Rapid Frontend Prototyping with Deep Learning

Simon Deussen
simon.deussen@gmail.com

1. November 2018

Die Arbeit handelt von der Generierung von HTML/CSS Code für Websites aus Screenshots. Aufbauend auf dem Pix2Code Paper [5], ist diese Arbeit eine erweiterte Implementierung mit komplexeren Websites. Beiträge dieser Arbeit sind eine neue, verbesserte Netzwerk Architektur und ein neues Datenset aus Screenshots mit dazugehörigen Code.

Inhaltsverzeichnis

1. Einleitung

Worin besteht der Nutzen automatisierter Code-Erstellung? Besonders im Bereich der Frontend-Entwicklung, bei der ein Team aus Leuten mit unterschiedlichen Fertigkeiten benötigt wird, kann es bei der Entwicklung zu Bottlenecks kommen. In der klassischen Entwicklung sieht diese Zusammenarbeit folgendermaßen aus:
Ein Designer macht einen grafischen Entwurf, dieser wird vom Kunden abgenommen, dann geht er zu einem Entwickler, der nun zuallererst Markup für den Content und anschließend das Design und die richtige Darstellung nachbauen muss. Für jede grafische Veränderung muss dieser Prozess wieder ausgeführt werden. Für die meisten Entwickler, ist die Markup und CSS-Erstellung der widrigste Part der ganzen Arbeit, da er recht zeit-aufwendig, repetitiv und langweilig ist. Hier kommt es schließlich zu den Bottlenecks und Unzufriedenheit bei der Arbeit. Deswegen kann besonders an der Schnittstelle zwischen Designern und Entwicklern mit Automation viel gewonnen werden. Es gab bisher viele Ansätze diese Arbeit zu automatisieren, zum Beispiel durch Tools in dem man gleichzeitig Designen und den Markup exportieren kann. Leider sind diese Tools jedoch nicht besonders gut darin, sowohl die Designs zu erstellen, als auch den Markup zu exportieren

Eine Abhilfe soll diese Arbeit liefern: Sie ermöglicht, dass der Designer mit seinem bevorzugten Tools das Design entwickelt und der Entwickler mit einem Mausklick das fertige Markup bekommt. So kann sich der Entwickler vollends auf die Realisierung des Verhaltens und der Logik der Anwendung konzentrieren.

Die große Frage ist wie weit dieser Ansatz gehen kann? Diese Arbeit, wird versuchen die bestehende Architektur mit eigenen Daten zu nutzen, um anschließend die Komplexität der Websiten, die erstellt werden können, zu erhöhen.

Überblick über die Arbeit

Die ersten beiden Kapitel dieser Arbeit stellen die Einleitung dar und umreisen die benutzen Technologien. Sie sollen den Leser durch Beschreibungen und Verweise auf weitere Ressourcen auf den Rest dieser Arbeit vorbereiten.

Anschließend erfolgt im dritten Kapitel ein Überblick über die Funktionsweise des Modells sowie eine detaillierte Beschreibung. In den nächsten beiden Kapiteln geht es im Detail um die Synthese des neuen Datensets sowie der Validierung der Experimente.

Im sechstem Kapitel werden die Experimente beschrieben, die nötig gewesen sind, das Modell zu trainieren und die Architektur zu verbessern. Schließlich endet die Arbeit mit einer Zusammenfassung der Experimente sowie einem Fazit.

1.1. Ähnliche Arbeiten

Diese Arbeit basiert auf dem Pix2Code Paper von Tony Beltramelli [5]. Er war der Erste, der Code anhand von visuellem Input mit neuronalen Netzwerken generieren konnte. Anderen Ansätze wie DeepCoder [4] benötigen komplizierte DSL als Input und schränken so die Benutzbarkeit stark ein. Visuelle Versuche mit Android-GUIs von Nguyen [17] benötigen ebenfalls unpraktische von Experten erstellte Heuristiken. Pix2Code ist das erste Paper, das einen allgemeinen Input hat, in Form einfacher Screenshots und diesen in drei verschiedene Targetsprachen übersetzten kann. Zum einen kann es HTMl/CSS Code erstellen, zum anderen aber auch Android- und iOS-Markup. Siehe Original Code auf Github [3]

