikspeicher. Anschließend wurde der unüberwachte Vorgang gestartet und es wurden im Grund direkt Erfolge erzielt, da bei der Challenge die Eingabe von $'$ schon zu einem PHH-Fehler führte. Dieser gibt jedoch auch Aufschluss auf die Angriffsfläche und ist daher ein wertvoller Erfolg. Die Challenge enthielt die Schwachstelle durch einfache, nicht vorbereitete Statements in PHP:

```
1 $sql = "SELECT id, user, password FROM users WHERE password = '" .
      $_POST["password"] . "';";
2 $result = mysqli_query($conn, $sql);
```

Listing 4.1: Angriffsfläche der Evaluationschallenge

Es ist offensichtlich, dass Passwörter wie *1' OR '1'='1* zu einem Erfolg führen. Jedoch liefert auch der Payload *'* einen Fehler, da das Statement nicht ordentlich beendet wurde und je nach Servereinstellungen kann hier die Zeile im Code sogar offenbart werden. Dies lässt auf diese Lücke schließen.
In den weiteren Trainingsdurchläufen sind einige Statistiken entstanden, die für die spätere Nutzung relevant sein könnten. Das Training eines Beispiels dauerte im Durchschnitt und je nach Konfiguration auf dem beschriebenen System zwischen 0.05 und 1 Sekunde. Es werden mindestens 50000 Trainingseinheiten - berechnet aus $Batchgre * Epochen$ - empfohlen. Die Fehler variieren selbstverständlich stark, da in der Konfiguration der *base_reward* eingestellt werden kann. Mit diesem lässt sich dem Netz sagen, wie hoch die Belohnung bei einem Sieg bzw. die Bestrafung bei einer Niederlage vor Anrechnung des γ-Faktors sein soll. Je nach Belohnungshöhe unterscheiden sich auch die Ausgaben des Netzes. Zudem sind eine hohe Anzahl an Payloads enthalten, die ebenfalls in Länge und Art variieren. Daher entstehen nur selten bekannte Zustände und es kann durch die Anpassung mit Backpropagation zur Änderung von anderen Gewichten kommen. Dadurch entstehen Ausreißer.
Des weiteren ist die Konfiguration des Netzes wichtig. Bei der Evaluation eines neuronalen Netzes kommt hierbei normalerweise ein Trainings- und ein Validierungsset zum Einsatz. Das Trainingsset wird zum Trainieren des Netzes verwendet, während das Validierungsset ausschließlich zum Evaluieren der Generalisierungsfähigkeit des Netzes verwendet wird. Die ist im Fall dieses Projektes nicht möglich, da selbst das Trainingsset sehr gering ausfällt. Es besteht aus gerade mal 231 Beispielen für SQL-Injection bzw. aus 357 für den vollen Traininssatz und wird vollständig für das Training gebraucht. Da allerdings ähnliche Beispiele generiert werden sollen, macht ein gewisser Fehler nichts aus.
Sieht man sich die Fehlerkurven der verschiedenen Trainingsdurchläufe an, so wird klar, welchen Einfluss die einzelnen Konfigurationsparameter haben. In Abb. 4.5 ist die Kurven für einen normalen Trainingsdurchlauf angegeben. In diesem wird das Netz insgesamt 200 mal darauf angesetzt, einen Payload zu generieren. Es wurden acht LSTM-Schichten genutzt, jede mit 256 Neuronen. Ein Dropout wurde nicht gesetzt. Diese Konfiguration wird die Referenz für die Evaluation sein. Zudem wurde der RMSProp-Optimizer verwendet. Es kann sehr deutlich gesehen werden, dass der Fehler zunächst in die Höhe schnellt. Das ist die Anpassung an die gegebenen Strings. Anschließend fällt er rapide wieder ab, wodurch ein Training erkennbar wird. Ein weiterer Anstieg kann verzeichnet werden, da ein komplett unterschiedlicher Payload genutzt wurde. Hier zeigt sich auch die größte Schwierigkeit des Datensatzes: Die Daten haben eine sehr geringe Korrelation. Beispielsweise hängen *NULL UNION ALL SELECT user,pass, FROM user_db WHERE user LIKE '%admin%/** und *a' or 1=1–* weder in Länge noch in Inhalt wesentlich zusammen. Aus diesem Grund wird jedes Mal, wenn ein signifikant anderer Payload zum ersten Mal eingelesen und trainiert wird, ein Spike angezeigt, der

einen deutlichen Anstieg des Fehlers indiziert.
Um der Menge an Elementen Herr zu werden, wurde die Kapazität des Netzes erhöht auf vier Schichten mit je 1024 Neuronen. Dieses Modell wurde über Nacht auch für 200 Epochen trainiert. Hier wurde jedoch ein Dropout von 20% gewählt, um mit einer möglichen Streuung in den Daten zurecht zu kommen. Die Fehlerkurve ist in Abb.4.6 zu finden. Reduziert man den Trainingsdatensatz auf nur

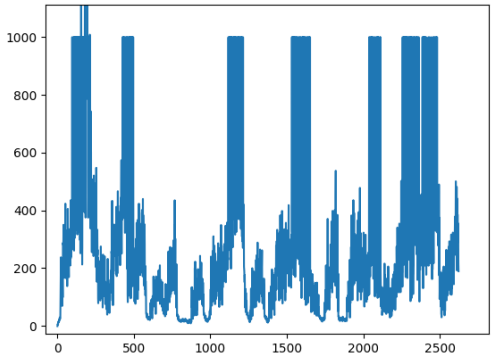

Abbildung 4.5: Standard Training mit 256 hidden Neuronen, 8 Schichten, keinem Dropout und 200 Epochen im LSTM-Netz mit den SQL-Injection-Daten

