

Erstellung von Irrbildern zur Überlistung einer Verkehrsschilder erkennenden KI

IT-Projekt Bericht

Studiengang *Informatik*

Technische Hochschule Georg Simon Ohm

von

Leonhard Applis, Peter Bauer, Andreas Porada und Florian Stöckl

Abgabedatum: 15.01.2019

Eidesstattliche Erklärung

Wir versichern hiermit, dass der IT-Projekt Bericht mit dem Thema

Erstellung von Irrbildern zur Überlistung einer Verkehrsschilder erkennenden KI
selbständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Nürnberg, den 2. Januar 2019

LEONHARD APPLIS, PETER BAUER, ANDREAS PORADA UND FLORIAN STÖCKL

Abstract

To be done

title: Fooling an TrafficSign-AI

authors: Leonhard Applis, Peter Bauer, Andreas Porada und Florian Stöckl

Kurzfassung

To be done

Titel: Erstellung von Irrbildern zur Überlistung einer Verkehrsschilder
erkennenden KI

Autoren: Leonhard Applis, Peter Bauer, Andreas Porada und Florian Stöckl

Inhaltsverzeichnis

| | |
|---|-----------|
| Abbildungsverzeichnis | VI |
| 1 Einleitung | 1 |
| 1.1 Motivation | 1 |
| 1.2 Ziel der Arbeit | 1 |
| 1.3 Aufbau der Arbeit | 2 |
| 1.4 Verwandte Werke und Primärquellen | 3 |
| 1.5 Rahmenbedingungen des Informaticups | 3 |
| 2 Analyse der Webschnittstelle | 4 |
| 2.1 Ursprungsdaten | 4 |
| 2.2 Modellschätzungen | 4 |
| 3 Degeneration | 5 |
| 3.1 Konzept | 5 |
| 3.2 Implementierung Remote | 8 |
| 3.3 Ergebnisse Remote | 10 |
| 3.4 Implementierung Lokal | 14 |
| 3.4.1 Batch-Degeneration | 16 |
| 3.4.2 Parallel-Degeneration | 17 |
| 3.4.3 Tree-Degeneration | 18 |
| 4 Saliency Maps ANPE | 19 |
| 5 Fazit | 20 |
| 5.1 Zusammenfassung | 20 |
| 5.2 Weiterführende Arbeiten | 20 |

Abbildungsverzeichnis

| | | |
|-----|-----------------------------------|----|
| 3.1 | Degeneration Tiefe 600 | 10 |
| 3.2 | Degeneration Tiefe 4000 | 11 |
| 3.3 | Plot Degeneration | 11 |
| 3.4 | Degeneration overfit | 12 |
| 3.5 | Batch-Degeneration-Plot | 16 |

1 Einleitung

1.1 Motivation

Optische
Täuschungen,
Irrbilder,
Rahmenbedingungen

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, Methoden und Herangehensweisen vorzustellen, um die Aufgabenstellung des Informaticup 2019 zu erfüllen:

Hyperlink!

Hierbei soll ein neuronales Netz, welches sich hinter einer Webschnittstelle verbirgt und Verkehrsschilder erkennt, erfolgreich *überlistet* werden - es sollen absichtlich Bilder erzeugt werden, welche für den Menschen keine Verkehrsschilder sind aber mit einer Konfidenz von über 90% als solche erkannt werden.

Die gefundenen Methoden sollen reproduzierbar sein und in einem Maße flexibel, um beliebig viele Irrbilder zu erzeugen.

Der erweiterte Rahmen dieser Arbeit umfasst eine Dokumentation der Methoden sowie Verbesserungen und Schlussfolgerungen aus den Implementierungen zu ziehen.

Ebenfalls geliefert werden alle Elemente, um die erzielten Ergebnisse zu reproduzieren und variieren.

Nicht Ziel dieser Arbeit ist, einen Überblick über neuronale Netze, künstliche Intelligenz oder Bildbearbeitung zu vermitteln.

Ebenfalls außerhalb dieser Arbeit liegt eine Auswertung, welche Bilder von einem Menschen als Verkehrsschilder erkannt werden. Die Aussagen über solche stützen sich ausschließlich auf die persönliche Einschätzung des Projektteams.

1.3 Aufbau der Arbeit

Innerhalb dieser Arbeit werden zunächst in Kapitel 2 Informationen über die Webschnittstelle gesammelt und aufbereitet.