1.2. Motivation

Computer- generierte Programme werden die Zukunft der Software Entwicklung sein und diesen Bereich auch grundsätzlich verändern. Schon jetzt im Bereich der Cloud mit Serverless Computing und Lambdas geht es oftmals nur noch darum, bestehende Software-Teile zu verbinden und zu konfigurieren. Diesen Trend, der eine Reduktion des Schreibens erreichen will, und dazu stark in die Richtung des Konfigurierens geht, sehe ich in Zukunft noch viel umfassender und allgegenwärtiger. Es wird so weit gehen, dass man - nicht nur wie hier in dieser Arbeit - das Markup generiert, sondern dass aus einer Skizze sofort eine fertige Website gebaut wird, und man mit ein paar Klicks das nötige Verhalten einfach hinzufügen kann. Dieser Trend ist nicht nur auf das Web bezogen. Ich denke das sich die Webtechnologien auch in der Desktop Umgebung durchsetzen, da sie Plattform unabhängig und sehr stark optimiert sind. Außerdem sind sie sehr einfach zu lernen und weit verbreitet. Zum Beispiel Electron [1] ermöglicht den einfachen Einsatz von Webtechnologien durch einen eingebetteten Browser in der Desktopwelt. Diese Beobachtungen motivieren mich diese Arbeit zu verfassen: Automatisierung ist unumgänglich, weshalb man diese selbst bauen und damit unzähligen Entwicklern das leben leichter machen sollte.

2. Benutzte Technologien

In dem folgenden Abschnitt werden die benutzten Technologien beschrieben. Diese Erklärungen sind recht generell und gehen zunächst nicht auf die genaue Verwendung der Technologien in dem Projekt ein, dies wird anschließend im Abschnitt 3 genauer beleuchtet.

2.1. Neuronale Netzwerke

Neuronale Netzwerke sind einfach zu benutzende Modelle, welche nicht-lineare Abhängigkeiten mit vielen latenten Variablen stochastisch abbilden können [20]. Die Stärke von neuronalen Netzen liegt darin, aus großen Mengen von Daten Gesetzmäßigkeiten oder Patterns zu erkennen.

Im einfachen Sinn, sind sie gerichtete Graphen, deren Knoten oder Nodes aus ihren Inputs Werte errechnen und diese an die folgenden Nodes weitergeben. Hierbei werden 3 verschiedene Arten von Nodes unterschieden:

Input Node Über diese Nodes bekommt das Netzwerke die Input Parameter.

Hidden Node Nodes, welche das Netzwerke-interne Modell repräsentieren.

Output Node Diese Nodes bilden die Repräsentation des Ergebnisses ab.

Nachdem die Node aus den Inputs einen Wert errechnet hat, geht dieser durch eine Aktivierungsfunktion. Diese Funktion stellt den Zusammenhang zwischen dem Input und dem Aktivitätslevel der Node her. Man unterscheidet zwischen folgenden Aktivitätsfunktionen

Lineare Aktivitätsfunktion Der einfachste Fall, linearer Zusammenhang zwischen Inputs und Output.

Lineare Aktivitätsfunktion mit Schwelle Linearer Zusammenhang ab einem Schwellwert. Sehr nützlich um Rauschen herauszufiltern. Ein häufig genutzte Abhandlung davon:

ReLU Hier werden nur positive Werte weitergeleitet: $f_x = x^+ = max(0, x)$

Binäre Schwellenfunktion Nur zwei Zustände möglich: 0 oder 1 (oder auch -1 oder 1)

Sigmoide Aktivitätsfunktion Benutzung entweder einer logistischen oder Tangens-Hyperbolicus Funktion. Diese Funktionen gehen bei großen positiven Werten gegen 1 und bei großen negativen Werten gegen 0 (logistische Funktion) oder -1 (Tangens-Hyperbolicus Funktion). Diese Funktion bietet den Vorteil das sie das Aktivitätslevel begrenzt.

Jede der Nodes hat eine bestimmte Anzahl an Verbindungen, diese hängt von der Art der Nodes und deren Zweck ab. Wichtig ist jedoch, dass jede Node mit mehreren anderen Nodes verbunden ist. Das bedeutet, den Output mehrerer Nodes als Input zu bekommen und den eigenen Output als Input für die folgenden Nodes weiterzuleiten. Die Stärke der Abhängigkeit zwischen zwei Nodes wird als Gewicht ausgedrückt. Jede Verbindung in einem neuronalen Netzwerk hat ein Gewicht, welches mit dem Output der vorangegangen Node verrechnet wird, während es als Input weiter verwendet wird.

Das Netzwerk-interne Modell wird in diesen Gewichten abgespeichert. Es repräsentiert also das Wissen, dass durch das Training entstanden ist.

