Hochschule Esslingen
University of Applied Sciences

Bachelor-Abschlussarbeit

Maschinelle Sicherheitsüberprüfung neuronaler Netze für biometrische Authentifizierung mithilfe von Fuzzing

im Studiengang Softwaretechnik
der Fakultät Informationstechnik
Sommersemester 2021

Nico Dietz

Zeitraum: 24. April 2021 bis 24. August 2021
Prüfer: Prof. Dr. Tobias Heer
Zweitprüfer: Prof. Dr. Markus Enzweiler

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 23. August 2021 _____

Unterschrift

Kurz-Zusammenfassung

Diese Arbeit behandelt die Thematik von False Positives bei neuronalen Netzen und wie diese gezielt erzeugt werden können. Als möglicher Ansatzpunkt wird insbesondere das Programm TensorFuzz in Betracht gezogen, welches jedoch sehr zufallsbasiert arbeitet und nur durch hohen Aufwand Verwendung finden würde.

Für die Eingabedaten des Fuzzing Prozesses kommt eine Generative Adversarial Network zum Einsatz. Die Eigenschaften dieses Netzes erlauben eine Interpolation zwischen zwei Zahlen, wobei jeder Zwischenschritt auch wieder eine Zahl ist. Durch ein weiteres neuronales Netz, welches auf die Erkennung von Zahlen trainiert ist, werden während der Interpolation mögliche Kandidaten für False Positives gefunden. Allerdings wurde keine Methode gefunden, um Kandidaten maschinell zu überprüfen, wodurch jeder Kandidat manuell von einem Benutzer überprüft werden muss. Mit einer Trefferquote von circa 70% ist die Methodik effektiv für die Findung von False Positives. Mit den gefundenen Kandidaten und dessen Auswertung können Defizite des neuronalen Netzes ausfindig gemacht werden.

Außerdem werden verschiedene Methoden für eine Coverage-Messung umgesetzt und untereinander verglichen. Aufgrund der fehlenden maschinellen Überprüfung der Kandidaten, hat die gemessene Coverage eine geringe Aussagekraft, bietet aber eine gute Grundlage, falls eine solche Überprüfung verfügbar ist.

Mit dieser Arbeit wird ein Grundstein für weitere Forschung und Entwicklung von richtigem Coverage Guided Fuzzing bei neuronalen Netzen geschaffen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Heutzutage sind Biometrische Authentifizierungsmethoden allgegenwärtig und selbst in mobilen Endgeräten etabliert. Menschen können Bilder von Personen mit relativer Leichtigkeit erkennen, auch wenn diese verrauscht sind. Bildrauschen beschreibt die Verschlechterung eines digitalen Bildes. Das Problem bei neuronalen Netzen ist, dass es ein Bild allerdings nicht visuell wahrnimmt, sondern als eine Matrix aus Zahlen. Durch das Addieren von Rauschen ändert sich die komplette Matrix und somit ist das Bild nun aus Sicht des Netzes ein völlig anderes, obwohl es sich optisch nahezu nicht ändert. Dadurch kann es vorkommen, dass das neuronale Netz plötzlich eine andere Person erkennt und somit jemand Zugriff auf Daten gibt, auf welche diese Person eigentlich keine Zugriffsrechte hat. So ein Fall wird auch False Positive genannt.

Um dieses Problem zu untersuchen kann ein Fuzzer verwendet werden. In der einfachsten Form generiert ein Fuzzer zufällige Eingaben und provoziert so bei Programmen einen Fehler, falls die generierte Eingabe nicht verarbeitet werden kann. Um den Faktor des Zufalls ausschließen zu können, kann die Methode des Coverage-Guided Fuzzing verwendet werden. Dabei erhält der Fuzzer die Möglichkeit nicht nur Eingaben zu generieren, sondern auch die jeweilige Ausgabe des Programms zu analysieren und basierend darauf die nächste Eingabe anzupassen. Damit kann die nächste Eingabe in die gewünschte Richtung gelenkt werden, um dem Ziel näher zu kommen.

1.1 Wissenschaftliche Fragestellung

Mit dieser Arbeit soll durch Fuzzing ein False Positive provoziert werden. Zusätzlich wird untersucht, ob die Trainingsdauer des Modells oder andere Parameter einen Einfluss auf diesen Prozess haben. Die Thematik ist unter dem Namen Adversarial Machine Learning bekannt [1]. Falls möglich soll ein Bild oder ein Filter gefunden werden, welcher als Master Key funktioniert und quasi immer einen False Positive hervorruft. Realistischer ist ein Algorithmus, welcher effizient und schnell False Positives generieren oder finden kann. Hierfür soll die Methodik des Coverage Guided Fuzzing genutzt werden. Demnach sind die Messung der Coverage und das Schließen der Schleife zwischen Ausgabe und nächster Eingabe wichtige Bestandteile der Aufgabe.

Daraus ergeben sich folgende Fragen:

WF1 Wie wird möglichst effizient ein großes Spektrum an False Positives generiert, um neuronale Netze auszutricksen?

WF1.1 Wie werden False Positives generiert und erkannt?

WF1.2 Eine hohe Coverage wird für ein großes Spektrum benötigt, aber wie wird Coverage bei neuronalen Netzen gemessen?

1.2 Ausgangssituation

Die Arbeit „On the Resilience of Biometric Authentication Systems against Random Inputs“ [2] ist der Ursprung dieser Arbeit. Hier wird der Einfluss der Trainingsart des jeweiligen Modells auf zufällige Eingaben analysiert. Im Umfang der Arbeit „TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing“ [3] ist das Programm TensorFuzz entstanden, mit dem, mittels Fuzzing, Fehler in neuronalen Netzen gefunden werden können. TensorFuzz bietet durch funktionale Beispiele eine mögliche Grundlage und zeigt damit Schnittstellen zwischen neuronalen Netzen und Fuzzing auf. Eine weitere Möglichkeit für die Lösung sind Generative Adversarial Networks. Durch das dabei entstehende Generator-Netz können Bilder erzeugt werden, welche als Eingabe verwendet werden können. Auf diesen Weg muss ein Bild nicht immer weiter bearbeitet werden, sondern es kann immer direkt ein neues Bild generiert werden.

2 Grundlagen

In diesem Kapitel werden die verwendeten Technologien, Methoden und Programme erklärt. Dazu zählen unter anderem neuronale Netze in verschiedenen Formen, Fuzzing und die Gewinnung von Eingabedaten. Neuronale Netze werden zunächst in der Grundidee erklärt und dann wird spezifisch auf die verwendeten Varianten eingegangen. Bei Fuzzing werden der Grundansatz und das Coverage Guided Fuzzing erläutert. Für die Gewinnung von Eingabedaten werden zwei Methoden aufgezeigt, welche für die Verwendung in Frage kommen.

2.1 Künstliche neuronale Netze

Künstliche neuronale Netze nutzen, wie in Abb. 2.1 zu sehen ist, verschiedene mathematische Ebenen, um Informationen zu verarbeiten. Innerhalb dieser Ebenen sind von ein paar wenigen bis hin zu tausenden Neuronen, welche auch Units genannt werden. Units führen mathematische Berechnungen auf der Grundlage ihrer Vorgänger durch und reichen sie an andere Units weiter, welche die Daten weiterverarbeiten. So ergibt sich aus einer Vielzahl untereinander vernetzter Units, die jeweils nur kleine Operationen ausführen, eine mögliche Lösung für ein komplexes Problem. Verschiedenste Informationen aus der realen Welt werden auf der Input-Ebene widergespiegelt. Das neuronale Netz verarbeitet diese Daten oder lernt aus ihnen. Hierfür laufen die Daten durch eine oder mehrere Hidden-Ebenen, wodurch die Eingabedaten so transformiert werden, dass die Output-Ebene diese interpretieren kann.

Die meisten neuronalen Netze haben alle Ebenen vollständig untereinander verbunden. Alle Verbindungen sind jedoch unterschiedlich gewichtet. Je stärker die Verbindung gewichtet ist, desto größer ist der Einfluss auf das Ergebnis. Während die Daten das Netz durchlaufen, lernt dieses immer mehr über die enthaltenen Informationen. Am Ende wird das Ergebnis über die Output-Ebene ausgegeben.

Wissenschaftler haben in der Vergangenheit sehr viel über das menschliche Gehirn gelernt. Ein sehr wichtiger Aspekt ist die Erkenntnis, dass unterschiedliche Teile des Gehirns, für unterschiedliche Aufgaben zuständig sind. Die Funktionsweise neuronaler Netze ist dem nachempfunden. Die Eingabe durchläuft die Neuronen und wird von Ebene zu Ebene immer dem Teilgebiet zugeordnet, zu welchem die Daten gehören. So wird immer weiter spezifiziert um welche Daten es sich handelt.

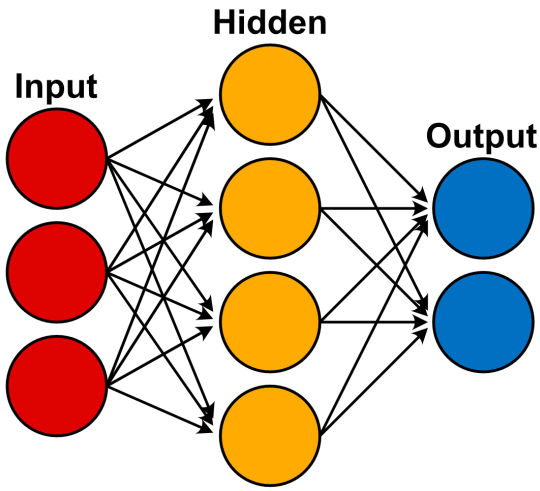

Abb. 2.1: Ebenen-Aufbau eines künstlichen neuronalen Netzes

Damit ein neuronales Netz trainiert werden kann, benötigt es sehr viele Daten, die als Datensatz zusammengefasst werden. Hierbei gibt es zu jedem Bild ein Label, in dem wichtige Daten, wie die Klassifizierung des Bildes, stehen. Um zum Beispiel einen Hund von einer Katze zu unterscheiden, werden tausende Bilder benötigt, auf denen entweder eine Katze oder ein Hund markiert sind. Bei einem Hundebild steht somit im Label, dass auf dem Bild ein Hund zu sehen ist. Sobald das Netz trainiert wurde, versucht es mit den gewichteten Verbindungen alle zukünftigen Eingaben richtig zu klassifizieren. Das Label wird in diesem Fall zur Kontrolle verwendet. Werden Bilder mit einer hohen Wahrscheinlichkeit richtig klassifiziert, ist das Training erfolgreich beendet. Falls es nicht richtig liegt, werden die Verbindungen angepasst und ggf. anders gewichtet. Dieser Vorgang wird als Deep Learning bezeichnet [4].

2.1.1 Convolutional Neural Network

Im Vergleich zu herkömmlichen künstlichen neuronalen Netzen sind bei Convolutional Neural Networks (CNN) nicht alle Ebenen untereinander vollständig verbunden. Es gibt vier verschiedene Arten von Ebenen. Das sind die Convolutional-, die Nonlinearity-, die Pooling- und die Fully-Connected-Ebenen. Während erstere und letztere Parameter haben, kommen die beiden anderen ohne Parameter aus. CNN's kommen besonders häufig bei Bildklassifizierung zum Einsatz, da sie bei dieser Art Problem eine hervorragende Performance aufweisen. Der Grundgedanke von künstlichen neuronalen Netzen wird stets beibehalten, auch wenn durch neue Arten von Ebenen, die Umsetzung eines Problems anders sein kann. So können beispielsweise durch diese Ebenen die wichtigsten Teile von Bildern ohne Berechnung herausgefiltert werden [5].