Im Abschnitt 2.1 werden hierfür zunächst die *German-Trafficsign-Recognition-Benchmark* (Kurz: GTSRB) vorgestellt, welche für das Training der Webschnittstelle verwendet wurden. Dieses Datenset bildet ebenfalls einen zentralen Ausgangspunkt für einige der verfolgten Ansätze.

Anschließend werden in Abschnitt 2.2 die Eigenschaften des Modells zusammengefasst. Diese bestehen zum einen aus den offiziellen Angaben der Gesellschaft der Informatiker, zum anderen aus gewonnenen Erkenntnissen. Dieses Kapitel bildet die Grundlage, um die Schnittstelle einzuschätzen.

Anschließend werden verschiedene Lösungsansätze vorgestellt, beginnend mit der *Degeneration* in Kapitel 3.

Dieser Ansatz verändert iterativ ein Verkehrsschild, und behält die Änderungen bei, sollte der erzielte Score im akzeptablen Bereich liegen. Mit passender Bildveränderungen erzielen höhere Iterationen unkenntliche Ergebnisse.

Innerhalb des Kapitels wird zunächst im Abschnitt 3.1 die Idee anhand von Pseudocode weiter erläutert und anschließend in Abschnitt 3.2 die Implementierung für die Webschnittstelle gezeigt. Die Ergebnisse liegen gesondert im Abschnitt 3.3 vor.

Neben der Implementierung für die Webschnittstelle werden zum Abschluss des Kapitels in Abschnitt 3.4 noch weitere Verbesserungen für eine lokale Implementierung vorgestellt, welche allerdings nicht für die Webschnittstelle tauglich waren.

ANPE

ANPE

Gradient
Ascent
Fooling
Aufbau!

Saliency Map
Aufbau!

Abschluss dieser Arbeit bildet im Kapitel 5 ein Fazit über die gefundenen Methoden, sowie ein Ausblick auf weiterführende Arbeiten.

1.4 Verwandte Werke und Primärquellen

1.5 Rahmenbedingungen des Informatiscups

2 Analyse der Webschnittstelle

In das Kapitel kommen die Dinge die wir über die Trasi-AI wissen

2.1 Ursprungsdaten

Hier gehen wir kurz auf die Trainingsdaten ein die wir haben, zeigen ein paar Bilder und wie fürchterlich hässlich die sind in 64x64

2.2 Modellschätzungen

Hier kommen unsere Erfahrungen, die wir mit dem Modell gemacht haben

1. Gekürzte Klassen: Aus unsererer MongoDB können wir ziemlich sicher sagen, dass wir nur 33 Klassen von Trasi haben, keine 43. Wir können nachsehen welche fehlen
2. Softmax-Ausgabefunktion
3. Interpolationsfunktion (vllt mit einem Bild in 3 Interpolationsversionen und jeweiligen Score)
4. Overfitting bei Trainingsdaten
5. unzuverlässigkeit bei nicht-Schildern (z.B. OhmLogo)

3 Degeneration

Innerhalb dieses Kapitels wird der Ansatz der *Degeneration* vorgestellt.

Die Benennung schöpft sich aus der Nähe zu genetischen Algorithmen, allerdings aus einer invertierten Perspektive: Um auf unbekannte Modelle einzugehen, wird hierbei von einem korrekt erkannten Bild *weggearbeitet*.

genauer, was das ist?

Zunächst wird das Konzept anhand von Pseudocode genauer erläutert. Anschließend wird die Implementierung des Algorithmus für die Verwendung der unbekannten Trasi-AI vorgestellt, und den Abschluss dieses Kapitels bildet eine lokale Implementierung zuzüglich einiger Verbesserungen, welche sich aufgrund der Limitierungen des Zugriffs auf die *remote-AI* nicht angeboten haben.

3.1 Konzept

Die Grundlegende Idee des Algorithmus bezieht sich darauf, ein Urbild i zu einem Abbild \hat{i} zu manipulieren, welches von dem unbekannten Klassifizierungsalgorithmus weiterhin korrekt erkannt wird.

Abhängig von der Stärke der Manipulation soll eine *Tiefe* gewählt werden, ab welcher der Algorithmus beendet wird. Als Beispiele der Manipulation seien insbesondere Rauschen und Glätten genannt, allerdings auch Kantenschärfung und Veränderungen der Helligkeit und anderer Metaparameter.