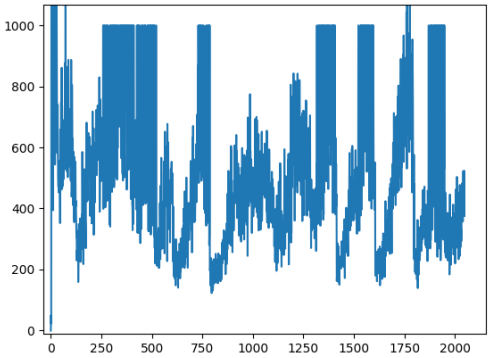

Abbildung 4.6: Training mit 1024 hidden Neuronen, 4 Schichten, 20% Dropout und 200 Epochen im LSTM-Netz mit den SQL-Injection-Daten

ein Trainingsdatum, so verschwinden die Spikes, wie in Abb. 4.7 zu sehen ist. Jedoch nur, solange das Netz ausschließlich gewinnt. Sobald wieder zufällige Daten generiert werden, um die Exploration zu fördern, erscheinen wiederum einige Spikes (siehe Abb. 4.8). Diese trainieren die Verluste des Netzes und sind somit nicht unbedingt negativ zu betrachten. Sie erscheinen nur so immens, da die Differenz für einen Sieg und eine Niederlage so groß ist. Beispielsweise generiert das Zeichen *End of Sequence* bei Zustand *a' or 1=1–* eine Belohnung von 100 (hier als BaseReward angenommen), jedoch dasselbe Zeichen bei Zustand *a' or 1=1-* eine Belohnung von -100. Dass das Netz überhaupt lernt, ist schnell erkennbar: Liefert man eine randomisierte Belohnung im Intervall von $[-BaseReward, BaseReward]$, so ist der Fehler konstant hoch, wie Abbildung 4.9 zeigt.

Ein weiterer Faktor ist der Optimizer. Wie sich im Test gezeigt hat, liefert RMSProp die besseren Ergebnisse vor Adam, was aus den Grundlagen vermutbar war. Das Ergebnis lässt sich in Abb. 4.10 sehen. Eine Verbesserung ergab jedoch die Nutzung eines höheren Weight Decays, einem Feature von RMSProp. Dabei werden die Gewichte mit einem Faktor kleiner eins nach jedem Trainingsdurchlauf multipliziert. Das führt dazu, dass sie nicht zu sehr in die Höhe schießen und gilt als eine Art Regulierungsterm [KK]. Hierdurch kann bessere Generalisierung erreicht werden. Das Ergebnis ist eine weniger verrauschte Lernkurve in Abb. 4.11.

Interessante Ergebnisse brachte jedoch die Modifikation der Größe des Gedächtnisses. Im Standardtrainingsdurchlauf war das Gedächtnis etwa so groß, wie die Batchgröße, sodass nahezu jedes

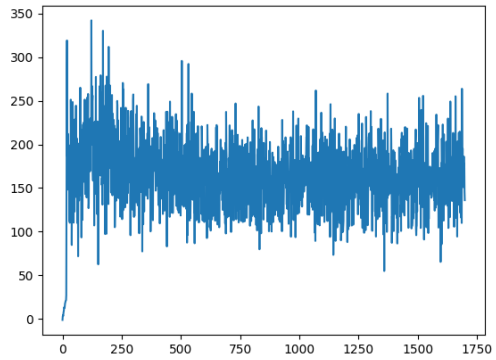

Abbildung 4.7: Fehlerkurve mit nur einem Trainingsdatum und dauerhaft garantiertem Sieg