2.1.2 Generative Adversarial Network

Generative Adversarial Networks bestehen aus zwei konkurrierenden Netzen. Während das eine Generator-Netz versucht realistisch aussehende Bilder zu generieren, versucht das Diskriminator-Netz die generierten Bilder zu entlarven. Wie in Abb. 2.2 zu sehen ist, wird der ganze Prozess so lange durchgeführt, bis das Diskriminator-Netz die generierten Bilder nicht mehr von echten unterscheiden kann.

Hierfür wird zunächst das Diskriminator-Netz mit realen Bildern trainiert. Nun generiert das Generator-Netz immer wieder Bilder und gibt diese zur Klassifizierung an das Diskriminator-Netz, welches dann als Feedback zurückgibt, ob es ein echtes oder ein generiertes Bild ist. Außerdem gibt es zurück, zu welcher Wahrscheinlichkeit das klassifizierte Bild ein generiertes beziehungsweise echtes Bild ist. So kann das Generator-Netz sich immer weiter vor arbeiten, bis es den Punkt erreicht, an dem die generierten Bilder als echte erkannt werden [6].

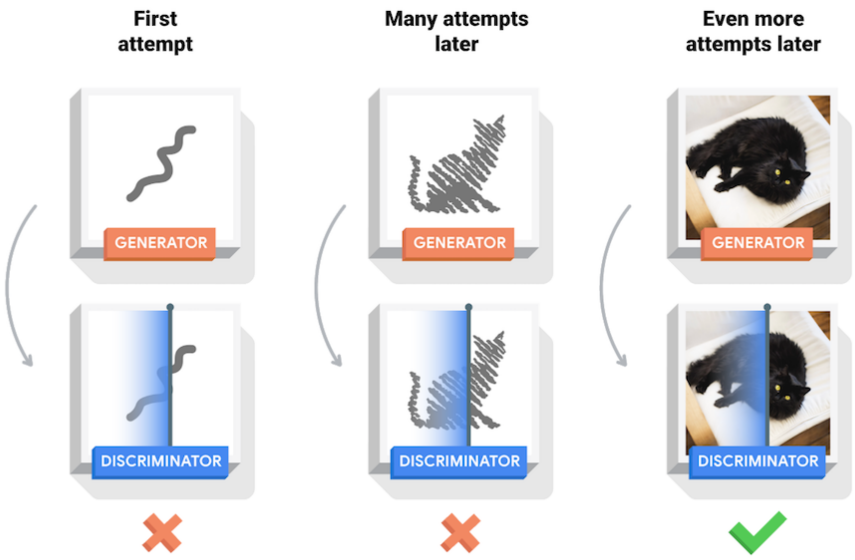

Abb. 2.2: Generative Adversarial Network Prinzip [7]

Eine sehr wichtige Eigenschaft eines trainierten Generator-Netzes ist, dass alle Bilder aus einem riesigen, aber endlichen Raum generiert werden. Dieser Raum ist ein hochdimensionaler Raum aller Bilder die von Menschen als echt wahrgenommen werden. Dieser Raum ist stetig, wodurch das Generator-Netz numerische Funktionen wie die Interpolation ermöglicht [6].

2.2 Fuzzing

Fuzzing oder auch Black-Box-Fuzzing ist eine automatisierte Technik für Softwaretests. Ursprünglich als Universitäts-Projekt entwickelt, wird die Methodik bis heute genutzt. Mit Fuzzing sollen in einem Programm Fehler und gegebenenfalls Sicherheitslücken gefunden werden [8].

Das Ganze funktioniert so, dass zufällige Eingabe an eine Programmschnittstelle gegeben werden. Wenn kein Fehler entsteht, wird eine Coverage erhoben. Die Coverage beschreibt die Menge an Pfaden im Programmablauf, welche bereits durchschritten wurden. Sobald alle Pfade abgelaufen wurden, beträgt die Coverage 100%. Wenn ein Fehler oder Absturz auftritt, wird der Test gespeichert, um ihn zu einem späteren Zeitpunkt wiederholen zu können. [9]

2.2.1 Coverage Guided Fuzzing

Coverage Guided Fuzzing verbessert den ursprünglichen Fuzzing-Prozess und passt basierend auf der Coverage-Analyse die nächst Eingabe an. Wie in Abb. 2.3 zu sehen ist, wird eine Eingabe in das Programm gegeben. Dann wird die Coverage erhoben oder ein Absturz gemeldet. Wenn eine Coverage erhoben wird, wird basierend darauf die nächste Eingabe so verändert, dass der Prozess beschleunigt wird, indem die Eingabe nun gezielt neue Pfade durchschreitet [10].

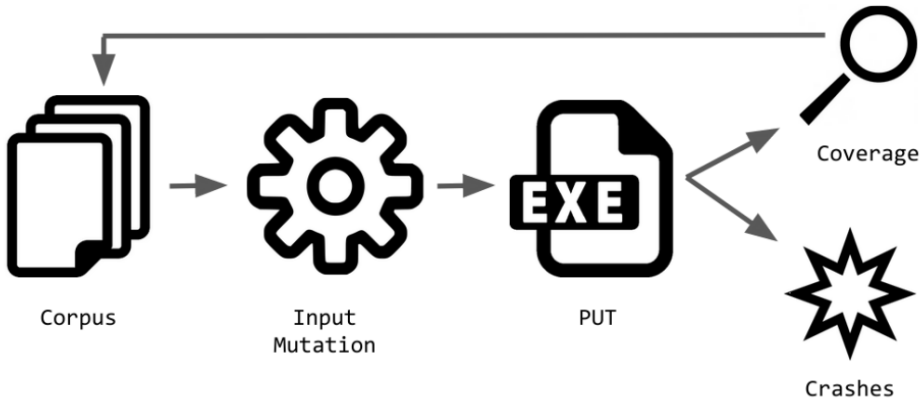

Abb. 2.3: Coverage Guided Fuzzing Ablauf [10]

2.3 Eingabedaten

Für den Fuzzing-Prozess werden Eingabedaten benötigt. Diese können unter anderem durch Mutation oder Generierung erzeugt werden. Im Umfang dieser Arbeit sind die Eingabedaten in der Form von Bildern, wodurch nur zwei Methoden zur Gewinnung übrig bleiben. Bei der Mutation werden vorhandene Bilder immer wieder verändert, während bei der Generierung keinerlei Bilder im voraus benötigt werden, da immer wieder neue Bilder generiert werden.

2.3.1 Mutation mit White Noise

Mutation von Bildern wird mit einem so genannten White Noise Adding durchgeführt. In Abb. 2.4 ist ein solcher White Noise zu sehen. White Noise ist ein Bild aus normalverteilten Werten. Dieser wird auf ein vorhandenes Bild addiert, wodurch das Bild aus digitaler Sicht ein völlig anderes ist, auch wenn es sich optisch unter Umständen kaum verändert hat [1].

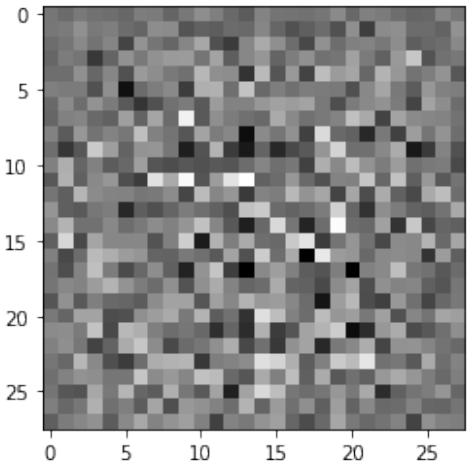

Abb. 2.4: White Noise

Mutation mit White Noise ist eine einfache Methodik für die Mutation von Bildern, da White Noise sehr einfach generiert werden kann und die Bilder bereits vorhanden sind. Das Problem hierbei ist allerdings, dass wenn zu oft White Noise auf ein Bild addiert wird, dass es irgendwann so verrauscht ist, dass von dem ursprünglichen Bild, selbst optisch, nichts mehr zu erkennen ist. In solchen Fällen sind False Positives nahezu garantiert, was allerdings keine Aussage mehr hat, da es selbst für einen Menschen nicht mehr klassifizierbar ist.

2.3.2 Generierung durch Interpolation

Durch Interpolation können mit Hilfe eines trainierten Generator-Netzes Bilder generiert werden. Bei der Interpolation werden zwei Punkte aus dem Raum aller möglichen Bilder des Generator-Netzes gewählt. Besonders effektiv ist die Methode, wenn die zwei ausgewählten Punkte nicht der gesuchten Klassifizierung entsprechen. Wie in Abb. 2.5 zu sehen ist, sind die Schritte der Interpolation durch die Stetigkeit des Generator-Netzes immer leichte Veränderungen in Richtung des Zielbildes. Wird nun beispielsweise von einer 3 zu einer 9 interpoliert, ist auf dem Weg keine 4 zu erwarten. Während zwischen zwei Bildern interpoliert wird, wird für jedes Bild eine Klassifizierung vorgenommen und falls eine Klasse erkannt wird, welche weder dem Start- noch dem Zielpunkt entspricht, gilt es als False Positive Kandidat. Es werden also nicht direkt False Positives erkannt, sondern lediglich Kandidaten gefunden, welche von einem Benutzer überprüft werden müssen. Die Kandidaten müssen weiterhin manuell verifiziert werden, aber sind in vielen Fällen False Positives.

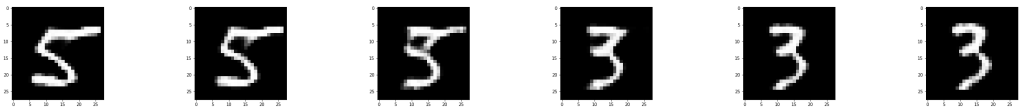

Abb. 2.5: Interpolation von der Zahl fünf zu der Zahl drei

2.4 Technologien

Im Umfang dieser Arbeit werden TensorFlow und Jupyter Notebooks verwendet. Zudem wird das Programm TensorFuzz angeschaut, ob es als mögliche Grundlage dienen kann. Auf alle drei Technologien wird im folgenden näher eingegangen.

2.4.1 TensorFlow

TensorFlow ist ein, von Google Brain Team, entwickeltes Open-Source Framework für maschinelles Lernen. Es bietet Bibliotheken, Tools und viele hilfreiche Ressourcen für das Entwickeln von künstlichen neuronalen Netzen. Während es zunächst als internes Projekt von Google gedacht war, wurde es 2015 veröffentlicht und ist seit dem durch eine große Community stetig gewachsen. Die aktuellste Version ist 2.6.0 und kann in den Sprachen C++ und Python verwendet werden. Ebenso gibt es viele öffentliche Tutorials, welche funktionierenden Code mit Erklärungen beinhalten. So wird ein schneller Start in die Thematik ermöglicht [11].

2.4.2 TensorFuzz

TensorFuzz ist das Ergebnis der Arbeit „TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing“. Hier wird der Sachverhalt von Coverage Guided Fuzzing bei neuronalen Netzen untersucht. Das dabei entstandene Programm ist ein Framework, für die Findung von verschiedenen mathematischen Problem bei neuronalen Netzen. Während mehrere Heuristiken für die Steigerung der Effizienz versucht werden, kommt als Ergebnis raus, dass das Programm am schnellsten mit zufälligen Arbeiten funktioniert.

Der Programmablauf beginnt mit den Eingabedaten. Die Eingabedaten, inklusive Labels, sind der Seed Corpus, welcher in Abb. 2.6 unten rechts zu sehen ist. Seed Corpus beschreibt valide Eingaben und ist der Beginn für den Fuzzing Prozess. Diese Daten werden an den Fuzzer übergeben und bilden zunächst den Input Corpus, woraus dann ein Bild vom Input Chooser ausgewählt wird.