Mit fortschreitender Tiefe wird nahezu jedes Bild unkenntlich. Zusätzlich sollten allerdings weitere Parameter als Abbruchkriterien aufgenommen werden, konkret eine Anzahl an Gesamt-Iterationen und ein Abbruch, sollten keine weiteren Fortschritte erreicht werden.

Pseudocode

Folgende Parameter erwartet unsere (generische) Implementierung des Degeneration-Algorithmus:

- Einen Eingabewert i
- Eine Manipulations-Funktion $a : i \rightarrow \hat{i}$
- Eine Klassifizierungsfunktion $p : i \rightarrow \mathbb{R}$
- Eine gewünschte Tiefe d (empfohlen, nicht notwendig)
- Eine Iterationszahl its (empfohlen, nicht notwendig)
- Ein Schwellwert t , um wie viel % die Vorhersage schlechter sein darf, als das vorhergegangene Bild

Auf einige der Punkte wird in den Anmerkungen gesondert eingegangen.

```
input :  $i, a, p, d, its, t$ 
output:  $\hat{i}, \text{score}$ 
 $depth \leftarrow 0, loop \leftarrow 0$ ;
 $s \leftarrow p(i)$ ;
 $ii \leftarrow i, is \leftarrow s$ ;
while  $depth < d \parallel loop < its$  do
     $ai \leftarrow a(i)$ ;
     $as \leftarrow p(ai)$ ;
    if  $as \geq is - t$  then
         $is \leftarrow as$ ;
         $ii \leftarrow ai$ ;
         $depth++$ ;
    end
     $loop++$ ;
end
return  $ii, is$ ;
```

Algorithm 1: Degeneration

Anmerkungen

Die Manipulationsfunktionen müssen genau ein Bild der Größe (x,y) erhalten und genau ein Bild der Größe (x,y) wiedergeben, und (für die generischen Implementierungen) keine weiteren Parameter erhalten.

Zusätzlich sollte die Manipulationsfunktion zufällige Elemente erhalten. Sollte eine einfache, idempotente Glättungsfunktion den Schwellwert nicht erfüllen, so wird niemals eine größere Tiefe erreicht.

Tiefe, Schwellwert und Manipulationsfunktion müssen aufeinander abgestimmt werden. Es gibt einige Funktionen, welche eine starke Veränderung hervorrufen, und für welche eine geringe Tiefe bereits ausreicht. Auf der anderen Seite dieses Spektrums können Funktionen, welche lediglich minimale Änderungen vornehmen, schnell große Tiefen erreichen, ohne ein merklich verändertes Bild hervorgerufen zu haben.

Diese Parameter auszubalancieren obliegt dem Nutzer.

Bei der Auswahl der Parameter sollte zusätzlich überschlagen werden, wie groß die letztendliche Konfidenz ist, falls die maximale Tiefe erreicht wird.

Innerhalb der Implementierungen sollte zusätzlich eine *verbose*-Funktion eingebaut werden. Hiermit kann zum einen ein ergebnisloser Versuch frühzeitig erkannt werden, und zusätzlich, ob der Algorithmus sich festgefahren hat. Üblicherweise kann man erkennen, wenn die Manipulationsfunktion *zu stark* ist (bzw. der Schwellwert zu niedrig gewählt wurde).

Der oben genannte Algorithmus lässt sich auch für Text- oder Sprach-basierte Klassifikationen adaptieren.

Hierfür müssen lediglich andere Manipulations- und Klassifizierungs-Funktionen gewählt werden.

3.2 Implementierung Remote

Im Rahmen des Wettbewerbs wurde mit einer Rest-API gearbeitet, welche besondere Herausforderungen mit sich bringt:

Marker nach
oben

- Anfragen können fehlschlagen
- zwischen Anfragen muss ein Timeout liegen
- Mehrere Nutzer, welche die API beanspruchen, blockieren sich

Zusätzlich wurde der Grundalgorithmus um die *Verbose*-Funktion und eine History erweitert. Mithilfe der *History* können anschließend hilfreiche Plots erstellt werden.

Diese wurden für den untenstehenden Code weggelassen.

Ebenso ist anzumerken, dass ignoriert wurde, welche Klasse zuerst erzeugt wurde. Solange irgendeine Klasse mit einer passenden Konfidenz gefunden wurde, wird dies als hinreichend erachtet. Im Normalfall bleibt es allerdings bei derselben Klasse.