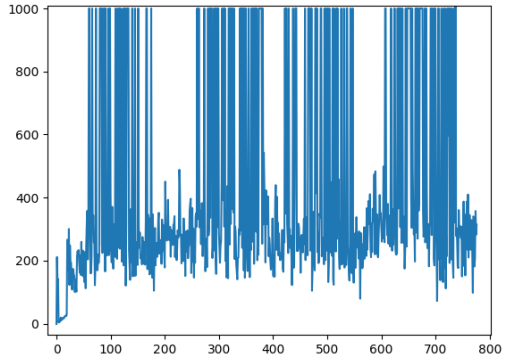

Abbildung 4.8: Fehlerkurve mit nur einem Trainingsdatum

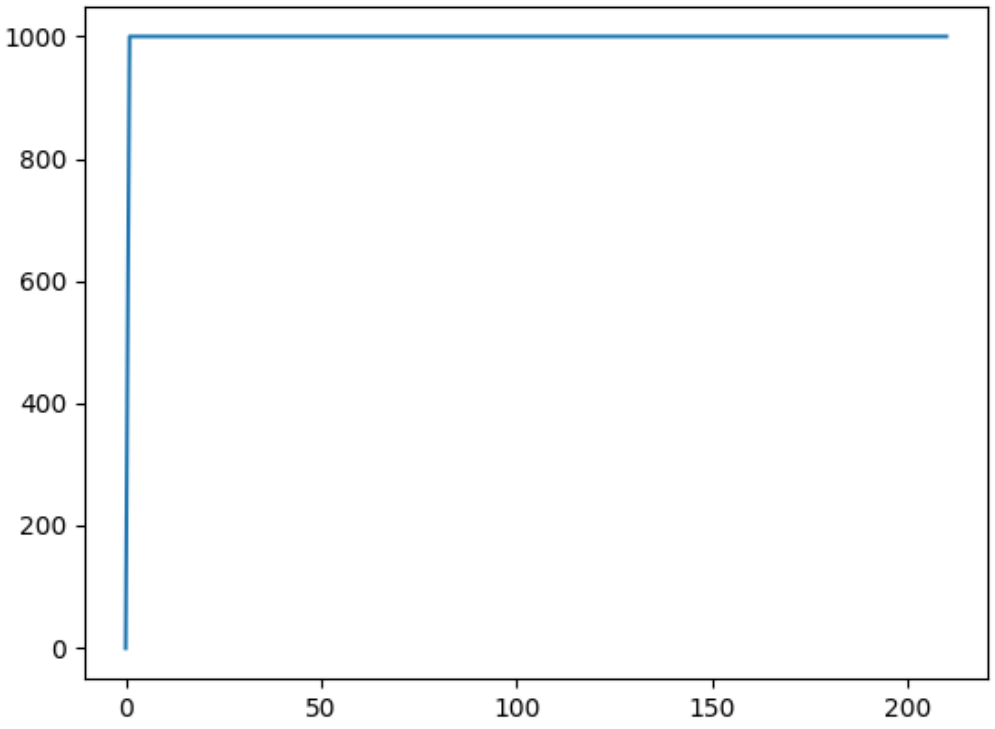

Abbildung 4.9: Training mit randomisierter Belohnung

Trainingsbeispiel auch trainiert wurde. Eine niedrigere Größe als das ist nicht möglich, um Overfitting zu verhindern. Eine höhere Kapazität führt jedoch zu einer erhöhten Anzahl der Spikes, wie in Abb.

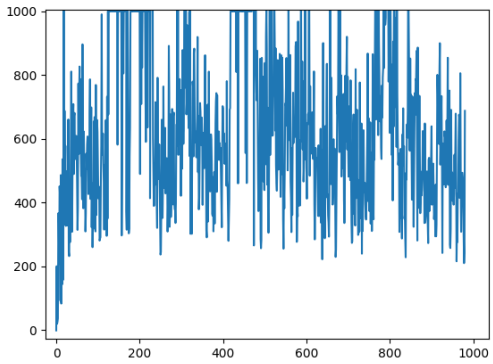

Abbildung 4.10: Fehlerkurve beim Training mit dem Adam-Optimizer

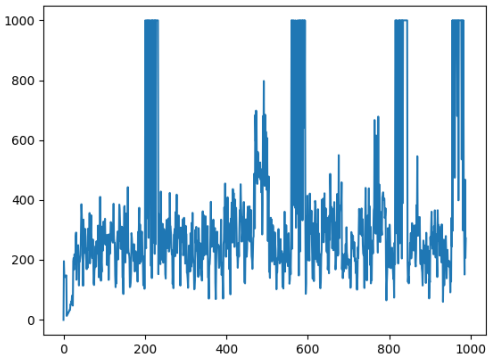

Abbildung 4.11: Fehlerkurve beim Training mit einem Weight Decay von 0.1 im RMSProp

4.12 dargestellt. Dies ist vermutlich auf zu wenige Trainingsepisoden zurückzuführen und könnte sich reduzieren.

Alles in allem kann vermutet werden, dass ein initiales Training zwar sehr wohl möglich ist, jedoch

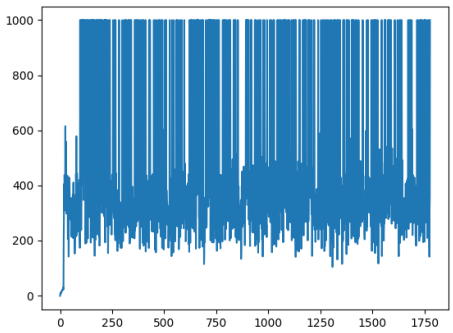

Abbildung 4.12: Training mit hoher Kapazität

in einigen Jahren erst mit einem hinreichend großem Netz und ausreichend vielen Epochen trainiert werden kann. Leider ist es dem Autor nicht möglich gewesen, das Netz lange genug zu trainieren, da schon 200 Epochen mit einem großen Netz 24 Stunden Training erforderten und auch das Netz konnte nicht auf eine Größe mit mehr als 8192 Neuronen im Gesamten gestellt werden, da sonst der Speicher der Grafikkarte zu gering war. Die Beobachtung liegt jedoch nahe, dass sowohl 200 Epochen für die Trainingsmenge nicht ausreichend sind, als auch, dass die Größe nicht genügt oder zumindest verbessert werden kann.