Das Training eines Netzwerkes ist das schrittweise Anpassen der Gewichte, bis es ein Modell des Problems gelernt hat. Um zu wissen, wie die Gewichte angepasst werden müssen, wird eine Loss-Funktion benötigt. Diese Funktion bestimmt den Modell-Fehlers und wird während dem Training minimiert.

Zunächst wird das Training in zwei verschieden Arten unterteilt:

Supervised learning Innerhalb des Trainingsdatensets hat jeder Datensatz ein vorgegebenes Output Label. Zum Beispiel ein Bild von einem Auto ist auch so gekennzeichnet. Nun werden so lange die Gewichte des Netzwerkes optimiert, bis ein jeweiliger Input auch den richtigen Output erzeugt.

Unsupervised learning Hier hat das Trainingsdatenset keine Label. Die Gewichtsveränderungen erfolgen im Bezug zu der Ähnlichkeit von Inputs. Das soll heißen, wenn es viele verschiedene Bilder bekommt, werden Bilder mit ähnlichen Inhalten eine hohe Nähe aufweisen, ein Bild von einem PKW wird näher an dem Bild von einem LKW sein als an dem Bild von einem Apfel.

2.2. Convolutional Neural Network - CNN

CNN sind tiefe neuronale Netzwerke mit einer speziellen Architektur und spezialisiert auf die Verarbeitung von Bildern. Da man die Anwendungsdomäne eingeschränkt hat, kann man bestimmte Annahmen treffen, welche die Anzahl der Verbindungen und damit der Rechenoperationen verringern und somit das Netz effektiver machen. Um aus Bildern, Informationen zu gewinnen, müssen die Ebenen des Netzwerkes nicht vollständig verbunden sein. Stattdessen werden Filter (Convolutions) und Sub-Sampling genutzt [18]. Filter sind kleine Matrizen, die bestimmte Features entdecken, zum Beispiel Kanten mit bestimmter Ausrichtung, siehe Abbildung 2. Durch das Erlernen der Filter im Training kann das Netzwerk aus den Pixelwerten, schrittweise abstraktere Features errechnen. Diese gehen von einfachen Kanten zu komplexeren Umrissen und schließlich zu vollständigen Teil-Objekten . Zum Beispiel wird aus vielen Kanten ein Kreis, aus mehreren einfachen Objekten ein detaillierter Umriss und schließlich entsteht aus den einfachen Features eine Repräsentation eines Kopfes.

Diese geschichtete Architektur ist vom Auge und der biologischen Signalverarbeitung inspiriert [15].

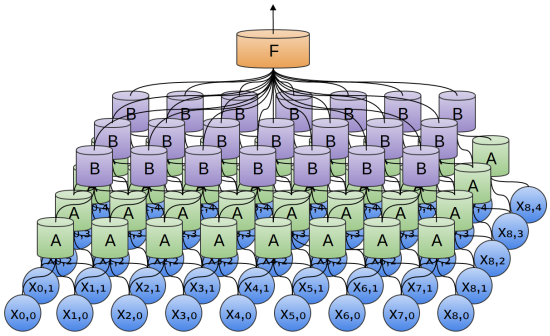

Abbildung 1: CNN von `http://colah.github.io/`

In neuronalen Netzen sind Fully-Connected-Layer die Ebenen mit den meisten Verbindungen. Bei Betrachtung der Abbildung 1 würde eine Fully-Connected-Layer insgesamt $n = 9^2 * 5^2 = 2025$ Verbindungen zwischen $x_{i,j}$ und A benötigen. Stattdessen benötigt die Convolutional-Ebene mit einem Kernel von $2 * 2$ nur $n = 8 * 5 * 4 = 128$ Verbindungen. Abgesehen von der Einsparung von Verbindungen, hat ein CNN viele verschiedene Kernel, welche sich jeweils die gelernten Gewichte teilen [18]. Für jeden Kernel A in der Abbildung werden die gleichen 4 Gewichte gebraucht. Diese Gewichte werden während des Trainings gelernt und bilden dann ein bestimmtes Feature der Input-Daten ab, wie oben beschrieben. Viele Kernel zusammen bilden ein Featureset, welches anschließend an die darüber liegende Ebene weitergegeben wird. Dies geschieht solange, bis die Feature-Extraktion abgeschlossen ist und eine letzte Ebene (zum Beispiel eine Fully-Connected) die finales Bestimmungen durchführt.