Die Klassifizierungsfunktion wird innerhalb der Remote-Degeneration durch einige Hilfsfunktionen umgesetzt. Diese bereiten ein als *Bytearray* vorliegendes Bild entsprechend auf und senden es an das Trasi-Webinterface (Die Methode *Scorer.Send_ppm_image(img)*) und gibt ein JSON-Array der Scores wieder.

Die Hilfsmethode *Scorer.get_best_score(response)* gibt den höchsten gefunden Score wieder.

```

# Parameters :
#   An image (as 64x64x3 Uint8 Array),
#   a function to alter the image ,
#   a threshold how much the image can be worse by every step
#   The # of Iterations i want to (successfully) alter my image
#   The # of loops which i want to do max
def remoteDegenerate(image, alternationfn = _noise, decay = 0.01, iterations = 1000, maxloops = 1000):
    # First: Check if the credentials are correct and the image is detected
    initialResp = Scorer.send_ppm_image(image)
    if(initialResp.status_code!=200):
        return
    totalLoops = 0 # Counts all loops
    depth = 0 # Counts successfull loops
    lastImage = image
    lastScore = Scorer.get_best_score(initialResp.text)
    # To check if we put garbage in
    print("StartConfidence:",lastScore)
    #We stop if we either reach our depth , or we exceed the maxloops
    while(depth<iterations and totalLoops<maxloops):
        totalLoops+=1
        # Alter the last image and score it
        degenerated = alternationfn(lastImage.copy())
        degeneratedResp = Scorer.send_ppm_image(degenerated)
        if (degeneratedResp.status_code==200):
            degeneratedScore= Scorer.get_best_score(degeneratedResp.text)
        else:
            print("Error, status code was: ", degeneratedResp.status_code)
        # If our score is acceptable (better than the set decay) we keep it
        if(degeneratedScore>=lastScore-decay):
            lastImage=degenerated
            lastScore=degeneratedScore
            depth+=1
        #We are working remote , we need to take a short break
        time.sleep(1.1)
    return lastScore,lastImage

```


3.3 Ergebnisse Remote

In diesem Abschnitt werden die mit der Degeneration erzielten Ergebnisse in Bezug auf die Trasi-Schnittstelle vorgestellt. Zunächst werden einige positive Beispiele (Erfolge) gezeigt, anschließend einige Probleme die aufgetreten sind und zuletzt noch ein kurzes Zwischenfazit gezogen.

Positive Ergebnisse

Die zuverlässigsten Ergebnisse wurden mit einfachem Rauschen erzeugt. Die Abbildung 3.1 zeigt, dass zunächst vorallem die Pixel außerhalb des eigentlichen Schildes verändert werden - ein erwartetes Verhalten. Die farbstarken bunten Pixel sind hierbei entstanden, da Werte welche die gültige Reichweite $[0,255]$ verlassen, wieder zyklisch zurück in den Farbbereich geholt werden. Sollte ein Farbwert durch das Rauschen einen Wert -2 erreichen, wird er auf 253 gesetzt.

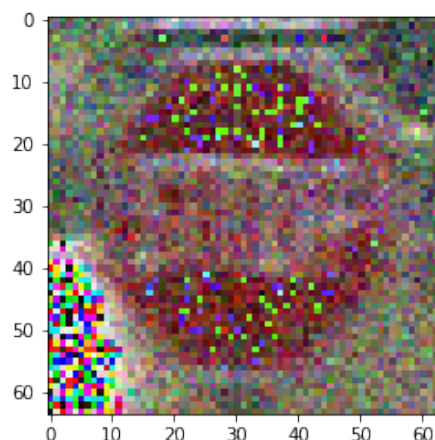


Abbildung 3.1: Rausch - Degeneration mit 600 Iterationen

Während die Abbildung 3.1 noch als Verkehrsschild zu erkennen ist, führt ein längeres ausführen der Degeneration zu einem Ergebniss wie in Abbildung 3.2. Um dieses Ergebnis zu erzielen wurden 4400 sekunden benötigt, also ca. 73 Minuten.