4.1.5.1 Austauschformat

Das Modell kann selbstverständlich auch gespeichert werden. Der Konstruktor erlaubt den optionalen Parameter *filename*. Falls dieser gesetzt ist, wird das Modell nach jeder Epoche an den entsprechenden Ort gespeichert. Zudem wird zu Beginn auch überprüft, ob an diesem Ort bereits ein Modell liegt. Falls ja, wird es geladen. Das Austauschformat beinhaltet die Netzstruktur sowie die erlernten Gewichte. Dadurch kann der Lernfortschritt vollständig übernommen werden, da sich in den Gewichten alle Informationen wiederfinden. Besonders für das Pretraining ist dieser Schritt essentiell, da sonst dieses jedes Mal von vorne ausgeführt werden müsste. Sinnvoll ist es, das Pretraining zu speichern und dieses Modell für jeden tatsächlichen Angriff zu übernehmen. Es muss dafür eine Kopie des Modells angelegt werden, damit die Gewichte nicht vom tatsächlichen Angriff überschrieben werden. Das ist deshalb nicht empfohlen, da sich die Gewichte für jeden Angriff unterscheiden, die erlernten Gewichte aus dem Pretraining jedoch eine initiale Allgemeinwissensbasis darstellen. Dennoch soll die Funktion zum Speichern während des Angriffs bestehen bleiben, da dieser unter Umständen auch lange dauern kann. So bleibt immer ein Backup bestehen.

4.2 Sequence-to-Sequence Neural Payload

Das Sequence-to-Sequence Netz, kurz, seq2seq, zielt darauf die Payloads im Vorfeld zu erstellen. Sie können anschließend völlig unabhängig verwendet werden, sei es im Neural Evaluator oder um die Sammlung in einem bewährten Fuzzingprozess zu vervollständigen. Der Ansatz ähnelt dem des Microsoft-Researchteams in [PG17], soll jedoch nicht auf den sehr speziellen Zweck von PDF-Objekten reduziert werden, sondern soll bösartige Benutzereingaben nachahmen. Für dieses Projekt steht keinerlei Code zur Verfügung, ebenso auch keine Parameterkonfiguration.

4.2.1 Funktionsweise

Das fertig trainierte Netz soll anhand des ersten gegebenen Buchstaben den darauf folgenden bestimmen können. Ebenso soll es jedoch den internen Zustand nach dieser Iteration ausgeben. Dadurch kann der Agent iterativ neue Buchstaben anhand des aktuellen Zustands produzieren, initiiert durch den gegebenen ersten Buchstaben. Das Ergebnis ist ein vollständiger Payload-String.
Dieser Prozess gleicht einem Encoder-Decoder Modell über die bereits produzierte Sequenz, um auf dieser das sequence-to-sequence-Verfahren anzuwenden.

Durch dieses Verfahren werden nach einem hinreichend langen Trainingsprozess Strings generiert, die der Masse derer im Trainingsset ähnlich sind. Somit können zum Beispiel aus einer recht geringen Menge an Payloadstrings eine größere Menge generiert werden oder für eine spezielle Aufgabe, die ausgiebiger getestet werden soll, neue Strings erstellt werden.

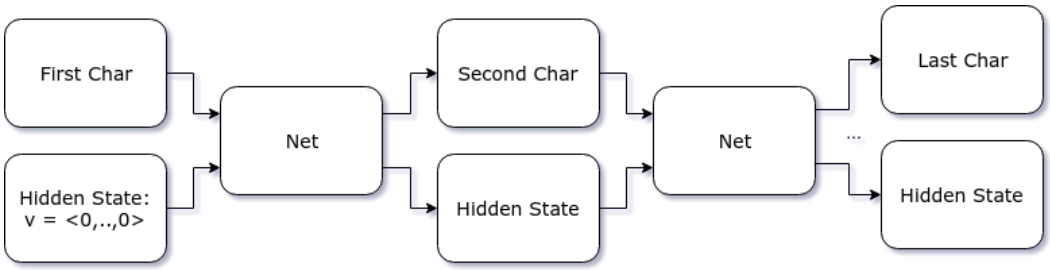

Abbildung 4.13: Funktionsweise des Sequence-to-Sequence Models

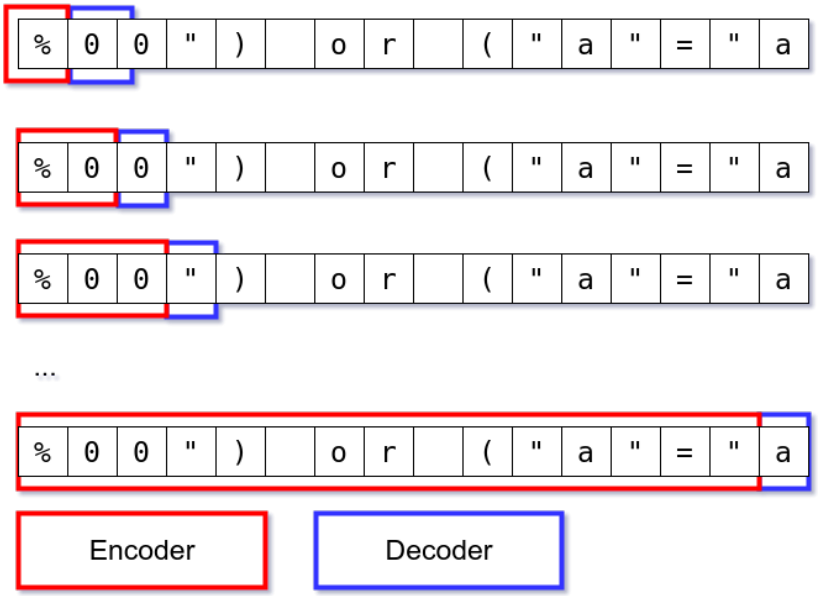

Abbildung 4.14: Encoder-Decoder Prozess des Modells

4.2.2 Architektur

Das Netz selbst ist simpel aufgebaut. Es besteht aus einer Variablen Anzahl an versteckten Schichten. Diese sind aus LSTMs aufgebaut, um die Probleme des Vanishing Gradients zu verhindern und dennoch die Beziehung zwischen Buchstaben zu enthalten. Anschließend ist eine Dropoutschicht angebracht mit einer variablen Menge an Verlust. Das soll das Netz zu einer besseren Generalisierung zwingen. Zuletzt wird ein Softmax auf der Ausgabe berechnet, wodurch garantiert wird, dass die Summe über alle Werte gleich eins ist. Dies geschieht mit