Die Operationen, die Kernel auf einen Input ausführen, sind in Abbildung 2 visualisiert. Hier wird durch eine einfache Convolution eine Repräsentation des Inputs gebildet, in der nur vertikale Kanten enthalten sind. Wenn nun ein Kernel mit einer derartigen Operation im Netzwerk aktiviert wird, teilt dieser dem Rest des Netzwerkes mit, dass hier eine vertikale Kante im Input ist.

Durch das Kopieren der Kernel und der geringeren Verbindungsanzahl sind nun sehr große und sehr tiefe Modelle möglich. 2012 haben Krizhevsky et al. die Welt der Bildklassifizierung mit ihrem Paper und den darin vorgestellten Modell revolutioniert [16]. Ihr tiefes Convolutional Neural Network konnte insgesamt 1000 Klassen unterscheiden mit einer Genauigkeit von 63%. Die Ergebnisse waren der Anfang einer neuen Art von Bildklassifizierung und es folgten viele weitere Paper, die auf ihrem Erfolg aufbauend CNNs tief in der Computer Vision verankerten [18].

	0	0	0	
	-1	1	0	
	0	0	0	

Abbildung 2: Beispiel Kernel aus der Gimp-Dokumentation [2]

2.3. Recurrent Neural Network - RNN

RNNs sind eine Erweiterung der klassischen Feedforward Netze, die imstande sind, Input-Sequenzen mit verschiedener Länge zu verarbeiten [10]. Die RNNs schaffen dies, indem sie einen inneren Zustand haben, dessen Aktivierungen von vorherigen Inputs abhängig sind. Realisiert wird das, imdem Verbindungen zu Nodes aus vorangehender Schichten eingefügt werden.

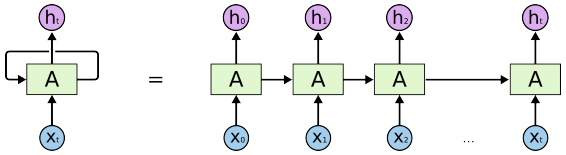

Abbildung 3: RNN von `http://colah.github.io/`

Wie in Abbildung 3 zu sehen ist, ähnelt der Verbund der rekurrenten Einheiten einer Liste. Damit ist diese Art neuronaler Netzwerke besonders gut geeignet sequentielle Daten abzuarbeiten [19]. Listen, Sprachsynthese und maschinelle Übersetzungen sind Bereiche, für die RNNs geeignet sind.

Klassische RNNs, also die reine Verkettung von einfachen rekurrenten Units, gelangen jedoch an einen Punkt, ab dem sie Sequenzen von zu großer Länge nicht mehr ausreichend gut modellieren können, da ab einer gewissen Länge, beim Training die Gradienten zu numerischer Instabilität neigen [11]. Das heißt, die Gradienten gehen gegen null oder in seltenen Fällen gegen ∞ [10], wodurch das Training scheitert. Damit scheidet das reine Verketten von RNN-Units aus, um lange Abhängigkeiten zu modellieren.

2.3.1. Long short-term Memory - LSTM

Ein LSTM ist eine bestimmte Form der RNNs. Ein LSTM ist so gebaut, dass die einzelnen Units selbst entscheiden können, welche Daten aus vorangehenden Inputs beibehalten oder vergessen werden - es hat einen eigenen Speicher, dessen Benutzung und Verwaltung von dem LSTM während dem Training gelernt wird. Durch diese dynamische Speicherverwaltung, kann es auf der einen Seite, bestimmte Informationen länger speichern, auf der anderen, kurzfristig benötigte Daten verarbeiten und dann wieder vergessen. Ein einfaches Beispiel ist ein Text-verarbeitendes LSTM. Dieses LSTM hat die Aufgabe aus Texten Informationen über Autos zu extrahieren. Sobald es in dem Text um ein anderes Auto der gleichen Marke geht, kann das LSTM, die Bezeichung des Autos, die Farbe und Zylinderanzahl des Motors vergessen, da die bei dem nächsten Auto verschieden sind. Die Marke hingegen bleibt gespeichert.

Während des Trainings eines LSTMs, erlernt dieses auch das Speichern und Löschen. Dadurch kann es sehr viel effizienter als reine RNNs Daten mit temporaler Dimension oder langen Abhängigkeiten auswerten. Fast jeder Erfolg, der mit RNNs erzielt wurde, ist auf LSTMs zurückzuführen [19].

Eingeführt wurde das LSTM von Hochreiter und Schmidhuber im Jahr 1997 [14]. Seit dem wurden viele Variationen davon getestet, verworfen und verbessert. In dieser Arbeit wird mit dem Begriff LSTM diese Original Spezifikation gemeint.