Die Plots in Abbildung 3.3 stellen den Verlauf des Algorithmus dar: Das erste Diagramm zeigt einen Verlauf der aktuellen *Tiefe* über die Iterationen dar, der

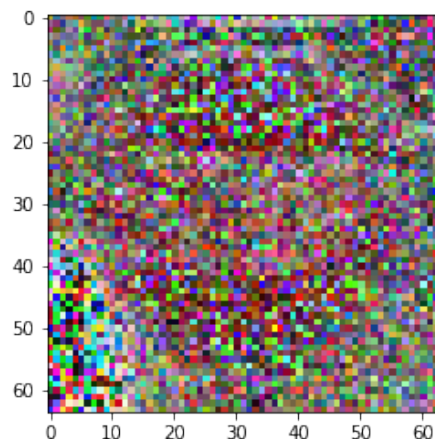


Abbildung 3.2: Rausch - Degeneration mit 4000 Iterationen

zweite die jeweils produzierten Genauigkeit der jeweiligen Iteration (nicht nur die der akzeptierten), und der letzte Plot visualisiert diejenigen Iterationen, an welchen eine Änderung stattgefunden hat (weißer Strich) oder keine (schwarzer Strich). Innerhalb der Implementierung werden ebenfalls standardmäßig diese Plots erzeugt.

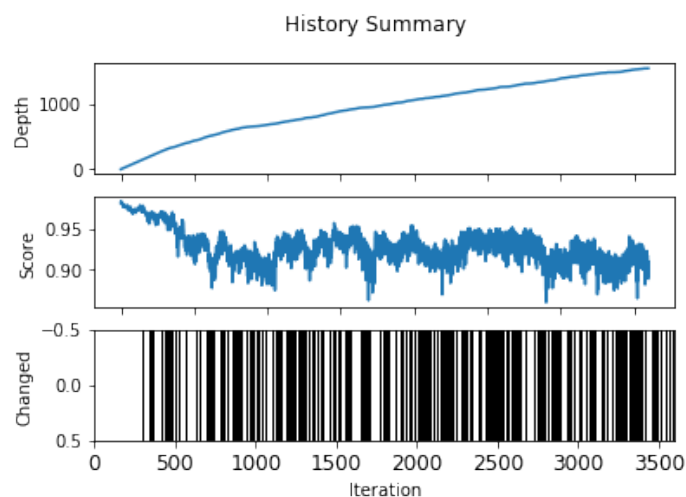


Abbildung 3.3: Plot der Rausch-Degeneration

Es wurden ebenfalls einige sehr positive Ergebnisse mit einer Mischung aus starkem Rauschen und Glätten erzeugt, allerdings waren diese nicht zuverlässig reproduzierbar.

Negative Ergebnisse

Es gab zwei primäre Fehlerquellen in Bezug auf die Remote-Degeneration: Die Auswahl von Bildern, welche im GTSRB-Training-Set waren, sowie die Auswahl ungeeigneter Manipulationsfunktionen.

Die AI innerhalb der Schnittstelle scheint sich die Bilder aus dem Trainingsset *gemerkt* zu haben. Bereits minimale, unwichtige Änderungen des Schildes (z.B. einfügen einiger blauer Punkte im Hintergrund) führen zu einer drastischen Verschlechterung des Ergebnisses. Dieses starke *Ausschlagen* des Scores machte die Benutzung der Degeneration unbrauchbar.

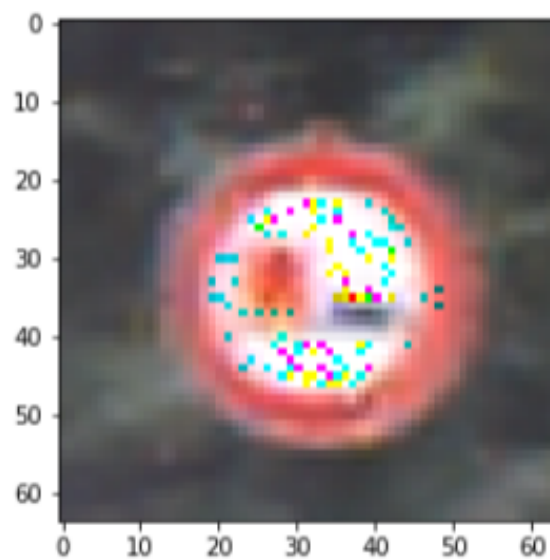


Abbildung 3.4: Rausch-Degeneration auf Trainingsbild - 36000 Iterationen

Dieses Problem hat sich herauskristallisiert, als über einen längeren Zeitraum (10 Stunden) kein Bild erzeugt wurde, welches auch nur leicht verändert wurde. Die einzigen Änderungen, welche erzielt wurden, waren innerhalb des weißen Bereiches des Überholverbotsschildes wie in Abbildung 3.4. Es wurden aber ebenfalls keine Änderungen vorgenommen außerhalb des Schildes, wo diese zu erwarten wären und bei vorhergehenden Versuchen auch zu beobachten waren (vgl. Abbildung 3.1).