Der Mutator verändert das Bild so, dass das neuronale Netz im besten Fall getäuscht wird. Die Veränderung ist ein sogenannter White Noise, welcher auf die Matrix des Bildes addiert wird [12] und von Menschen als eine Art Bildrauschen wahrgenommen wird.

Das veränderte Bild wird dem neuronalen Netz übermittelt, welches dann dem Bild eine Klasse zuordnet. Das Ergebnis erfüllt entweder die Objective Function und beendet somit den Fuzzer oder es muss analysiert werden und der Vorgang beginnt von vorne. Die Objective Function wird aktiviert, sobald ein Bild einer falschen Klasse zugeordnet wird. Dies wäre beispielsweise damit vergleichbar, dass einer falschen Person Zugriffsrechte gewährt werden, da sie als eine andere Person erkannt wird. Ist die Objective Function nicht erfüllt, wird das Ergebnis analysiert und für die nächste Eingabe als Information verwendet.

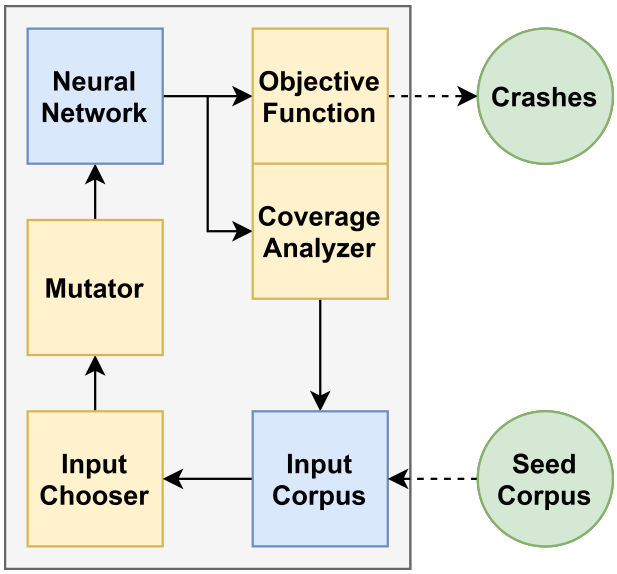

Abb. 2.6: Programmablaufplan von TensorFuzz [3]

2.4.3 Jupyter Notebook

Jupyter Notebook verbindet ausführbaren Quelltext mit Dokumentation. Es ist eine webbasierte Entwicklungsumgebung in der sowohl programmiert, als auch parallel ordentlich dokumentiert werden kann. Es gibt drei Arten von Zellen, mit denen ein Notebook zusammengesetzt werden kann. Es gibt Quelltext-, Text- und Plot-Zellen. Während in den Quelltext-Zellen mit Python programmiert werden kann, wird in Text-Zellen dokumentiert und in Plot-Zellen werden Plots angezeigt [13].

In Abb. 2.7 ist ein Ausschnitt aus einem Jupyter Notebook zu sehen. Ganz oben ist eine Text-Zelle mit Informationen über den folgenden Code. Direkt darunter ist der Quelltext, welcher sofort über den Pfeil an der linken Seite ausgeführt werden kann. Darunter ist dann die Ausgabe des Programms zu sehen. Durch Jupyter Notebook ist die Entwicklung sehr flexibel. Es können einzelne Programmteile überschrieben werden, in dem eine Zelle erneut ausgeführt wird. Außerdem ist es sehr gut geeignet, um mit mehreren Personen zusammen an einem Projekt zu arbeiten, da die Dokumentation sehr einfach zu erstellen ist und direkt in der Entwicklungsumgebung wiederzufinden ist.

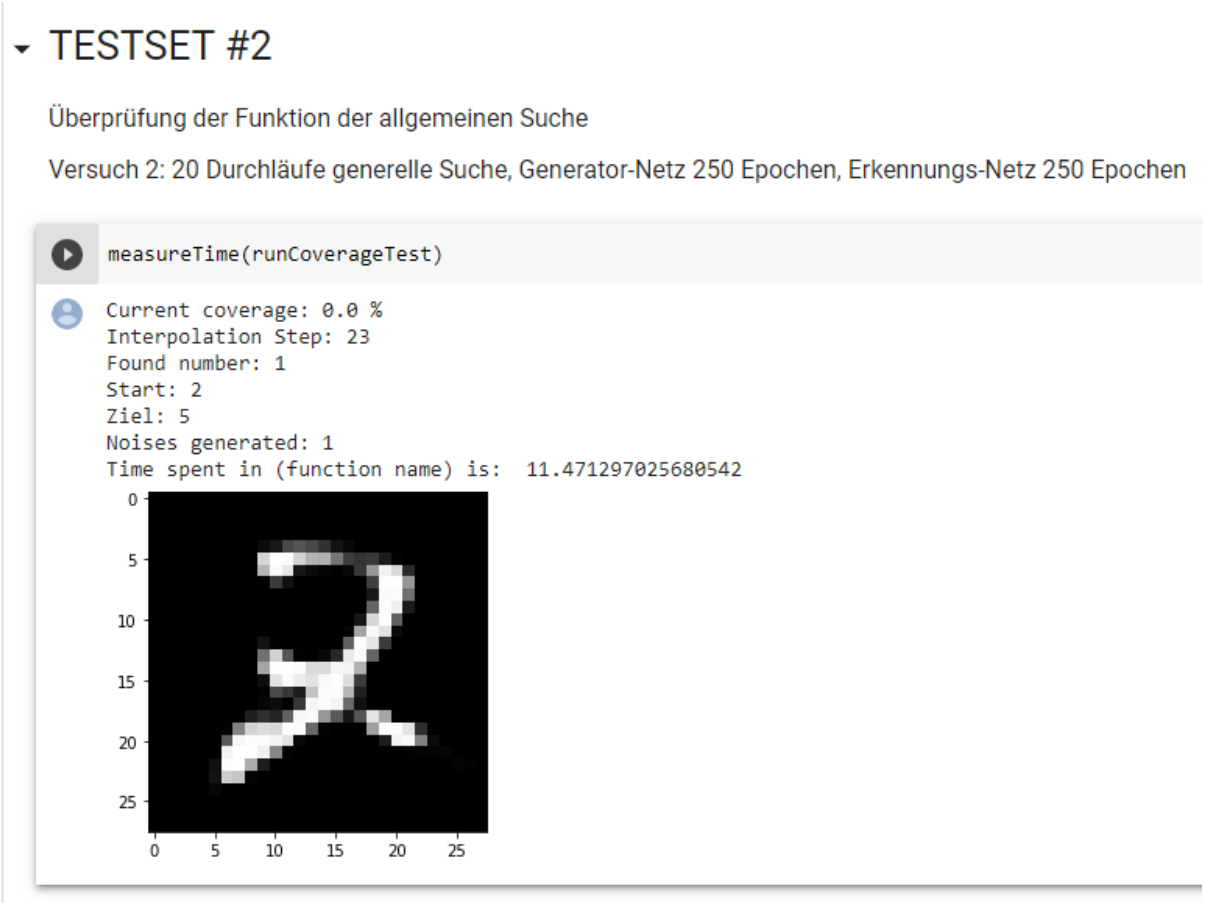

Abb. 2.7: Bildausschnitt Jupyter Notebook

3 Konzeption

Basierend auf den Grundlagen und der Analyse von möglichen Ansätzen, wird eine Konzeption ausgearbeitet, welche als Leitfaden für die Realisierung zu verstehen ist. Die Umsetzung erfolgt in Python und für das neuronale Netz wird die Plattform TensorFlow verwendet. Für das Fuzzing wird das Framework TensorFuzz in Betracht gezogen. TensorFlow ist eine Open-Source Lösung, welche aus Tools, Bibliotheken und Ressourcen besteht und das Arbeiten mit neuronalen Netzen erleichtert. TensorFuzz wurde entwickelt, um Fuzzer und insbesondere Coverage Guided Fuzzing mit neuronalen Netzen zu verwenden [3]. Die Alternative zu TensorFuzz als Grundlage ist es einen eigenen Algorithmus zu schreiben.

Um die Komplexität zu reduzieren, wird mit dem MNIST Datensatz gearbeitet. Wie in Abb. 3.1 zu sehen ist, besteht dieser aus handgeschriebenen Zahlen. Datensätze mit Gesichtern oder Fingerabdrücken sind von der Datenmenge deutlich größer, da die Bilder hochauflösend sind. Dies erhöht die benötigte Rechenleistung enorm und verlangsamt dadurch den gesamten Prozess. Ebenso ist das Training aufwendiger und gute Datensätze in Richtung biometrischer Erkennung sind selten.

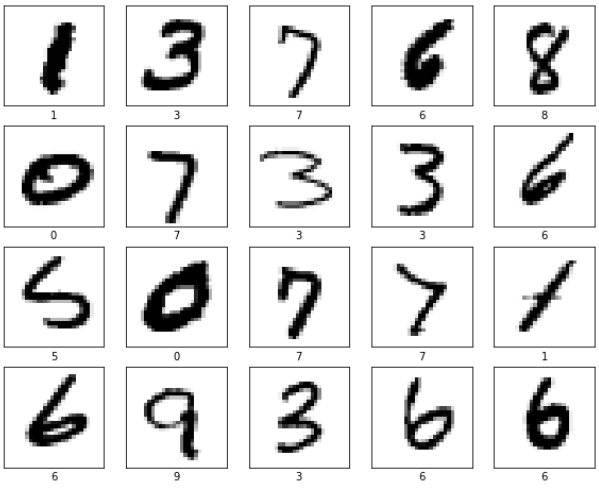

Abb. 3.1: Beispiel Datensatz MNIST [14]

3.1 Coverage Guided Fuzzing bei neuronalen Netzen

Coverage Guided Fuzzing im Zusammenhang mit neuronalen Netzen funktioniert anders als bei herkömmlichen Programmen. Normalerweise wird die Coverage anhand der durchlaufenen Programmpfade gemessen, was bei einem neuronalen Netz nicht möglich ist. Es gibt allerdings die Möglichkeit eine Art Coverage über Neuronen-Aktivität zu messen.

Folgende drei Methoden werden umgesetzt:

1. False Positive Kandidat für jede Klasse finden
2. die aktivsten Neuronen pro Ebene überprüfen (die X höchsten Werte)
3. aktive Neuronen pro Ebene überprüfen (jeder Wert über Schwellenwert X)

Methode 1 hebt sich stark von den anderen beiden ab. In diesem Fall wird nicht direkt die Neuronen-Aktivität analysiert, sondern es wird lediglich für jede Klasse ein False Positive Kandidat gesucht. Es wird davon ausgegangen, dass solange bis für jede Klasse ein Kandidat gefunden wird, genug Neuronen aktiv geworden sind. Außerdem wird verglichen, wie lange es pro Klasse dauert, bis ein Kandidat gefunden wird. So soll erkannt werden, bei welchen Klassen das neuronale Netz gegebenenfalls Schwächen aufzeigt.

Bei Methode 2 werden alle Neuronen in einer Ebene des Netzes überprüft. Es werden aber nur die Neuronen mit der höchsten Aktivität gezählt, hierfür muss allerdings durch ein Experiment entschieden werden, wie viele der höchsten Werte gezählt werden. Je nach Ergebnis könnten es beispielsweise die drei, fünf oder auch zehn höchsten Werte sein.

Methode 3 ist ähnlich der 2. Methode und unterscheidet sich nur darin, dass alle Neuronen die eine bestimmte Aktivität aufweisen, also einen Wert größer dem gewählten Schwellenwert haben, als aktiv gespeichert werden. Sobald alle Neuronen aktiv waren, ist eine Coverage von 100% erreicht. Durch einen Versuch wird ermittelt, welcher Schwellenwert gut geeignet ist.