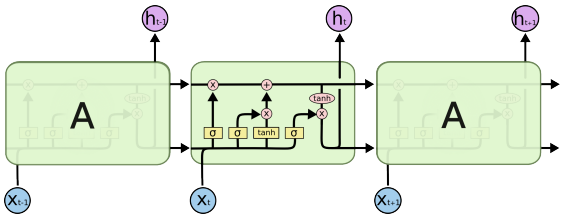

Abbildung 4: LSTM von `http://colah.github.io/`

Eine LSTM-Unit teilt sich mit allen anderen LSTM-Units des Netzwerkes einen Cell-State, in der Abbildung die obere, horzontale Linie. Jede Unit kann diesen auf zwei Arten verändern, mit einem `forget`-Gate werden Informationen vergessen, mit einem `add`-Gate Informationen hinzugefügt. Schließlich berechnet jede Unit einen eigenen Hidden-State anhand des Inputs, des geteilten Cell-States und dem Hidden-State vorangegangener Units.

2.3.2. Gated Recurrent Unit - GRU

Die GRU ist eine Abwandlung des klassischen LSTMs. Die Architektur wurde 2014 von Kyunghyun Cho et al. eingeführt [6], um eine RNN Einheit zu erhalten, welche es schafft,

selbstständig Features aus verschiedenen, weit auseinander liegenden Zeitpunkten zu extrahieren [10]. Was diese Variante ausmacht, ist eine Zussammenführung des `forget`- und `add`-Gates, sowie das Weglassen des Cell-States. Dadurch hat sie weniger Parameter als die klassische Variante. Trotz dieser geringeren Anzahl an Parametern, ist die Perfor-

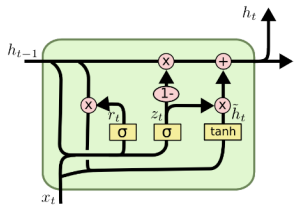

Abbildung 5: GRU von `http://colah.github.io/`

mance ähnlich der klassischen Variante[12], trainiert aber schneller, da weniger optimiert werden muss. Im LSTM erfolgt eine Kontrolle darüber, welcher Anteil des Cell-States mit in eine Berechnung einfließt. Im GRU dagegen, wird immer die volle Historie mit in die Berechnung aufgenommen [10].

2.4. Training

Als Training wird der Prozess bezeichnet, durch dem ein neuronales Netzwerk Wissen aus vorliegenden Daten extrahiert. Genau dieser Vorgang sorgt für den großen Erfolg von neuronalen Netzen. Anders als bei herkömmlichen statistischen Methoden können NNs aus riesigen Datenmengen Patterns und Sachverhältnisse lernen. Damit dies funktioniert, muss eine Kostenfunktion gebildet werden können, anhand derer bestimmt werden kann, wie weit das genutzte Modell von der optimalen Lösung entfernt ist. Durch diesem Abstand, können die Parameter des gewählten Modells angepasst werden, um dem Optimum näher zu kommen.

2.4.1. Stochastig Gradient Descent - SGD

SGD ist eine iterative Methode um ableitbare Funktionen und Modelle zu optimieren. Um ein Modell zu optimieren, wird eine Verlustfunktion benötigt. Diese zeigt den Fehler an, der entsteht wenn das Modell mit falschen Parametern Entscheidungen trifft. Die richtigen Parameter werden während des Trainings gefunden, bei Minimierung der Verlustfunktion. Bei SGD werden durch zufällige Samples des Trainingssets, Näherungen an den Gradienten gebildet, die dann wiederum genutzt werden um die Parameter zu optimieren. Nach mehreren Durchläufen durch das Trainingsset wird ein Set an Parameters gefunden, die ein Minimum der Verlustfunktion bilden [23].

RMSprob Eine Variante des SGD, die sich besonders gut eignet um RNNs zu optimieren [13]. Die Besonderheit dieser Optimierungsmethode ist, dass die Größe der Updates durch die Größe der vorangehender Gradienten skaliert wird.

2.4.2. Overfitting

Overfitting tritt ein, wenn das Modell zu komplex für das gewählte Ziel ist, und deswegen die Trainingsdaten auswendig lernt. Es kommt zu sehr guten Ergebnissen auf dem Trainingsset, sobald es aber unbekannte Daten verarbeitet, stürzt die Performance ab da es nicht verallgemeinern kann. Eine Art Overfitting zu lösen, ist es das Netzwerk zu verkleinern oder spezielle Ebenen wie Dropouts einzubauen, um dem entgegen zu wirken.