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{i=1}^{|z|} e^{z_k}} for j = 1, ..., |z|$$

Das führt dazu, dass extreme Werte vermieden werden und gleichzeitig eine prozentuale Konfidenz des Netzes für jeden Buchstaben ausgegeben wird. Die LSTMs geben jedoch noch zusätzlich ihren

geheimen Zustand aus, welcher auch in der finalen Ausgabe enthalten ist.
Um nun das Netz iterativ zu nutzen, wird zunächst der erste Buchstabe ins Netz eingegeben. Da jedoch auch der geheime Zustand für die versteckten Schichten mit übergeben werden muss, wird er initialisiert mit dem null-Vektor. Das Netz gibt anschließend das Ergebnis der Berechnung zurück, ebenso wie den internen Zustand. Nun kann der zweite Buchstabe und der eben zurückgegebene Wert eingegeben werden. Dieser Vorgang wird wiederholt bis das Netz entweder *End-of-Sequence* zurückgibt oder aber eine vom Nutzer festgelegte Maximallänge erreicht wurde. Dadurch ist ein vollständig neuer String entstanden.

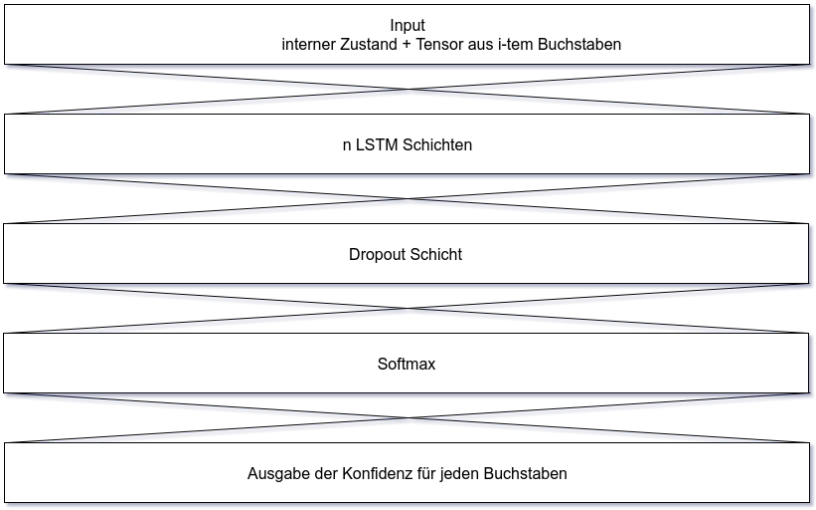

Abbildung 4.15: Architektur des Sequence-to-Sequence Modells

4.2.3 Lernen

Um das Netz zu trainieren, muss eine Liste an bereits bekannten Payloads vorliegen. Eine Epoche bezeichnet hier eine Iteration durch all diese Payloads. Für jeden Payload wird zunächst das *End-of-Sequence*-Zeichen angehängt. Der erste Buchstabe wird dann als gegeben angenommen. Mit ihm wird nach dem Verfahren, beschrieben in 4.2.2, ein Payload erzeugt. Für jeden der hier erzeugten Buchstaben wird der Unterschied zu dem Payload aus der Liste berechnet. Dies geschieht nach der Formel des *Negative Log-Likelihoods*, welche in der Literatur fast immer mit dem Softmax kombiniert wird:

$$L(y) = -log(y)$$

wobei dies für alle korrekten Klassen aufsummiert wird. In dem Fall dieses Netzes existiert hier nur eine Klasse, welche korrekt war, nämlich die des gegebenen Buchstaben. Wenn man sich den Graphen für die Funktion ansieht, welcher in Abb. 4.16 zu finden ist, wird schnell klar, dass Werte gegen null gegen unendlich laufen. Minimiert wird der Fehler hingegen, wenn die Werte höher sind. Höhere Werte entsprechen jedoch ebenfalls einer höheren Konfidenz, die genau das ist, was das Netz für

die korrekte Klasse ausgeben soll. Eine Konfidenz von 100%, also eine absolute Sicherheit, würde hier den Fehler minimieren. Dieser Fehler wird nun propagiert und die Gewichte werden mit einer

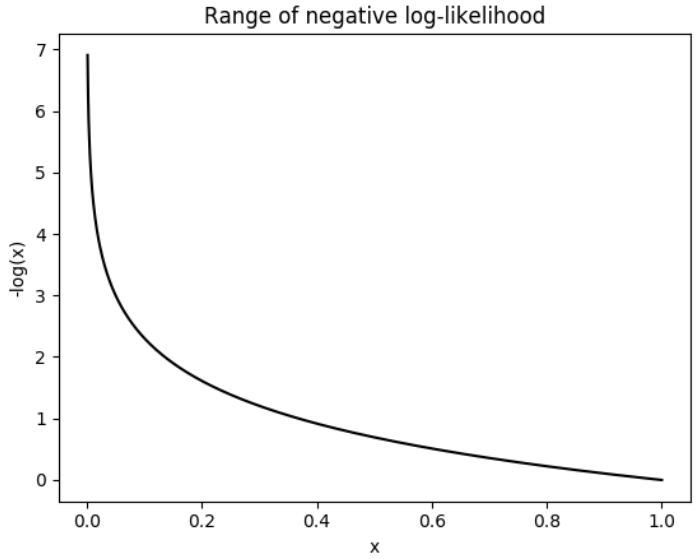

Abbildung 4.16: Verlauf der NLL-Funktion

ebenfalls vom Nutzer festzulegenden Lernrate verbessert.