Für Coverage Guided Fuzzing müssen immer wieder neue Eingaben in das neuronale Netz gegeben werden. Die Eingaben können auf folgende Wege erhalten werden:

1. Bilderpool und Veränderung durch Mutation
2. neue Bilder mit Hilfe eines Generative Adversarial Network generieren

Bei Variante 1 müssen einerseits Bilder für den anfänglichen Pool gesucht werden, als auch ein Algorithmus geschrieben werden, welcher mit Bildrauschen das Bild verändert. Die größte Limitierung ist hierbei, dass der Pool stark begrenzt ist und die Bilder immer nur verrauschter werden. Durch das Verrauschen der Bilder wird es irgendwann garantiert anders als urprünglich klassifiziert. Das Problem hier ist nämlich, dass das neuronale Netz die Klassen null bis neun kennt und die Eingabe immer einer Klasse zuordnen muss.

Variante 2 ist komplexer, aber flexibler, denn hier kommt ein Generative Adversarial Network (GAN) zum Einsatz. Bestehend aus einem Generator- und einem Diskriminator-Netz

muss das GAN erst unabhängig trainiert werden, bevor anschließend das Generator-Netz zum Einsatz kommt. Sobald Das Generator-Netz vollständig trainiert ist, ist es in der Lage aus einem erzeugbaren Rauschbild ein Bild mit einer Zahl zu generieren. Das Rauschbild besteht in diesem Fall aus einem 1D-Array mit 100 Normalverteilten Werten. Hier besteht jetzt die Möglichkeit, jederzeit ein neues Bild zu erzeugen oder das vorherige zu verändern. Falls nun Muster bei der Ausgabe erkannt werden, dass zum Beispiel die erste Bildhälfte bei hohen Wert ganz besonders oft ein False Positive provoziert, so könnten die Werte in der ersten Hälfte des Rauschbilds angehoben werden. Eine weitere Option mit einem GAN ist, dass zwei Rauschbilder gewählt werden zwischen denen interpoliert wird.

Aufgrund der Komplexität wird nur Variante 2 umgesetzt, da hier die Flexibilität überwiegt. Um nicht willkürlich Bilder zu generieren wird hierbei auf die Methodik der Interpolation gesetzt. Die Idee hierbei ist, dass wenn beispielsweise von Zahl sieben zu Zahl neun interpoliert wird, dass dazwischen keine anderen Zahlen auftreten sollten, sondern irgendwann der Punkt kommt, wo das Erkennungs-Netz nicht mehr die erste, sondern die zweite Zahl erkennt. Wird bei der Interpolation allerdings eine andere Zahl klassifiziert, könnte es sich um einen False Positive handeln und das Bild wird als Kandidat abgespeichert. Sobald dieser Punkt erreicht ist und die Idee hinter der Methode erfolgreich ist, können die Zusammenhänge zwischen False Positives analysiert werden.

Die Analyse kann dann genutzt werden, um die nächste Eingabe anzupassen. Hierfür werden die Ergebnisse einer Ebene des neuronalen Netzes überprüft, um Muster zu erkennen. Als Beispiel ist die Eingabe eines Bildes mit einer 3, welches allerdings als 6 klassifiziert wird. Im Vergleich für die Analyse dienen nun Bilder mit einer 3, die auch als eine 3 klassifiziert werden, als auch Bilder mit einer 6, die auch als 6 klassifiziert werden. Sollten mehrere Bilder mit einer 3, die als 6 klassifiziert werden, entstehen, werden diese auch untereinander verglichen. Sobald Muster und Abhängigkeiten von Eingabe und Ausgabe bestehen, kann ein Algorithmus geschrieben werden, um die Eingabe entsprechend anzupassen.

3.2 TensorFuzz als Grundlage

Aufgrund der Gegebenheiten von TensorFuzz wirkt es zunächst als geeignete Grundlage. TensorFuzz ist insbesondere gut geeignet, um mathematische Probleme bei neuronalen Netzen zu finden. Jedoch arbeitet TensorFuzz nicht nach Coverage Guided Fuzzing wie im eigentlichen Sinne. Verschiedene Heuristiken sollen mittels Coverage-Messung und Analyse den Prozess beschleunigen. Allerdings sind die Heuristiken langsamer als Zufallseingaben, weswegen es sich mehr um reines Fuzzing handelt.

Die vordefinierten Schnittstellen sind auf die mathematische Fehlerfindung ausgelegt, was zur Folge hat, dass auf vorhandene Funktionen zurückgegriffen werden muss, um eine komplette Neuentwicklung des Programms zu vermeiden. Die Anpassung von TensorFuzz, für das gegebene Problem, ist aus mehreren Gründen sehr aufwändig:

- Schnittstellen sind tief im Programm integriert

- TensorFuzz basiert auf TensorFlow 1.6.0 und das Update auf Version >2.0 ist nicht ohne Weiteres möglich
- TensorFlow 1.6.0 erlaubt nicht die neueste Python Version und zwingt auch einige Abhängigkeiten zu alten Versionen
- Dokumentationen über TensorFlow sind primär nur noch ab Version 2.0 verfügbar

Nicht nur Schnittstellen müssen dadurch angepasst werden, auch der Mutator müsste durch ein Generative Adversarial Network ausgetauscht werden. Wie in Abb. 3.2 zu sehen ist, müsste nach dem Input Chooser das Generator-Netz sein, um Bilder erzeugen zu können. Dadurch würden allerdings der Seed Corpus, der Input Corpus und der Input Chooser überflüssig werden. So würde alleine für Anpassungen das komplette Programm umgeschrieben werden, ohne neue Funktionen hinzugefügt zu haben.

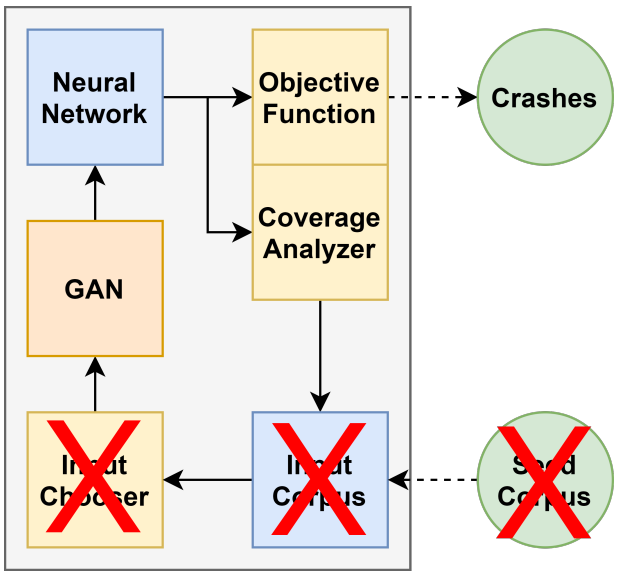

Abb. 3.2: Veränderter Programmablaufplan von TensorFuzz

Somit muss von TensorFuzz als Grundlage abgesehen werden. TensorFuzz auf den speziellen Anwendungsfall umzuschreiben, wäre gleichzusetzen mit dem Aufwand ein Programm von Grund auf neu zu schreiben. Daher wird das Programm von Grund auf neu entwickelt und entsprechend den Anforderungen implementiert.

4 Realisierung

Das Projekt wird mit Jupyter Notebooks umgesetzt. Da neuronale Netze insbesondere während des Trainings enorme Rechenleistung benötigen, wird das Notebook auf einem Server mit Nvidia Tesla GPUs ausgeführt. Hierdurch kommt es allerdings zu dem Problem, dass Messergebnisse unterschiedlich ausfallen können, da die Auslastung des Servers variiert. Außerdem ist eine Session auf vier Stunden limitiert.

Da die Session auf dem Server zeitlich begrenzt ist und danach alle Dateien gelöscht werden, müssen Hilfsfunktionen implementiert werden, welche den schnellen Download von Dateien ermöglichen. Im List. 4.1 sind die wichtigsten zu sehen. Mit einem *!* werden Shell-Befehle ausgeführt, welche eine Dateiverwaltung ermöglichen. Mit *!zip* können alle Dateien im Arbeitsverzeichnis in einen ZIP-Ordner gepackt werden, welcher dann mit *files.download()* heruntergeladen werden kann. Dies ist sehr hilfreich, wenn Modelle gespeichert werden sollen, oder Dateien analysiert werden müssen, da die Oberfläche von Jupyter nicht für Dateiverwaltung optimiert ist.

Besonders für Messungen sind auch die *!rm* Befehle interessant, da sie Dateien, welche nicht mehr benötigt werden oder bereits heruntergeladen wurden, löschen können.

```
1 !zip -r /content/file.zip /content/*
2
3 files.download("/content/file.zip")
4
5 !rm /content/*.png
6 !rm /content/*.txt
7 !rm /content/*.zip
```

List. 4.1: Hilfsfunktionen Download / Dateiverwaltung

4.1 Generator-Netz

Im ersten Schritt wird das Generative Adversarial Network benötigt, da ohne dieses keine Eingabedaten existieren. TensorFlow bietet die Grundlage für dieses Projekt als Tutorial an. Das von TensorFlow bereitgestellte Jupyter Notebook zu dem Thema Deep Convolutional Generative Adversarial Network wird verwendet [7].

Nachdem das Training abgeschlossen ist, wird das Diskriminator-Netz nicht mehr benötigt und lediglich das Generator-Netz wird genutzt. Um mit dem Generator-Netz zu interagieren, werden die zwei Funktionen aus List. 4.2 benötigt. Als Eingabe wird ein Rauschen benötigt, welches in Form eines 1D-Arrays mit 100 Werten sein muss. Hierfür kann die TensorFlow Funktion *random.normal()* genutzt werden, welche normalverteilte Werte generiert. Um ein Bild daraus zu erzeugen, wird die Funktion *generator()* genutzt. Hier ist darauf zu achten, dass das Parameter training auf False gesetzt wird, da sonst das Netz weiter trainiert werden würde, was nicht gewünscht ist.

```
1 input = tensorflow.random.normal([1, 100])
2 output = generator(input, training=False)
```

List. 4.2: Input/Output Funktionen Generator-Netz

Ein Training mit 250 Epochen dauert sehr lange, weswegen es hilfreich ist das trainierte Modell abzuspeichern und wieder zu laden, anstatt jedes mal neu zu trainieren. Hierfür wird standardmäßig alle 15 Epochen ein Checkpoint gespeichert. Wird der letzte Checkpoint heruntergeladen, kann mithilfe von *checkpoint.restore()*, wie in List. 4.3 zu sehen, das Modell erneut geladen werden. Wurde ein zufriedenstellendes Modell trainiert, kann auf diese Weise jedes mal circa eine Stunde Zeit gespart werden.

```
1 checkpoint.restore(tensorflow.train.latest_checkpoint(PathToModel))
```

List. 4.3: Modell des Generator-Netz speichern/laden

4.2 Erkennungs-Netz

Als Erkennungs-Netz kommt ein Convolutional Neural Network zum Einsatz. Hierfür wird der Quelltext von Rajpurohit Dhanpal Singh als Grundlage verwendet [15]. Bis auf die Anpassung der Epochen-Anzahl, wird hier nichts verändert.