2.4.3. Underfitting

Dies ist das Gegenteil von Overfitting; das Netzwerk schafft es nicht ein Set an Parametern zu finden, um das Problem zu approximieren mit der gegeben Architektur. Da es zu einfach aufgebaut ist. Eine gängige Lösung ist es, das Netzwerk zu vergrößern damit es das komplexe Problem besser lösen kann.

2.4.4. Dropout

Dropout ist eine Technik, die genutzt wird, um Overfitting zu vermeiden. Während des Trainings werden zufällig Nodes sowie ihre Eingangs- und Ausgangsverbindungen aus dem Netzwerk entfernt [22].

2.4.5. Transferlernen

Transferlernen ist ein Prozess, bei dem ein fertiges Netzwerk, welches auf ein bestimmtes Problem trainiert ist, für ein andere Problemlösung genutzt wird. Anstatt das Netzwerk mit zufälligen Gewichten zu initialisieren, werden die bereits erlernten Gewichte weiter trainiert bis diese gelernt haben, das neue Problem zu lösen. Zum Beispiel im Bereich der Bildklassifizierung ist diese Technik sehr sinnvoll, da allgemeine Features wie verschieden ausgerichtete Kanten oder einfache geometrische Objekte universell sind und so nicht erst gelernt werden müssen. Dadurch bekommt ein Netzwerk eine gewisse Wissensbasis, die für die neue Problemdomäne als Grundbaustein dient.

2.5. Sampling

Während des Samplings, wird ein fertig trainiertes neuronales Netzwerk genutzt um, aus Daten Schlussfolgerungen abzuleiten. Im Kontext dieser Arbeit, wird probiert aus einem Screenshot, eine bedeutungsvolle DSL-Sequenz zu erstellen.

2.6. Keras

Um die in dieser Arbeit genutzten tiefen neuronale Netzwerke zu definieren, wird das Framework Keras [7] genutzt. Keras macht es sehr einfach neuronale Netzwerke zu erstellen und zu verwalten. Es selbst verwaltet nur die Definition von Modellen, die eigentlichen Berechnungen werden im Backend getätigt, von Theano, Tensorflow oder CNTK. So ermöglicht es schnelles Experimentieren mit wenig Overhead und ohne Boilerplate-Code.

2.7. Domain-specific language - DSL

Eine Programmiersprache, die auf eine einzelne Problem-Domäne spezialisiert ist, wird DSL genannt. Im Gegensatz zur DSL steht die General Purpose Language, welche eine Programmiersprache ist, die sehr breit, für viele verschiedene Anwendungen, benutzt werden kann. Die Trennung zwischen DSL und GPL ist nicht immer klar: Es gibt zum Beispiel Teile einer Sprach, die hoch spezialisiert für eine bestimmte Aufgabe sind, aber andere Teile können allgemeinere Aufgaben lösen. Auch historisch bedingt kann sich die Einordnung einer Sprache ändern. JavaScript wurde ursprünglich für ganz einfache Steuerungen von Websites eingeführt, kann aber inzwischen für alles mögliche eingesetzt werden - vom Trainieren von CNNs im Browser, zu klassischen Backend-Jobs. In dieser Arbeit, wird eine hoch spezialisierte Sprache aus Token genutzt, um die Websites für das neuronale Netzwerk zu beschreiben. Die Token-Sprache wird genutzt damit das Netzwerk einfach Vokabeln erlernen kann, wodurch die Komplexität der Sprachmodellierung, die das Netzwerk intern machen muss, stark sinkt. Die Ausgabe des Netzwerkes erfolgt ebenso in der Token-Sprache und kann anschließend in valides Markup kompiliert werden.

Hypertext Markup Language - HTML HTML ist die Standardprogrammiersprache zur Erstellung von Websites. Mit einzelnen HTML Elementen beschreibt sie den semantischen Zusammenhang von Websites.

Cascading Style Sheets - CSS CSS beschreibt die Präsentation, also das Aussehen des Content einer Markup-Language (zum Beispiel HTML). Klassische Inhalte sind Farben, Positionen und Effekte von User Interface Elementen.

3. Rapid Frontend Prototyping - Überblick

RFP ist ein Tool, welches aus einfachen Website-Screenshots HTML/CSS Markup erzeugen kann. Dafür lernt ein System aus neuronalen Netzwerken den Zusammenhang von Screenshots und deren Code.