4.2.4 Evaluation

Die Tests wurden auf zwei unabhängigen Versionen getestet. Die erste Version verfolgte einen anderen Ansatz und sollte einen Autoencoder nutzen, um ähnliche Payloads zu generieren. Die zweite Version ist oben beschrieben.
Der Autoencoder encodiert zunächst die Eingabedaten mit einem LSTM-Netzwerk in einen einzelnen Tensor, um diesen anschließend wieder so ähnlich wie möglich auszugeben. Dies geschieht über einen Decoder, der den vom Encoder ausgegebenen Tensor ebenfalls über ein LSTM-Netz wieder in einen String umwandelt. Dieser Prozess ist in den Abbildungen 4.17 verdeutlicht. Der übrige Prozess gleicht dem der zweiten Version.
Die Ergebnisse waren wider Erwarten sehr ernüchternd, was auch der Grund für das Erstellen der zweiten Version war. Im Test hat das Netz ausschließlich eine variable Anzahl an *a*s ausgegeben. Der Fehler war dabei zwar konstant hoch mit Werten zwischen 25 und 60 für den vollständigen String, jedoch waren die Gradienten für die Schichten des Netzes permanent null. Dadurch waren auch die Veränderungen der Gewichte nicht existent. Es wird vermutet, dass ein Fehler im PyTorch-Framework dazu geführt hat. Es wäre sinnvoll, diesen Ansatz in einer weiterführenden Arbeit zu untersuchen.
In der zweiten Version wurde im Vergleich immer nur ein Buchstabe generiert, jedoch waren für diesen auch nur die Informationen aller bisherigen Buchstaben im versteckten Zustand und im Zellenzustand

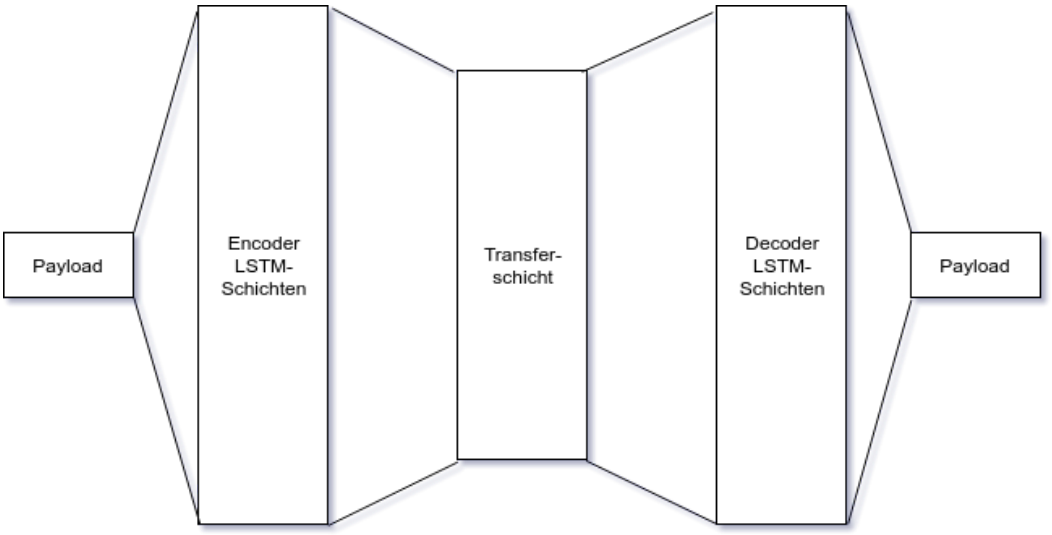

Abbildung 4.17: Struktur des Autoencoders

zusammengefasst. Dies führte dazu, dass das Netz einzelne Payloads sehr akkurat nachbilden konnte. In Abbildung 4.18 ist die Fehlerkurve bei insgesamt 1200 Neuronen, verteilt über 3 Schichten zu sehen. Ein Sampling anhand dieses trainierten Modells resultierte in dem Originaltext.

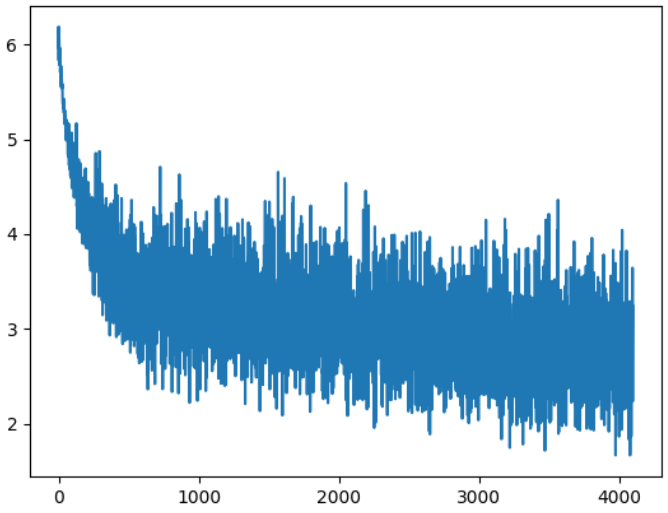

Abbildung 4.18: Fehlerkurve bei einem Training über 4100 Epochen mit 400 versteckten Neuronen in jeder der 3 Schichten und einem Dropout von 30%