Zusätzlich wird eine Funktion für die Klassifizierung implementiert. Die Funktion *classify(image)*, wie sie im List. 4.4 zu sehen ist, nimmt als Eingabe ein Bild, welches vom Erkennungs-Netz klassifiziert werden soll. Als Rückgabewert kommt die Klasse des Bildes. Die zweite Funktion *getEachLayerOutput(image)* nimmt erneut ein Bild, gibt allerdings nicht die Klasse zurück, sondern speichert die Ausgabe aller sechs Ebenen aus dem Netz und speichert sie in einem großen Array, welches dann zurückgegeben wird. Während die erste Funktion primär für die Erzeugung von False Positive Kandidaten benötigt wird, ist die zweite Funktion wichtig für die Coverage-Messung.

```
1 def classify(image):
2     pred_array = model.predict(image)
3     pred_array = numpy.argmax(pred_array)
4     return pred_array
5
6 def getEachLayerOutput(image):
```

```
    inp = model.input
    outputs = [layer.output for layer in model.layers]
    functors = [K.function([inp], [out]) for out in outputs]
    layer_outs = [func([image]) for func in functors]
    return layer_outs
```

List. 4.4: Hilfsfunktionen Erkennungs-Netz

4.3 False Positive Kandidaten

Um Eingaben zu generieren wird zunächst ein Rauschen benötigt. In List. 4.5 ist die verwendete Funktionen zu sehen. Diese Funktion hat lediglich den Zweck ein Rauschen zu erzeugen, welches als eine andere Zahl klassifiziert wird, als die Zahl die als Parameter mitgegeben wird. Wenn Kandidaten für eine bestimmte Zahl gesucht werden, dann wird hierdurch die einzige Bedingung umgesetzt. Die Bedingung ist, dass weder Start- noch Zielpunkt für die Interpolation genau dieser Zahl entsprechen. Falls grundsätzlich nach Kandidaten gesucht werden soll, können Start- und Zielpunkt jeweils mit *random.normal()* erstellt werden, ohne darauf zu achten, als was die Bilder klassifiziert werden.

```
def getNoiseDifferentToNumber(number):
  while True:
    noise = tensorflow.random.normal([1, 100])
    classified = classify(generator(noise, training=False))
    if classified != number:
      return noise
```

List. 4.5: Noise generieren

Die Interpolation wird auf zwei Weisen umgesetzt. Da die Suche nach einem Kandidaten für eine bestimmte Zahl eine andere Abbruchfunktion benötigt, als die generelle Suche. In List. 4.6 ist der Ablauf der Interpolation zu erkennen. Und zwar werden die zwei Rauschen zuerst voneinander subtrahiert, um deren Abstand im Raum herauszufinden. Nun wird die Interpolation in 100 Schritten durchgeführt. Das heißt es wird auf das eine Rauschen jedes mal ein Hundertstel des Abstands addiert. Aus der Summe, welche auch wieder ein Rauschen ist, wird ein Bild generiert und dann klassifiziert. Sollte die neue Klassifikation der gesuchten Zahl entsprechen, wird der Durchlauf beendet. Dann können beispielsweise das Bild und benötigte Werte gespeichert werden.

```
def interpolate(noise1, noise2, number):
    tensorDiff = noise2 - noise1
    noiseTensor = tensorflow.Variable(noise1)

    for add in range(0,100):
        noiseTensor = tensorflow.add(noiseTensor, tensorDiff/100)
        generated_image = generator(noiseTensor, training=False)
```

```
8
9           if classify(generated_image) == number:
10            return 0
11
12      return 1
```

List. 4.6: Interpolation gezielte Suche

Die allgemeine Suche nach Kandidaten ist, wie in List. 4.7 zu sehen ist, sehr ähnlich aufgebaut, aber benötigt mehr Informationen. Da die übergebenen Parameter zwei Rauschen sind, müssen diese erst dem Generator-Netz gegeben werden, dass dieses daraus Bilder generiert, welche dann klassifiziert werden. Die Interpolation funktioniert gleich wie im List. 4.6, mit dem Unterschied, dass hier beendet wird, sofern bei einem Schritt eine andere Zahl, als die zwei am Anfang klassifizierten, gefunden wird.

```
1  def interpolate2(noise1, noise2):
2      number1 = classify(generator(noise1, training=False))
3      number2 = classify(generator(noise2, training=False))
4
5      tensorDiff = noise2 - noise1
6      noiseTensor = tensorflow.Variable(noise1)
7
8      for add in range(0,100):
9          noiseTensor = tensorflow.add(noiseTensor, tensorDiff/100)
10         generated_image = generator(noiseTensor, training=False)
11         numberGen = classify(generated_image)
12
13         if numberGen != number1 and numberGen != number2 :
14           return 0
15
16     return 1
```

List. 4.7: Interpolation allgemeine Suche

Die Interpolation ist immer nur ein Durchlauf von 100 Schritten für je zwei Rauschen. Das ganze muss nun in eine Schleife gepackt werden, um das Ganze so lange zu wiederholen, bis ein Treffer gefunden wird. In List. 4.8 ist ein Beispiel für die Zahlensuche zu sehen. Solange kein Kandidat gefunden ist, werden zwei neue Rauschen zufällig generiert und dann wird zwischen diesen beiden interpoliert.

```
1  def findNumber(number):
2      while True:
3        noiseStart = getNoiseDifferentToNumber(number)
4        noiseZiel = getNoiseDifferentToNumber(number)
5        ret_val = interpolate(noiseStart, noiseZiel, number)
6
7      if ret_val == 0:
```

```
8             break
```

List. 4.8: Interpolation-Schleife

4.4 Coverage-Messung

Die Coverage-Messung benötigt zunächst einen Weg zum Speichern der bereits aktiv gewesenen Neuronen. Hierfür wird ein Boolean Array erstellt, welches exakt so groß wie Anzahl der zu untersuchenden Neuronen ist. Im Array werden alle Werte als False initialisiert und sobald ein Neuron als aktiv ermittelt wird, wird der Wert bei dem exakt gleichen Index auf True gesetzt. In List. 4.9 ist zu sehen, wie nun daraus die Coverage ausgelesen werden kann und das unabhängig davon, wie aktive Neuronen ermittelt werden. Hierzu wird mit der Numpy Funktion *sum()* gezählt wie viele Werte bereits auf True sind. Diese Anzahl wird durch die Anzahl der Neuronen geteilt und mal 100 genommen, um einen Prozentwert zu erhalten.

```
1 def getCurrentCoverage():
2     count = numpy.sum(coverageArray)
3     cov = count / len(coverageArray) * 100
4     return cov
```

List. 4.9: Momentane Coverage ermitteln

Für die Auswertung der Neuronen wird die zweite Hilfsfunktion aus dem List. 4.4 benötigt. Mit der Ausgabe aller Ebenen können nun einzelne Neuronen betrachtet werden. Im Fall von der generellen Suche, wie in List. 4.10 zu sehen ist, wird mit Hilfe von Numpy und der Funktion *argwhere()* herausgefunden an welchen Indexen der Schwellenwert überschritten wird. Diese Liste wird dann abgearbeitet und die entsprechenden Indexe im Coverage Array als aktiv gespeichert.

```
1 def getOverValues(arr):
2     result_args = numpy.argwhere(arr > schwellenwert)
3
4     for e in result_args:
5       coverageArray[e] = True
```

List. 4.10: Aktive Neuronen über einem Schwellenwert ermitteln und Coverage aktualisieren

Die X-höchsten Werte sind sehr ähnlich zu ermitteln, hier ändert sich lediglich die Numpy Funktion *argwhere()* in *argpartition()*. Die restliche Funktion ist gleich aufgebaut.

4.5 Zusammenführung

In Abb. 4.1 ist der momentane Programmablauf zu sehen. Da die Schleife für Coverage Guided Fuzzing nicht geschlossen werden konnte, arbeitet der Noise Generator per Zufall. Wenn die Schleife geschlossen wäre, würde ein Pfeil von „Coverage Analyzer“ zum „Noise Generator gehen“.

Die eigentliche Logik steckt im „Candidate Searcher“. Hier wird die Interpolation durchgeführt und je nach Bedarf ein neues Rauschen angefordert, Bilder aus Rauschen generiert oder ein Bild klassifiziert. Außerdem werden die Ergebnisse, falls ein Kandidat gefunden wurde, für die Überprüfung der aktiven Neuronen an den Coverage Analyzer übergeben. Die Umsetzung ist in Kap. 4.3 erläutert worden.

Jeder Durchlauf beginnt mit dem „Noise Generator“, welcher zu Beginn des Kapitels Kap. 4.3 implementiert wird und lediglich aus der Funktion *getNoiseDifferentToNumber()* besteht. Alternativ bei einer allgemeinen Suche nur aus der Funktion *random.normal()*, was in Kap. 4.1 bereits erläutert wurde.

Im nächste Schritt muss im „Image Generator“ aus dem Rauschen ein Bild erzeugt werden. Dies wird durch das Generator-Netz umgesetzt. Klassifizierung dieses Bildes findet im „Neural Network“ statt, welches das trainierte Erkennungs-Netz aus Kap. 4.2 ist. Der „Coverage Analyzer“ ist die Coverage-Messung aus Kap. 4.4.

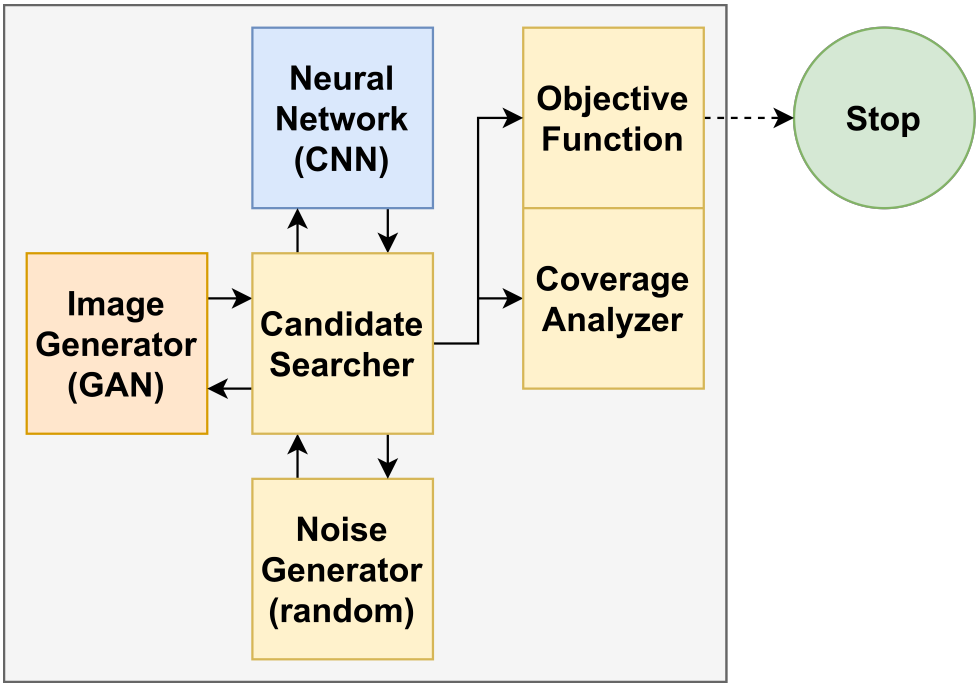

Abb. 4.1: Programmablauf

Die „Objective Function“ variiert je nach gewünschtem Ziel. Bei der Suche nach Kandidaten, beendet sie das Programm, sobald die gewünschte Anzahl an Kandidaten gefunden

ist. Alternativ kann bei auch einer erreichten Coverage beendet werden.

Da die einzelnen Module bereits unabhängig funktionieren, müssen lediglich die Funktionsaufrufe an den richtigen Stellen erfolgen. So muss in den Prozess der Interpolation aus Kap. 4.3 nach dem Finden eines Kandidaten die Coverage-Funktion aus Kap. 4.4 aufgerufen werden. Wenn nun die Abbruchbedingung auf eine Coverage umgestellt wird, ist der gesamte Programmablauf fertig.

In Abb. 4.2 ist das Ergebnis nach einem Durchlauf zu sehen. Hier werden einerseits generelle Informationen zu dem Durchlauf aufgezählt, als auch der letzte gefundene Kandidat gezeigt. Das Bild wurde als eine 7 klassifiziert und ist bei der Interpolation von einer 9 zu einer 8 gefunden worden. Insgesamt wurden 428 Interpolation durchgeführt, was circa 4173 Sekunden gedauert hat. Des Weiteren ist eine Coverage von ungefähr 75,78% erreicht worden.