Um den Code für die Website Screenshots zu generieren, wird ein Modell bestehend aus zwei rekurrenten Netzwerken und einem Convolutional Neural Network erstellt. Das hier genutzte Modell ist eine Abwandlung des im Original Paper verwendeten Modells. Da die neue DSL komplexer als die im Original Paper[5] ist, mussten die RNNs verändert werden.

Von einem Computer abgespeicherte Bilder sind ein denkbar schlechtes Format, um aus den Rohdaten auf deren Inhalte zu schließen. Die Pixel, die einem Bild zugrunde liegen, sind lediglich Helligkeitsinformationen, die einfach genutzt werden können, um für Menschen Bilder darzustellen, aber in roher Form eben sehr wenig Informationen enthalten, was in einem Bild dargestellt wird. Um aus diesen Helligkeitsinformationen Schlussfolgerungen zu ziehen, müssen diese in mehreren Schritten abstrahiert werden, um bedeutungsvolle Features zu extrahieren. Hierfür wird ein CNN eingesetzt. Dieses verarbeitet das gegebene Input-Bild und erstellt eine niedrig-dimensionale Repräsentation aus bedeutungsvollen Features.

Als nächstes muss das Modell die Verwendung und Bedeutung der Sprache, also der den Screenshot beschreibende Token-Sequenz, erlernen. Dafür wird ein RNN, das Sprach-Modell, genutzt, welches eine Netzwerk-interne Repräsentation der Sprache erlernt. Dem Netzwerk direkt HTML/CSS-Markup beizubringen, würde einen großen Overhead erzeugen, da pro Feature des Screenshots viele Tags, Klassen und Symbole benötigt werden, die aber auch einfach duch einen Token mit dazugehörigen Code repräsentiert werden können. So kann die Sprache einfach per One-Hot-Encoding übergeben werden und das Netzwerk kann sich auf die Zusammenhänge von High-Level Features des Screenshots und Token-Elementen konzentrieren.

Das CNN und das RNN erzeugen eine interne Repräsentation des Input-Bildes und der Input Sequenz. Nun müssen die beiden Elemente zusammengeführt und verstanden werden. Dies geschieht im zweiten RNN. Das Decoder-Modell bekommt die beiden Elemente als Input und erlernt eine Sequenz an Token zu erstellen, die das Input-Bild beschreibt. Zu sehen ist dieses Zusammenspiel in der Abbildung 6.

Die linke Seite der Abbildung 6 stellt den Zustand während des Trainings dar. Die rechte Seite während der Predictions.

1. Ein Screenshot I und die zugehörigen Token Sequenz x_t wird als Input in das Netzwerk angelegt. Der Screenshot an das CNN, die Token-Sequenz an das LSTM
2. Das LSTM erstellt eine interne Repräsentation q_t und das CNN erstelle die Repräsentation p

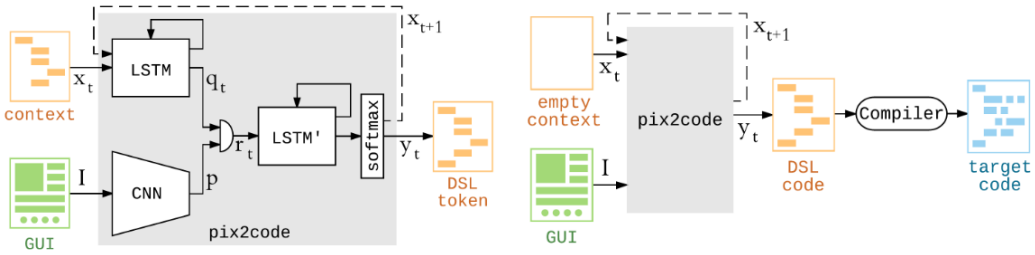

Abbildung 6: Architektur des Netzwerkes, Grafik aus dem Original Paper [5]

3. q_t und p werden zu dem Vektor r_t zusammengefügt
4. LSTM' erstellt aus dem Vektor r_t die Prediction y_t
5. y_t wird dem Input x_{t+1} angehängt.

Während einer Prediction, erstellt das Netzwerk Token für Token, bis es feststellt, dass die Prediction abgeschlossen ist. Die Details hierzu sind in Abschnitt 3.5 und Abschnitt 3.6.