Wird nun aber die Anzahl an Payloads erhöht, so verschwimmt auch das Training. Die Trainingskurve zeigt mehr Fehler auf und die generierten Payloads machen einen sehr zufälligen Eindruck oder sind

gänzlich leer. Um nun die Daten zu bereinigen, wurden einige der Payloads entfernt, um so die Varianz zu verringern. Tatsächlich hat dies die Resultate wieder verbessert. Es wird somit empfohlen, einige ähnliche Payloads für das Training zu kombinieren und dadurch mehrere Modelle zu pflegen. Auch kann der Trainingsprozess meist bereits ab etwa der zehnten Epoche verwendet werden. Die Varianz in den Payloads ist zwar zu Beginn noch deutlich höher als später, dient aber auch dem Input Fuzzing. So können nach jeder Trainingsepoche neue Payloads generiert werden um an die Zielwebsite geschickt zu werden. Das Training kann somit auf beliebige Genauigkeit und Dauer durchgeführt werden. So kann beispielsweise eine neue finale Version ausführlicher getestet werden als ein Nightly-Build.

5 Neural Evaluator - Analyse von Verwundbarkeit des Quelltextes

5.1 Ziel

5.2 Funktionsweise

5.3 Architektur

5.4 Setup und Konfiguration

5.5 Prototyp des Netzes

5.6 Trainingsdaten

5.6.1 Trainings- und Validierungsset

5.7 Evaluation und Verbesserung

5.7.1 erste Iteration

5.7.1.1 Evaluierung

5.7.1.2 Verbesserungen

6 Visualisierung durch eine RESTful-Schnittstelle

6.1 Inhaltliche Eingrenzung

6.2 Einarbeitungsphase

6.3 Zusammensetzung des Teams

6.4 Aufgabenverteilung

6.5 Ergebnisse

7 Arbeitsphase bei IC-Consult

In diesem Abschnitt soll die Zusammenarbeit mit IC-Consult beschrieben werden, wo auch der Einsatz der Software zumindest teilweise stattfindet. Hierdurch wird die praxisnahe Forschung gefördert, sodass auch eine einsatzfähige Software entsteht, die sinnvoll genutzt werden kann.
Da jedoch nicht klar ist, welche Ergebnisse zu erwarten sind, wird nicht zwangsläufig die vollständige Anwendung ausgerollt. Speziell durch die Zusammenarbeit mit Praktikanten, war zu Beginn nicht klar, wie viele Features die Web Anwendung als Rahmenwerk zu Ende der Arbeit enthalten wird. Aus diesem Grund wurde im Dialog mit IC-Consult deutlich, welche der Anwendungen für die Praxis im Vordergrund stehen soll. In der Produktion wird bereits die Anwendung *Zed Attack Proxy* wird. Diese ist im Abschnitt 7.1 näher erläutert, führt jedoch zusammengefasst automatisiert Angriffe durch. Hierbei kommen hauptsächlich fertige Listen aus Angriffsvektoren zum Einsatz, es wird also eher Input Injection statt Input Fuzzing betrieben.
Zed Attack Proxy kann mit Nutzerscripten erweitert, welche verschiedene Zwecke erfüllen können. Bei den automatisierten Tests entstehen jedoch sehr viele Falsch-Positiv-Meldungen, die manuell überprüft werden müssen. Hierbei geht ein großer Teil der Arbeitszeit verloren, welche anderswo besser nutzbar wäre.
Ziel ist es nun, den Neural Evaluator in Zed Attack Proxy zu integrieren, sodass das neuronale Netz auswerten kann, ob der automatisierte Angriff erfolgreich war. So soll weniger manuelle Arbeit erforderlich sein.

7.1 Zed Attack Proxy

7.2 Integration und Ausrollung

7.3 Ergebnisse

7.4 Sonstige Arbeiten vor Ort

8 Anhang

A Abbildungsverzeichnis

B Tabellenverzeichnis

C Quelltextverzeichnis

D Literaturanalysen

Angabe der Literaturanalyse zu den wichtigsten Dokumenten, die in der Diplomarbeit referenziert wurden. Die analysierten Dokumente sind gemäß ihrer Relevanz zu ordnen. Die Literaturanalysen zu den drei ersten Dokumenten gehen verstärkt in die Bewertung der Diplomarbeit ein.

D.1 Learn&Fuzz: Machine Learning for Input Fuzzing

[PG17] Patrice Godefroid, Hila Peleg, Rishabh Singh: Learn&Fuzz: Machine Learning for Input Fuzzing, Microsoft Research, The Technion.

Inhalte Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Input Fuzzing mit rekurrenten neuronalen Netzen

(I2) Generierung von PDF-Objects

Defizite Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Bisherige Ansätze sind zeitlich aufwendig oder

(D2) nicht effektiv genug

(D2) Bislang keine Nutzung von neuronalen Ansätzen

Prämissen Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Es existiert keine Garantie für das Finden einer optimalen Lösung, da der Ansatz auf unsupervised Deep Learning basiert

(P2) Die Autoren beschränken sich auf die Generierung von PDF-Objects, welche im Browser Microsoft Edge getestet werden

Lösungen Was sind die eigenen Lösungen?