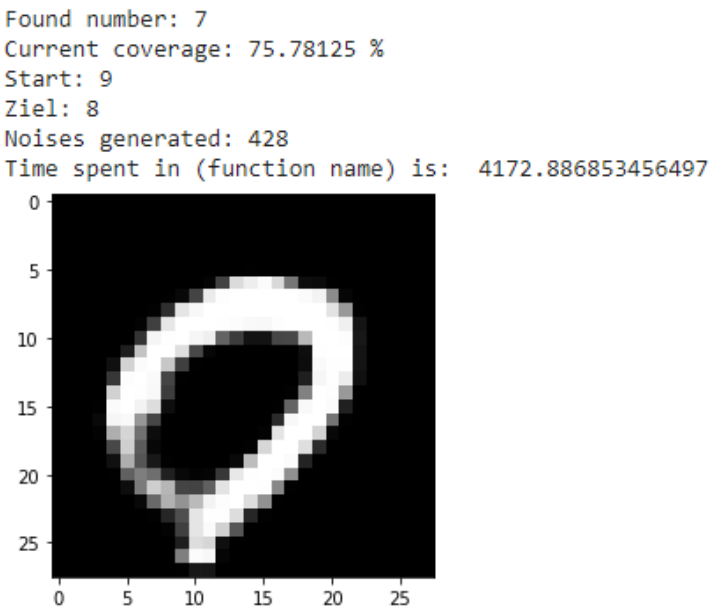

Abb. 4.2: Beispiel Ausgabe

5 Evaluation

Das Problem False Positives zu finden, ist nicht das eigentliche Generieren, sondern diese zu erkennen. Bei TensorFuzz sind die Eingaben völlig zufällig, was es nahezu unmöglich macht False Positives gezielt zu finden, sofern sie nicht maschinell erkannt werden können. Die einzige Möglichkeit für die Erkennung wäre, wenn im neuronalen Netz Muster zu erkennen sind, welche bei allen False Positives gleich wären. Ohne eine derartige Erkennung, müsste jedes erzeugte Bild manuell durch einen Benutzer überprüft werden, was den gesamten Prozess ineffizient macht. Selbst mit einer funktionierenden Erkennung muss die Effizienz von zufälligen Eingaben erst überprüft werden, da beispielsweise ein Generator-Netz zwar Bilder aus einem endlichen Raum generiert, aber dieser Raum dennoch so groß ist, dass es sehr lange dauern kann. Aufgrund fehlender maschineller Erkennung von False Positives wird die Methode der Interpolation implementiert. Der Vorteil zu TensorFuzz und reinem Zufall ist, dass nicht jedes Bild überprüft werden muss, sondern bereits eine sehr gute Vorauswahl getroffen wird.

In der ersten Variante des Algorithmus werden nach jeder Interpolation neue Start- und Zielpunkte ausgewählt, ohne darauf zu achten, wie diese klassifiziert werden, was sehr viele Kandidaten generierte, wenn durch Zufall einer der beiden Punkte die gesuchte Zahl waren. Mit der Bedingung, dass die Interpolationspunkte einer anderen Klasse entsprechen, zeigten sich erste Erfolge auf. Hierbei war allerdings die Beschränkung, dass immer nach einer spezifischen Zahl gesucht werden muss. Um ohne spezifische Suche Kandidaten zu finden, musste zunächst immer eine zufällige Zahl generiert werden, nach der gesucht wird. In der letzten Änderung werden nun alle klassifizierten Bilder als Kandidaten ausgewählt, wenn die Klasse weder dem Start- noch dem Zielpunkt entspricht. So kann zwar nicht mehr nach spezifischen Klassen gesucht werden, allerdings können so deutlich schneller Kandidaten gefunden werden, welche mit sehr ähnlicher Trefferquote False Positives sind.

Da kein vollständiger Coverage Guided Fuzzing Prozess erreicht wurde, werden das Problem der False Positive Findung und das Problem der Coverage Messung einzeln betrachtet. So soll mit Hilfe von verschiedenen Versuche gezeigt werden, ob es dennoch möglich ist False Positives gezielt zu generieren.

Für ein besseres Verständnis, wie ein False Positive definiert wird, sind in Abb. 5.1 vier Kandidaten zu sehen. Mit diesem Beispiel wird verdeutlicht, was nun als False Positive zählt und was nicht. Von links nach rechts, sind die ersten zwei Bilder als richtig klassifiziert und die anderen beiden als falsch klassifiziert eingeordnet werden. Das erste Bild wird als 4 erkannt, was bei der Überprüfung bestätigt wird, da für eine 9 der obere Kreis geschlossen sein müsste. Im letzten Bild wird eine 5 erkannt, was als falsch klassifiziert eingeordnet

wird, da eindeutig eine 6 auf dem Bild zu sehen ist. Die beiden mittleren Bilder sind nicht so eindeutig. Hier liegt es im ermessen der Person, die die Kandidaten überprüft, ob es als False Positive gewertet wird oder eben nicht. Das zweite Bild wird als 8 erkannt, was akzeptiert wird, da der Kreis in der Mitte zusammen geht. Für eine Null müsste der Kreis runder sein. Das dritte Bild wird als 3 eingestuft. Hier wird es als False Positive eingestuft, da eher eine 7 zu erkennen ist. Für eine 3 müsste der untere Bogen deutlicher sein.

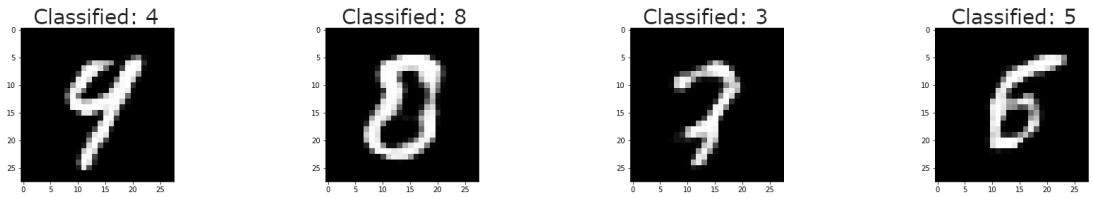

Abb. 5.1: Beispiele für Kandidaten, klassifiziert als 4, 8, 3, 5 (links nach rechts)

Bei unterschiedlichen Personen, könnten somit unterschiedliche Trefferquoten erreicht werden. Aus diesem Grund werden alle Kandidaten von der selben Person überprüft.

5.1 False Positives Problem

Das False Positive Problem bezieht sich auf die Erzeugung und das Finden von False Positive Kandidaten. Hierfür werden zwei Varianten des entwickelten Algorithmus verglichen. Variante 1 ist die spezifische Zahlensuche, bei der False Positive Kandidaten für nur eine ausgewählte Zahl gesucht werden. Variante 2 ist die allgemeine Suche nach False Positive Kandidaten. Der Vergleich findet bei der benötigten Anzahl an Interpolationen, der benötigten Zeit und der Trefferquote statt. Die Trefferquote definiert sich daraus, wie viele Kandidaten gefunden werden und wie viele davon False Positives sind. Außerdem wird der Zusammenhang mit der Genauigkeit des Erkennungs-Netzes untersucht. Dafür wird die Epochen-Anzahl des Trainings variiert. Beim Generator-Netz wird von einer Trainingsvariation abgesehen, da schlechtere Bilder hier bedeuten, dass für den Mensch keine Zahl mehr erkennbar ist und somit definitiv ein False Positive erzeugt wird.

Für die Übersichtlichkeit gibt es zwei Testsets und einen abschließenden Vergleich. Testset 1 ist für die spezifische Zahlensuche. Hier wird zusätzlich die Zeit gemessen, wie lange es dauert, bis ein False Positive (kein Kandidat) gefunden wird. In Testset 2, in dem es um die allgemeine Suche geht, wird zusätzlich die Anzahl der Kandidaten pro Zahl angegeben.

Die Aussagekraft der Anzahl von Interpolationen und der Zeit sind geringer als die der Trefferquote anzusetzen, da es keinen geschlossenen Coverage Guided Fuzzing Prozess gibt und somit die Start-/Zielpunkte zufällig gewählt werden. Dennoch ist dadurch eine Richtung zu erkennen, welche Zahlen besonders anfällig sind und welche zuverlässig erkannt werden.

5.1.1 Spezifische Zahlensuche

In diesem Testset werden pro Zahl je fünf Kandidaten gesucht und ausgewertet. Alle Zahlen in der Tab. 5.1 sind der Durchschnitt aus den fünf Durchläufen. Für diesen Versuch wurde das Erkennungs-Netz 50 Epochen lang trainiert und hat bei der Validierung eine Genauigkeit von 98,72% erreicht. Für den zweiten Versuch, dessen Ergebnisse in Tab. 5.2 zu sehen sind, wurde das Netz 250 Epochen trainiert und erreichte eine Genauigkeit von 98,65%. Die Genauigkeit täuscht in diesem Fall allerdings, da für diese Messung dem Netz eine bestimmte Anzahl Bilder, auch Validierungsdaten genannt, gegeben werden und basierend darauf wie viele davon richtig klassifiziert werden, wird die Genauigkeit bestimmt. Diese Art der Messung spiegelt allerdings nicht zwingend wider, ob dies auch der Fall ist, wenn die Anzahl der Validierungsdaten erhöht wird. Im ersten Versuch mit 50 Epochen wird eine durchschnittliche Trefferquote von 74% erreicht, während im zweiten Versuch nur noch eine Trefferquote von 68% im Schnitt erreicht wird. Es deutet somit darauf hin, dass ein intensiveres Training des Erkennungs-Netzes weniger Bilder falsch klassifiziert. Ein Faktor für dieses Ergebnis kann die geringe Anzahl an Durchläufen sein, wobei deutlich größere Untersuchungen zeitlich den Rahmen dieser Arbeit bei weitem überschreiten würden. Es muss jeder Kandidat, aufgrund fehlender maschineller Unterscheidung manuell überprüft werden, was für große Zahlen an Durchläufen sehr schnell sehr viel Aufwand bedeutet.

Den Ergebnissen der Messung lässt sich entnehmen, dass besonders die Zahlen 3 und 5 oft klassifiziert werden, obwohl andere Zahlen auf dem Bild zu sehen sind. Besonders in Versuch 2 fällt die Zahl 3 aus der Reihe, da alle fünf Durchläufe nach durchschnittlich 1,6 Interpolationen bereits beendet waren. Dazu kommt, dass auch alle Durchläufe wirkliche False Positives waren.

Grundsätzlich sind bis auf einzelne Ausreißer die meisten Werte im Durchschnitt und die Trefferquote liegt insgesamt um die 70%.