Dieses Problem tritt ausschließlich (allerdings zuverlässig) bei der Verwendung von Bildern aus dem Trainingsset auf. Es tritt nicht auf sobald man Bilder aus dem Test-Set verwendet oder *GTSRB-fremde* Bilder (vorausgesetzt, sie besitzen eine

akzeptable Startkonfidenz).

Innerhalb der lokalen Implementierung ist dieses Problem ebenfalls aufgetreten, konnte allerdings behoben werden, sobald man das Overfitting erkannt hatte.

Fazit

Innerhalb dieses kurzen Zwischenfazits sollen noch einmal die Vor- und Nachteile der Degeneration zusammengefasst werden:

| Vorteile | Nachteile |
|---|---|
| Model-Agnostic: Der Algorithmus funktioniert unabhängig und ohne Wissen über das zugrundeliegende Modell | Zeitintensiv: v.A. die Remote-Variante benötigt größere Zeitspannen |
| Kontext-Unabhängig: Die Herangehensweise ist nicht auf Bilderkennungen limitiert | Vorwissen benötigt: Der Sinn des zugrundeliegenden Modells muss bekannt sein, und ein geeignetes Startbild muss ausgewählt werden |
| Erweiterbar: Die Manipulationsfunktionen können weiter ausgebaut werden und haben noch großes Potenzial | Die Degeneration erzielt bei empfindlichen Modellen schlechter Ergebnisse - gerade <i>professionelle</i> Modelle sollten lange brauchen, um so überlistet zu werden |
| Einfach: Der Algorithmus ist einfach implementiert und erläutert, er benötigt keine höhere Mathematik oder Vorwissen zur Thematik <i>Machine Learning</i> und der Modelle/Verfahren im Speziellen | Im Remote-Umfeld kann die Degeneration als DDoS wahrgenommen werden und entsprechend frühzeitig unterbunden werden. |

Caption und Label

Als besonderen Fall sind solche Modelle zu nennen, die mit jeder Anfrage *hinzulernen*:

Diese sind entweder besonders anfällig gegenüber der Degeneration, weil sie die bereits veränderten Bilder als *korrekte* klassifizieren und somit den Entstehungsprozess der Degeneration verinnerlichen, oder sie *härten* sich mit jedem Versuch gegen die neuen Änderungen und sind de facto immun gegen diesen Angriff.

Solche Modelle sind selten im Einsatz, weil zuviel Schabernack damit getrieben werden kann. Hierfür brauche ich noch eine Quelle

3.4 Implementierung Lokal

Anpassungen und Verbesserungen

Innerhalb dieses Abschnittes werden zunächst die Änderungen bei der lokalen Verwendung des Algorithmus kurz behandelt, und anschließend zwei konzeptionelle Verbesserungen vorgestellt: Parallel- und Batch-Varianten des Algorithmus.

Auf weitere Code-Beispiele wird im Rahmen des Umfangs verzichtet - sie befinden sich im Anhang.

Anpassungen

Für die lokale Implementierung wurde zunächst ein eigenes Modell (von Grund auf) trainiert mithilfe der GTSRB-Daten. Das *Scoring* der Remote-Implementierung wird durch die *predict()*-Funktion des Models ersetzt.

Als zusätzliche Erweiterung wurde für die lokale Implementierung umgesetzt, dass sich der Nutzer für eine bestimmte Klasse entscheiden kann, auf welche die Degeneration ausgelegt ist. Es wird also zuverlässig bspw. ein Stoppschild erzeugt, und kein beliebiges Schild mit hohem Score.

Des Weiteren entfällt die Wartezeit, welche für die Schnittstelle benötigt wurde, sowie die Wartezeit. Letzteres erhöht die Geschwindigkeit des Algorithmus maßgeblich.

Eine zusätzliche, passive Verbesserung wurde erzielt, indem die Verwendung der *GPU-Acceleration* von Tensorflow eingebunden wurde. Diese beschleunigte nicht nur das Training des lokalen Models maßgeblich, sondern auch die Vorhersagen, insbesondere für die Batch-Variante, wurden um ein vielfaches (\approx Faktor 20) schneller.

Quelle? Oder Ausarbeiten?