(L1) Mit einem rekurrenten neuronalen Netz werden PDF-Objects erstellt

(L2) Diese werden für Input Fuzzing genutzt

(L3) Eine hohe Überdeckung aller PDF-Befehle wird automatisch erreicht, dadurch muss keine aufwendige Grammatik erstellt werden

Nachweise Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Die Überdeckung über alle PDF-Befehle beträgt mehr als 95%

(N2) Es wurde ein bislang unentdeckte Stack-Overflow-Schwachstelle entdeckt

Offene Fragen Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Möglichkeit, diesen Ansatz auf gewöhnliche Nutzereingaben auszuweiten

(O2) Nutzung eines von RNNs verschiedenen Ansatzes

(O3) Automatisierte Auswertung

Sonstiges
Punkte, die in keine der oben genannten Kategorien passen

D.2 Titel der analysierten Publikation

[?] Vorname Name: Titel der analysierten Publikation, weitere Angaben.

Inhalte Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) ...

(I2) ...

(I3) ...

Defizite Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) ...

(D2) ...

(D3) ...

Prämissen Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) ...

(P2) ...

(P3) ...

Lösungen Was sind die eigenen Lösungen?

(L1) ...

(L2) ...

(L3) ...

Nachweise Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) ...

(N2) ...

(N3) ...

Offene Fragen Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) ...

(O2) ...

(O3) ...

Sonstiges
Punkte, die in keine der oben genannten Kategorien passen

E Literaturverzeichnis

[Bia] BISHOPFOX: *deephack*. https://github.com/BishopFox/deephack

[Bib] BISHOPFOX: *DEF CON 25 (2017) - Weaponizing Machine Learning - Petro, Morris - Stream - 30July2017*. https://www.youtube.com/watch?v=wbRx18VZlYA

[CD] CHRISTOPH DERNBACH, Jörg B.: *Sicherheitskreise: Hacker drangen in deutsches Regierungsnetz ein*. https://www.heise.de/newsticker/meldung/Sicherheitskreise-Hacker-drangen-in-deutsches-Regierungsnetz-ein-3983510.html

[Co] COOPERATION, The M.: *CVE - Common Vulnerabilities and Exposures*. https://cve.mitre.org/

[Ec14] ECKERT, Claudia: *IT-Sicherheit, Konzepte, Verfahren, Protokolle, 9. Auflage*. München, Bayern, Deutschland : Oldenbourg Wissenschaftsverlag GmbH, 2014. – ISBN 978–3–486–77848–9

[Gr] GRIGOREV, Alexey: *Overfitting*. http://mlwiki.org/index.php/Overfitting

[Hi] HINTON, Geoffrey: *Overview of mini--batch gradient descent*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[Ki] KIMMINICH, Björn: *OWASP Juice Shop Project*. https://www.owasp.org/index.php/OWASP_Juice_Shop_Project

[KK] KEVIN KILGOUR, Karlsruher Institut für T. Prof. Alex Waibel W. Prof. Alex Waibel: *Vorlesung Neuronale Netze*

[Li] LIMITED, DeepMind T.: *AlphaGo*. https://deepmind.com/research/alphago/

[Mc] MCLEOD, S. A.: *Pavlov's dogs*. https://www.simplypsychology.org/pavlov.html

[Ne] NEWS, NBC: *Sony's New Movies Leak Online Following Hack Attack*. https://www.nbcnews.com/tech/tech-news/sonys-new-movies-leak-online-following-hack-attack-n258511

[Of] OFFNFOPT, Wikipedia-Nutzer: *Künstliches neuronales Netz*. https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz

[OW] OWASP: *Fuzzing*. https://www.owasp.org/index.php/Fuzzing

[Pa] PASZKE, Adam: *Reinforcement Learning (DQN) tutorial*. http://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#sphx-glr-intermediate-reinforcement-q-learning-py

[PDHH] PROF. DR. HANNES HARTENSTEIN, Karlsruher Institur für T. Forschungsgruppe DSN D. Forschungsgruppe DSN: *Vorlesung IT-Sicherheitsmanagement für vernetzte Systeme*

[PG17] PATRICE GODEFROID, Rishabh S. Hila Peleg P. Hila Peleg: *Learn&Fuzz: Machine Learning for Input Fuzzing*. Microsoft Research, 25.01.2017

[Pr] PROJECT, Open Web Application S.: *OWASP WebGoat Project*. `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`

[PSA] PROF. SEBASTIAN ABECK, Karlsruher Institut für T.: *Web-Anwendungen Und serviceorientierte Architekturen (WASA)*

[te] TEAM, DVWA: *Damn Vulnerable Web Application*. `https://github.com/ethicalhack3r/DVWA`

[TLH] THANH-LE HA, Karlsruher Institut für T. Prof. Alex Waibel W. Prof. Alex Waibel: *Praktikum Neuronale Netze*

[Vea] VERSCHIEDENE: *Is PyTorch faster than MXNet or TensorFlow?* `https://www.quora.com/Is-PyTorch-faster-than-MXNet-or-TensorFlow`

[Veb] VERSCHIEDENE: *TensorFlow 60-80% slower than PyTorch on training Wide ResNet*. `https://github.com/tensorflow/tensorflow/issues/9322`