Zahl	Anz. Interpolationen	Trefferquote	Zeit bis Treffer
0	25,0	60%	913,37s
1	34,0	80%	178,95s
2	33,2	100%	178.27s
3	12,4	100%	276,59s
4	35,0	40%	548,79s
5	30,6	100%	121,63s
6	28,0	60%	683,25s
7	42,2	60%	124,41s
8	23,2	60%	610,97s
9	31,2	80%	89,63s

Tab. 5.1: Testset 1 Versuch 1: je fünf Durchläufe spezifische Zahlensuche, Generator-Netz 250 Epochen, Erkennungs-Netz 50 Epochen

Zahl	Anz. Interpolationen	Trefferquote	Zeit bis Treffer
0	23,4	60%	727,69s
1	25,2	40%	1153,71s
2	12,6	80%	222,19s
3	1,6	100%	17,43s
4	13,2	60%	399,68s
5	29,6	100%	264,56s
6	42,6	40%	555,09s
7	30,0	40%	574,96s
8	41,0	80%	161,85s
9	44,8	80%	83,52s

Tab. 5.2: Testset 1 Versuch 2: je fünf Durchläufe spezifische Zahlensuche, Generator-Netz 250 Epochen, Erkennungs-Netz 250 Epochen

5.1.2 Allgemeine Suche

Auch dieses Testset besteht aus zwei Versuchen. Es werden pro Versuch je 20 Durchläufe durchgeführt. In Versuch 1 wird das Erkennungs-Netz mit 50 Epochen verwendet und in Versuch 2 das mit 250 Epochen Trainingszeit. Wie in Tab. 5.3 kann es sein, dass nach 20 Durchläufen nicht für jede Zahl ein Kandidat gefunden wird, was allerdings dem Zufall geschuldet ist. Da die Start- und Zielpunkte zufällig generiert werden, kann es sein, dass bei der Interpolation bestimmte Zahlen nicht erreicht werden können. In Tab. 5.4 ist zu sehen, dass im 2. Versuch für die Zahl 1 kein Kandidat gefunden wird. In Versuch 1 wurden für die Zahlen 4 und 7 keine Kandidaten gefunden.

Zahl	Kandidaten	Treffer	Trefferquote
0	1	0	0%
1	1	1	100%
2	1	1	100%
3	1	1	100%
4	-	-	-
5	5	4	80%
6	1	0	0%
7	-	-	-
8	2	1	50%
9	8	6	75%

Tab. 5.3: Testset 2 Versuch 1: 20 Durchläufe generelle Suche, Generator-Netz 250 Epochen, Erkennungs-Netz 50 Epochen

Beide Versuche haben für die Kandidaten im Schnitt 70% Trefferquote und sind somit im gleichen Bereich wie Testset 1. Besonders bei den Zahlen für die fünf oder mehr Kandidaten

gefunden wurden, ähneln die Trefferquoten denen aus Testset 1. Wenn nur ein oder zwei Kandidaten gefunden wurden, ist die Trefferquote deutlich weniger zu gewichten, da der Probenumfang zu gering ist.

Zahl	Kandidaten	Treffer	Trefferquote
0	-	-	-
1	1	0	0%
2	2	1	25%
3	5	5	100%
4	1	0	0%
5	1	1	100%
6	2	1	25%
7	1	1	100%
8	2	2	100%
9	5	3	60%

Tab. 5.4: Testset 2 Versuch 2: 20 Durchläufe generelle Suche, Generator-Netz 250 Epochen, Erkennungs-Netz 250 Epochen

5.1.3 Vergleich

Werden nun Testset 1 und 2 in den direkten Vergleich gezogen, sind starke Ähnlichkeiten vorhanden. In Tab. 5.5 ist zu sehen, dass sowohl die Anzahl benötigter Interpolationen, als auch die Trefferquote im gleichen Größenverhältnis. Die Zeit bei der spezifischen Suche ist deutlich höher, da nicht jeder Kandidat verwendet werden kann, sondern eben erst wenn die spezifische Zahl gefunden wird.

Bezüglich Zeit und benötigte Interpolationen ist ein ähnliches Muster zwischen den verschiedenen Erkennungs-Netzen zu erkennen. Bei beiden Testsets hat das kürzer trainierte Netz schneller einen Treffer hervorgebracht, während es aber mehr Interpolationen benötigt hat, was im Widerspruch steht. Mehr Interpolationen bedeutet mehr Rechenaufwand und sollte eigentlich mehr Zeit benötigen. Entweder kommen die Zeiten daher, dass der Server unterschiedlich ausgelastet war oder es liegt an anderen Faktoren, welche nicht ermittelt werden konnten.

Variante	spezifische Zahlensuche		generelle Suche	
Anz. Epochen Erkennungs-Netz	50	250	50	250
Zeit bis Treffer	394,18s	416,07s	17,86s	22,00s
Anz. Interpolationen	29,5	26,4	27,6	22,4
Trefferquote	74%	68%	70%	70%

Tab. 5.5: Vergleich spezifische Zahlensuche und generelle Suche

Zusammengefasst ist durch diese Versuche deutlich zu sehen, dass das Erkennungs-Netz bei 70% der gefundenen Kandidaten eine falsche Klassifizierung durchgeführt hat. Alle Kandidaten müssen manuell überprüft werden, was den Vorgang nicht komplett automatisieren lässt. Dennoch kann mit der Trefferquote eine gewissen Stabilität eines Netzes nachgewiesen werden. Je höher die Trefferquote ist, desto schlechter ist die Klassifizierung. Eine 100%ige Trefferquote heißt aber nicht direkt, dass die Klassifizierung vollends versagt, sondern lediglich, dass False Positives durchaus auftreten können. Eine 0%ige Trefferquote sagt auch nicht aus, dass es keine False Positives gibt, sondern, dass mit diesem Algorithmus keine gefunden werden können. Für eine genaue Analyse des Netzes sollte eine große Stichprobe erfasst werden. Ab einer Stichprobengröße von 500 wird sie ansatzweise repräsentativ, wobei sie erst ab einer Größe von circa 10.000 wirklich repräsentativ ist. Vorher kann die Streuung der Werte zu groß sein, was zu verfälschten Ergebnissen führt. Durch Datenbereinigung, also dem entfernen von Extremwerten, kann eine kleinere Stichprobe unter Umständen auch repräsentativ sein [16].

False Positives werden nicht direkt generiert. Es können jedoch schnell Kandidaten gefunden werden, welche mit hoher Trefferquote False Positives sind. Kandidaten müssen noch manuell geprüft werden, aber sollte hierfür ein Algorithmus gefunden werden, kann der gesamte Prozess automatisiert werden. Auch mit manueller Überprüfung kann eine Analyse der Schwachpunkte des Netzes durchgeführt werden. So kann mit entsprechender Stichprobengröße erkannt werden, welche Zahlen öfter falsch klassifiziert werden und dementsprechend kann das Training angepasst werden.

5.2 Coverage Analyse

Die Arbeiten „TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing“ [3] und „Testing Deep Neural Networks“ [17] zeigen auf, dass es schwierig ist die Coverage bei neuronalen Netzen zu messen. Besonders die Aussagekraft der gemessenen Werte ist nicht immer hoch. Zunächst wurden in Kap. 3.1 drei Methoden ausgewählt, welche umgesetzt werden sollen. Methode 1 ist, dass für jede Zahl ein Kandidat gesucht wird und die Coverage 100% erreicht, sobald dies erfüllt ist. Das bringt allerdings keinerlei Aussage über das Netz mit sich. Wie in Kap. 5.1 bereits gezeigt wird, muss der Stichprobenumfang groß genug sein, um überhaupt eine Aussage treffen zu können. Somit ist diese Methode ungeeignet, um eine Coverage zu messen und wird nicht näher betrachtet.

Methode 2 und 3 analysieren die Ergebnisse einzelner Ebenen des Erkennungs-Netzes. In Methode 2 werden alle Neuronen als aktiv gewertet die zu den höchsten Werten zählen. In Methode 3 werden nicht nur die Top-Werte gewertet, sondern alle die einen gewissen Schwellenwert überschreiten.

In Abb. 5.2 ist ein beispielhaftes Netz mit sechs Ebenen zu sehen. Während die rote Ebene die Eingabe und die blaue Ebene die Ausgabe ist, sind die orangenen Ebenen die Hidden-Ebenen. Die vorletzte Ebene ist die, die betrachtet wird, da es eine Dropout-Ebene ist.

Bei einer Dropout-Ebene werden zufällige Neuronen deaktiviert. Bevor dies geschieht, soll analysiert werden, welche Neuronen aktiv sind.

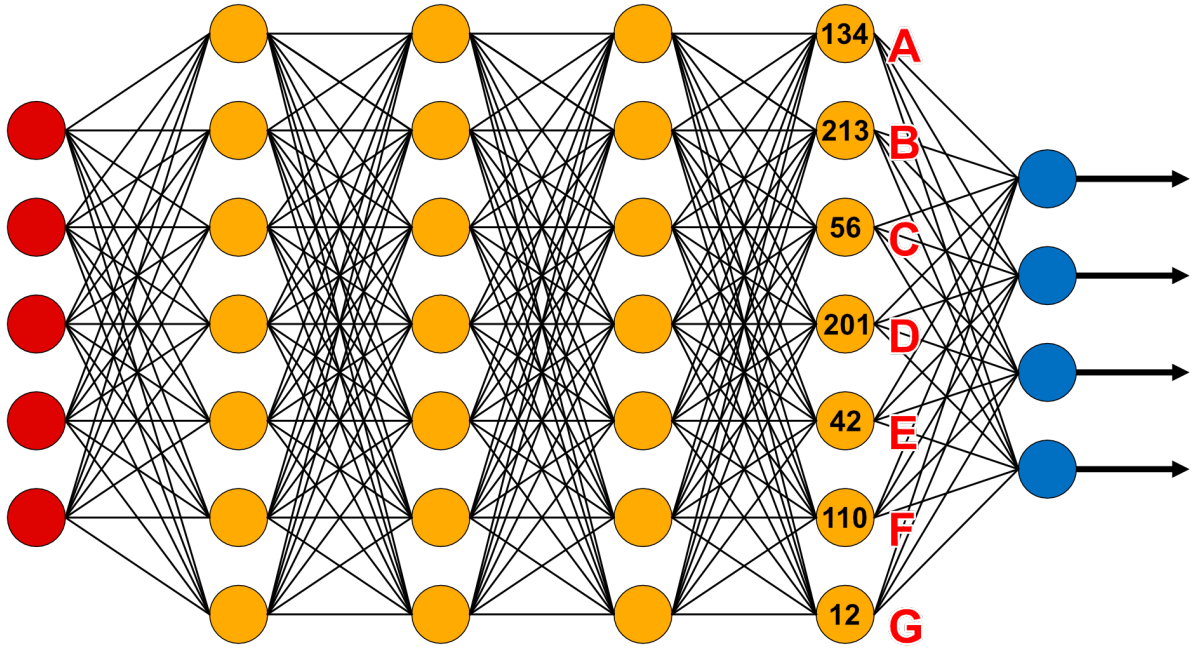

Abb. 5.2: Beispiels-Netz mit sechs Ebenen

5.2.1 Top-Werte

Für die Coverage Messung anhand der höchsten Werte werden sowohl das Generator- als auch der Erkennungs-Netz 250 Epochen lang trainiert. Das Erkennungs-Netz arbeitet mit sechs Ebenen. Für den Versuch wird die vorletzte Ebene analysiert. Diese ist ein Dropout-Ebene und führt am Ende zur Klassifizierung. Sie besteht aus 128 Neuronen. Wenn nun ein Bild als Eingabe an das Netz weitergegeben wird, kommt bei dieser Ebene in der Ausgabe ein Array aus 128 Werten die von 0 aufwärts bis circa 3.000 reichen. Wenn ein Kandidat gefunden wird, werden aus diesem Array nun beispielsweise die zehn höchsten Werte ermittelt und dann als „bereits aktiv“ gewertet und gespeichert. Sollten nun alle Werte einmal unter den zehn höchsten gewesen sein, ist eine Coverage von 100% erreicht.

Die Anzahl wie viel höchste Werte gewertet werden, kann variiert werden. Wie in Tab. 5.6 zu sehen ist, gab es drei Durchläufe. Einmal mit den zehn höchsten, dann mit den fünf höchsten und als letztes mit den drei höchsten Werten. Ein Durchlauf gilt als beendet, wenn eine Coverage von mindestens 75% erreicht wird.

Im ersten Durchlauf mit den zehn höchsten Werten wird nach 26 Minuten eine Coverage von 75% erreicht. In dieser Zeit sind 328 Kandidaten gefunden worden, welche zu 70,12% False Positives sind.