Fazit

Das wichtigste Fazit, welches im Umgang mit der lokalen Implementierung gezogen

werden konnte, ist die nicht-verwendbarkeit der lokalen Bilder für die Schnittstelle. Während dies ursprünglich die Motivation war, schnell lokal Irrbilder zu erzeugen und Remote zu verwenden, stellte sich heraus das die lokalen Irrbilder keine guten Scores an der Schnittstelle erzielten und vice versa.

Es ist anzunehmen, das die Modelle dieselben stärken haben (Verkehrsschilder korrekt zu erkennen), allerdings unterschiedliche *Schwächen*. Die erzeugten Irrbilder scheinen im Model selbst zu fußen und sind somit hochgradig spezifisch.

Die meisten stark veränderten Bilder, welche i.A. nicht mehr vom Menschen als Verkehrsschilder erkannt werden, erzeugen bei der jeweilig erstellen AI Werte >90%, und bei der anderen Implementierung zuverlässig einen Score von $\tilde{30}\%$.

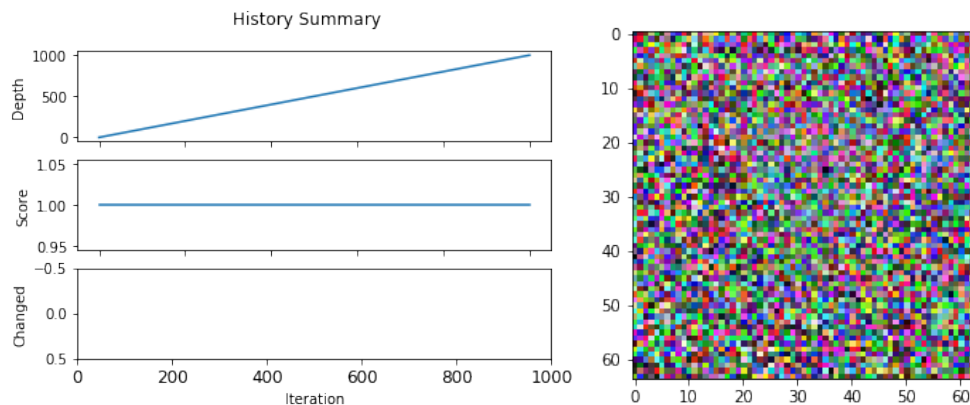
Für ein Bild, welches an sich nichts mehr mit einem Verkehrsschild zu tun hat, sind dies immernoch unwahrscheinlich hohe Werte, und die Zuverlässigkeit mit der dieser Zusammenhang auftritt, lässt einen leichten, inhaltlichen Zusammenhang der Bilder erahnen.

Beispielbild,
Remote 90
% und Lokal
30% oder vice
versa

3.4.1 Batch-Degeneration

Innerhalb des Batch-Variante wird anstatt eines einzelnen Bildes ein Array aus n veränderten Bildern erzeugt.

Diese werden alle bewertet und falls das beste Bild des Batches den Threshold erfüllt wird mit dem besten Bild weitergearbeitet.



Nähe zu
genetischen
Algorithmen
herausbringen,
Quelle!

Abbildung 3.5: Verlauf der Batch-Degeneration mit Batchsize=10 und Tiefe=1000

Abbildung 3.5 zeigt den deutlich besseren Verlauf der Batch-Degeneration gegenüber der unveränderten Implementierung in Abbildung 3.3. Auffallend ist die durchgängige Veränderung und die gleichbleibend hohe Konfidenz von mehr als 99.9% ¹.

Dieses Verhalten für Remote einzusetzen ist möglich, allerdings wurde aufgrund der Wartezeit zwischen den Anfragen davon abgesehen.

Diese Variante profitiert maßgeblich von der *GPU-Acceleration* innerhalb Tensorflows.

Selbst ohne die Verwendung des CUDA-Frameworks ist ein Tensorflow-Model auf Batch-Verarbeitung ausgelegt.

Die optimale Batchgröße zu finden ist Systemabhängig und sollte kurz getestet werden. Insgesamt benötigt die Batch-Degeneration trotzdem maßgeblich mehr Zeit: Für das Beispiel in Abbildung 3.5 wurden ca. 15 Minuten benötigt, was knapp 5 mal so lange ist wie die ursprüngliche Implementierung.

¹Innerhalb des Plots wird es auf 1 gerundet

Anmerkung: Ein prinzipielles Problem der Batch-Degeneration liegt in der Zufälligkeit der Manipulationsfunktion. Als Beispiel sei einfaches Rauschen gewählt worden.

Ein naheliegendes Verhalten für den Algorithmus ist, von den 100 erzeugten Bildern dieses auszuwählen, welches das geringste Rauschen aufweist, und als solches am wenigsten verändert wurde. Im Normalfall weist das am wenigsten veränderte Bild den nächsten Score auf.

Glücklicherweise ist dies ein hypothetisches Problem, und in der tatsächlichen Implementierung nicht aufgetreten. Dennoch sollte es vor Allem für die Manipulationsfunktion berücksichtigt werden. Im Falle einer Manipulationsfunktion, welche konstante Elemente beinhaltet (zum Beispiel Glätten oder statische Veränderungen der Helligkeit) fördert die Batch-Degeneration den selektiven Ansatz.

3.4.2 Parallel-Degeneration

Die Parallel-Variante stützt sich auf die Idee, mehrere Threads zu starten, welche gleichzeitig eine Degeneration durchführen.

Sobald ein einzelner Thread die gewünschte Tiefe erreicht hat, wird der Prozess beendet.

Die Implementierung der Parallel-Degeneration ist aufgrund mehrerer technischer Gründe gescheitert:

- **Modelgröße:** Jeder Thread braucht ein eigenes Model, welches allerdings zu groß war. Naive Benutzung eines gemeinsamen Models führen zu Race-Conditions, *geschickte* Benutzung des Models führen zu einem Verhalten wie innerhalb der Batch-Variante
- **Numpy-Arrays:** Die Bilder für die lokale Degeneration lagen als Numpy-Arrays vor, welche ein besonderes Verhalten und eine besondere Benutzung innerhalb der Parallelverarbeitung benötigen ².
- **Grafikkarteneinbindung:** Sobald die GPU-Acceleration innerhalb Tensorflows eingerichtet ist, werden (nahezu alle) Anfragen an die Grafikkarte weitergeleitet. Diese unterstützt das parallele Verhalten der einzelnen Threads nicht.

²Dieses Problem ist sicherlich lösbar - allerdings tief verankert im Bereich der Parallelverarbeitung und somit nicht im Scope dieser Arbeit

Die Probleme sind Hardware- oder Frameworkbezogen. Je nach Umfeld können diese somit entfallen. Race-Conditions entfallen beispielsweise, sollte man in der Cloud arbeiten.

Diese Variante war für die Remote-Implementierung nicht umsetzbar, da gleichzeitige Anfragen (mit dem selben API-Key) fehlschlagen. Ein internes Scheduling der Anfragen führt nicht zu schnelleren Ergebnissen.

3.4.3 Tree-Degeneration

Diese Variante führt eine Merkstruktur ein, welche die bisherigen Ergebnisse und Schritte zwischenspeichert.

Das bisherige Verhalten entspricht dem einer Liste, bei welcher lediglich der letzte Knoten verwendet wird. Mit dem jeweils letzten Bild wird weitergearbeitet, bis entweder ein neues korrektes Bild erzeugt wurde, oder der Algorithmus endet.

Schaubild:
Liste vs. Tree

Die Variation beinhaltet das Führen eines *Retry-Counters* welcher bei jedem Versuch von einem Knoten erhöht wird. Sollte eine gewisse Anzahl an Versuchen ergebnislos bleiben, wird der aktuelle Knoten verworfen und der Vorgänger benutzt.

Dieses Verhalten führt, je nach gewählter Maximalanzahl Kinder eines Knotens, zu einem (Binär-)Baum. Das Abbruchkriterium der Tiefe kann weiterhin beibehalten werden und entspricht der Tiefe des Baumes. Im Falle eines Versuch- oder Zeitbedingten Abbruchs wird das Bild mit der bisher größten Tiefe ausgegeben.

Die Entwicklung dieser Variante entstand durch Beobachtung, dass die Geschwindigkeit der Degeneration stark abhängig sind vom Ausgangsbild. Es kann ein Bild erreicht werden, welches sehr *sensibel* wahrgenommen wird und deutlich schwerer Änderungen *toleriert*.

Die Batch-Degeneration ist mit dieser Variante frei kombinierbar.

4 Saliency Maps ANPE

Hier kommt Andreas und Peters Teil.

Das File zu Evolutionären Algorithmen ist noch vorhanden, nur nicht eingebunden.

5 Fazit

5.1 Zusammenfassung

5.2 Weiterführende Arbeiten

Literaturverzeichnis