Mit den fünf höchsten Werten dauert es bereits 98 Minuten. In dieser Zeit wurden 328 Kandidaten gefunden, welche zu 69,15% False Positives sind.

Der Durchlauf mit den drei höchsten Werten musste abgebrochen werden, da nach über vier Stunden noch immer keine Coverage von 75% erreicht wird und die Testserver keine längeren Sessions zulassen. Jedoch ist auch eine derart lange Dauer für eine Coverage von 75% sehr hoch und würde so vermutlich nicht zum Einsatz kommen. Im Beispiel der Abb. 5.2 würden bei den drei höchsten Werten die Neuronen A, B und D als aktiv bezeichnet werden.

Variante	Dauer bis >75%	Kandidaten	Trefferquote
Top 3	>4h	-	-
Top 5	~98min	328	70,12%
Top 10	~26min	94	69,15%

Tab. 5.6: Coverage Versuch 1: Top X Werte, bis Coverage >75%, Generator-Netz 250 Epochen, Erkennungs-Netz 250 Epochen

Es wird außerdem ein Durchlauf mit den zehn höchsten Werten gestartet, um zu schauen wie lange es dauert 100% Coverage zu erreichen. Nach circa zwei Stunden wird eine Coverage von 93,75% erreicht, welche sich in den folgenden zwei Stunden nicht mehr änderte. Somit war eine 100% Coverage nicht erreichbar.

5.2.2 Schwellenwert

Ähnlich zu der Methode mit Top-Werten wird dieser Versuch mit den gleichen Netzen und gleichen Voraussetzungen durchgeführt. Der einzige Unterschied ist, dass alle Werte über einem gewissen Schwellenwert als aktiv gewertet werden.

In Tab. 5.7 ist zu sehen, dass als Schwellenwert von 0 bis 2.000 in 500er Schritten je ein Durchlauf gemacht wird. Einzig der Durchlauf mit Werten über 2.000 hat länger als eine Viertelstunde gedauert und genug Kandidaten hervorgebracht, um Aussagen über das Netz treffen zu können. In Abb. 5.2 würden bei einem Schwellenwert von 100 die Neuronen A, B, D und F als aktiv bezeichnet werden.

Variante	Dauer bis >75%	Kandidaten	Trefferquote
Werte >2000	~70min	249	67,87%
Werte >1500	~14min	58	81,03%
Werte >1000	~7min	36	63,89%
Werte >500	~4min	12	75,00%
Werte >0	36s	5	80,00%

Tab. 5.7: Coverage Versuch 2: Werte größer X, bis Coverage >75%, Generator-Netz 250 Epochen, Erkennungs-Netz 250 Epochen

5.2.3 Vergleich

Methode 2 und 3 sind von der Funktionsweise und den Ergebnissen sehr ähnlich. Ganz grob lässt sich sagen, dass die fünf höchsten Werte im Schnitt einen Wert von etwas über 2.000 haben. Die zehn höchsten Werte sind im Schnitt zwischen 1.500 und 2.000. Beide Methoden sind gleichermaßen gut geeignet. Sie sollten allerdings so gewählt werden, dass entweder die gewünschte Coverage höher gesetzt wird oder die Hürde bis ein Neuron als aktiv gilt so hoch angesetzt wird, dass die Anzahl der Kandidaten hoch genug ist, um repräsentative Analysen des Netzes durchführen zu können.

5.3 Aussagekraft

Gut zu sehen ist, dass auch bei höheren Stichproben aus Kap. 5.2 die Trefferquote um die 70% liegt, was Kap. 5.1 somit trotz der geringen Stichprobe bestätigt und die geringeren Stichproben in diesem Fall rechtfertigt.

Zu beachten ist, dass die Trefferquote nicht mit der Coverage zusammenhängt. Bei jedem gefundenen Kandidaten werden die aktivsten Neuronen gewertet, auch wenn nach Überprüfung kein False Positive ist. Somit hat die Coverage durch die fehlende maschinelle Erkennung von False Positives eine geringere Aussagekraft.

Der Prozess der Coverage-Messung ist jedoch trotzdem hilfreich für die Analyse des neuronalen Netzes. So können alle gefundenen Kandidaten ausgewertet und analysiert werden, um gegebenenfalls Probleme in der Klassifizierung zu finden. Solange keine maschinelle Erkennung der False Positives vorhanden ist, kann dieser Prozess allerdings ohne eine Coverage-Messung auskommen, in dem eine gewünschte Anzahl an Kandidaten gesucht wird. So kann der Stichprobenumfang frei gewählt werden.

Sobald durch Erkennung wirklich False Positives und nicht nur Kandidaten gefunden werden, kann mit diesen Werten unter Umständen auch die Schleife für Coverage Guided Fuzzing geschlossen werden. Sodass der gesamte Prozess gezielter und schneller durchgeführt werden kann und die gemessene Coverage eine höhere Aussagekraft hat.

6 Abschluss

6.1 Fazit

Da es zu keiner Übereinstimmung zwischen Coverage und einem finalem Ergebnis kam, ist es sehr unwahrscheinlich, dass ein Master Key existiert. Allerdings wurde ein Algorithmus entwickelt, welcher mit einer recht hohen Trefferquote von ~70% False Positives finden kann.

Wenn zwischen zwei Punkten im Raum des Generator-Netzes interpoliert wird, kommt es vor, dass Zahlen klassifiziert werden, welche weder dem Start- noch dem Zielpunkt entsprechen. Diese Fälle sind mögliche Kandidaten für False Positives. Auf diese Art werden False Positives generiert, was den ersten Teil der Frage WF1.1 beantwortet. Auf den zweiten Teil gibt es nur die Antwort, dass sie nicht maschinell erkannt werden, sondern von einem Benutzer überprüft werden müssen.

Frage WF1.2 ist nur schwer zu beantworten. Die angeschauten Methoden ermöglichen es eine Coverage zu bestimmen, indem aktive Neuronen analysiert werden. Jedoch ist die Aussagekraft sehr gering, da kein Vorteil aus der Coverage gezogen werden kann. Ohne geschlossene Coverage Guided Fuzzing Schleife ist es schwer eine Coverage-Messung zu bewerten, weil es keine Parameter gibt, an denen gemessen werden kann, welche Coverage-Messung nun besser geeignet ist.

Zusammenfassend lässt sich die Frage WF1 damit beantworten, dass mittels Interpolation zügig Kandidaten gefunden werden können, aber durch fehlende maschinelle Erkennung ist der Prozess nur bedingt effizient. Wäre eine solche Erkennung möglich, könnte der gesamte Prozess optimiert werden. So würde auch die manuelle Überprüfung entfallen und die Entscheidung, was noch als False Positive zählt und was nicht, würde nicht mehr dem Benutzer überlassen werden.

Mit dem Algorithmus wird eine Trefferquote von über 50% erreicht, was sehr zufriedenstellend ist. Coverage Guided Fuzzing konnte nicht vollends umgesetzt werden, aber wurde dennoch untersucht. Diese Arbeit kann als Grundlage für weitere Forschung genutzt werden.

6.2 Ausblick

Der Sachverhalt der Coverage-Messung und der Zusammenhang zwischen Eingabe und Ausgabe müssen weiter untersucht werden. Für die Coverage muss eine maschinelle Erkennung der False Positives entwickelt werden. Des Weiteren muss danach die Ausgabe analysiert werden, um die nächste Eingabe dementsprechend verändern zu können. Sobald dieser Punkt erreicht ist, könnte der Algorithmus so weiterentwickelt werden, dass er für alle Arten von Klassifizierung funktioniert. Das ist darauf bezogen, dass beispielsweise Gesichtserkennung nicht mit einer Klassifizierung von 0 bis 9 arbeitet und somit mit dem momentanen Algorithmus nicht getestet werden kann. Falls möglich wäre der letzte Schritt, die Erkenntnisse des Coverage Guided Fuzzings in Blackbox Fuzzing zu überführen und ohne die interne Struktur des Netzes zu kennen einen effizienten Algorithmus zu finden, der False Positives gezielt erzeugen kann.

Literatur

[1] D. Geng und R. Veerapaneni. (2019). „Tricking Neural Networks: Create your own Adversarial Examples.“ [Abgerufen am 06.08.21], Adresse: https://medium.com/@ml.at.berkeley/tricking-neural-networks-create-your-own-adversarial-examples-a61eb7620fd8.

[2] B. Z. H. Zhao, H. J. Asghar und M. A. Kaafar. (2020). „On the Resilience of Biometric Authentication Systems against Random Inputs.“ [Abgerufen am 06.08.21], Adresse: https://arxiv.org/pdf/2001.04056.pdf.

[3] A. Odena, C. Olsson, D. G. Andersen und I. Goodfellow. (2019). „TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing.“ [Abgerufen am 06.08.21], Adresse: https://arxiv.org/pdf/1807.10875.pdf.

[4] B. Marr. (2020). „What are artificial neural networks - a simple explanation for absolutely anyone.“ [Abgerufen am 06.08.21], Adresse: https://bernardmarr.com/default.asp?contentID=1568.

[5] S. Albawi, T. A. Mohammed und S. Al-Zawi. (2017). „Understanding of a Convolutional Neural Network.“ [Abgerufen am 23.08.21], Adresse: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8308186.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville und Y. Bengio. (2020). „Generative Adversarial Networks.“ [Abgerufen am 23.08.21], Adresse: https://dl.acm.org/doi/pdf/10.1145/3422622.

[7] G. LLC. (2021). „Deep Convolutional Generative Adversarial Network.“ [Abgerufen am 22.08.21], Adresse: https://www.tensorflow.org/tutorials/generative/dcgan.

[8] B. Miller. (1988). „Operating System Utility Program Reliability - The Fuzz Generator.“ [Abgerufen am 23.08.21], Adresse: http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

[9] M. Woo, S. K. Cha, S. Gottlieb und D. Brumley. (2013). „Scheduling Black-box Mutational Fuzzing.“ [Abgerufen am 23.08.21], Adresse: https://dl.acm.org/doi/pdf/10.1145/2508859.2516736.

[10] A. Fioraldi und L. P. Pileggi. (2021). „FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques.“ [Abgerufen am 23.08.21], Adresse: https://arxiv.org/pdf/2102.02527.pdf.

[11] G. B. Team. (2021). „TensorFlow.“ [Abgerufen am 23.08.21], Adresse: https://www.tensorflow.org/.

[12] Z. You, J. Y. amd Kunming Li, Z. Xu und P. Wang. (2018). „Adversarial Noise Layer: Regularize Neural Network By Adding Noise.“ [Abgerufen am 06.08.21], Adresse: https://arxiv.org/pdf/1805.08000.pdf.

[13] P. Jupyter. (2021). „Project Jupyter.“ [Abgerufen am 23.08.21], Adresse: https://jupyter.org/.

[14] L. Tonin. (2020). „Mnist extended: a dataset for semantic segmentation and object detection.“ [Abgerufen am 06.08.21], Adresse: https://awaywithideas.com/mnist-extended-a-dataset-for-semantic-segmentation-and-object-detection/.

[15] R. D. Singh. (2021). „Handwritten Digit Detector.“ [Abgerufen am 22.08.21], Adresse: https://github.com/dhanpalrajpurohit/handwritten-digit-detector/.

[16] M. L. Blatchford, C. M. Mannaerts und Y. Zeng. (2021). „Determining representative sample size for validation of continuous, large continental remote sensing data.“ [Abgerufen am 22.08.21], Adresse: https://www.sciencedirect.com/science/article/pii/S0303243420308783#sec0045.

[17] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill und R. Ashmore. (2019). „Testing Deep Neural Networks.“ [Abgerufen am 22.08.21], Adresse: https://arxiv.org/pdf/1803.04792.pdf.