3.1. Vision Modell

Das CNN ermöglicht eine aussagekräftige Repräsentation des Input-Screenshots zu erstellen. Jeder Screenshot wird durch das CNN auf einen Vektor mit 1024 Features reduziert, welcher genug High-level-Informationen enthält um das Bild zu beschreiben.

In dieser Arbeit geschieht das Lernen des CNNs unüberwacht. Das gesamte Modell wird zu einem möglichst geringen Klassifikationsfehler optimiert, im Zuge dessen erlernt das CNN die Representation des Bildinhaltes.

Die Screenshots werden wie um Original Paper auf 256 mal 256 Pixel skaliert, ohne Bewahrung der Seitenverhältnisse. Wie das VGGNet [21] nutzt das CNN Convolutions mit einem 3 mal 3 Pixel großen Kernel, welche über ein ReLU aktiviert werden. Die 3x3 Convolution Ebenen werden jeweils zweimal hintereinander genutzt. Anschließend wird eine Pooling Ebene genutzt und ein Dropout ausgeführt. Wie im Code zu sehen ist, wird die Anzahl der Features pro `Conv2D/Conv2D/MaxPooling2D/Dropout`-Modul jeweils verdoppelt.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 254, 254, 32)      896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 252, 252, 32)      9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 126, 126, 32)      0
```

```
_________________________________________________________________
dropout_1 (Dropout)          (None, 126, 126, 32)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 124, 124, 64)      18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 122, 122, 64)      36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 61, 61, 64)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 61, 61, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 59, 59, 128)       73856
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 57, 57, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 28, 28, 128)       0
_________________________________________________________________
dropout_3 (Dropout)          (None, 28, 28, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 100352)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              102761472
_________________________________________________________________
dropout_4 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              1049600
_________________________________________________________________
dropout_5 (Dropout)          (None, 1024)              0
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 48, 1024)          0
=================================================================

Trainable params: 104,098,080
```

Danach werden 2 Fully-Connected Ebenen (`Dense` im Code) genutzt, um aus den gefundenen Features eine 1024-dimensionale Repräsentation des Bildes zu finden. Anschließend wird der Output über den `RepeatVector` an das Decoder-RNN weitergegeben.

3.2. Sprach Modell

In diesem Teil des Netzwerkes wird eine interne Repräsentation der DSL erlernt. Damit das Netzwerk die DSL verwenden kann, wird diese per One-Hot-Encoding übergeben.

Weder die Repräsentation als String, noch die als One-Hot-Encoding gibt den einzelnen Token eine Bedeutung untereinander. Genau dafür wird das Sprach-Modell benötigt, denn die Netzwerk-interne Repräsentation erschafft eine detaillierte Sprache, welche ähnlich den Features aus höheren CNN Ebenen mehr Informationen über den Kontext enthält.

Realisiert wird das Sprach-Modell durch zwei Ebenen aus rekurrenten Units. Im Original Paper wurden hier LSTM Units benutzt, in der Implementierung dieser Arbeit jedoch GRU Units. Beide rekurrenten Units unterscheiden sich hauptsächlich in der internen Verwaltung vorheriger Zustände und der Modellierung von längeren Beziehungen in den Input-Daten [19].

Das Sprach Modell gibt nach der Enkodierung des Sprach-Inputs diesen an das Decoder-Modell weiter.

Die RNN des Sprach Modells geben als Output die gesamte verarbeitete Sequenz zurück. Dadurch hat dieser Output die Maße `CONTEXT_LENGTH` $*$ `ANZAHL_RNN_UNITS`.

3.3. Decoder Modell

Der Decoder-Teil generiert aus den zusammengefügten Outputs des Vision- und des Sprach-Modells die Token-Sequenz. Nachdem der Input-Screenshot sowie die dazugehörigen Tokens verarbeitet wurden, muss das Netzwerk nun den Zusammenhang zwischen diesen Daten erlernen. Übergeben werden die Daten folgendermaßen: `decoder = concatenate([encoded_image, encoded_text])`. Durch die `concatenate`-Funktion, wird jeder einzelne kodierte Sprach-Token des Input-Kontexts mit einer Repräsentation des Screenshots verknüpft.

Auch dieser Teil des Netzwerkes besteht aus rekurrenten Units. Zum Vergleich, das Decoder-Modell hat viermal so viele Einheiten wie das Sprach Modell. Auch hier wurden bessere Ergebnisse bei der Benutzung von GRUs anstelle von LSTMs erzielt. Im Gegensatz zu dem Sprach Modell, gibt dieser Teil des Netzwerkes nicht die gesamte Sequenz des Input-Kontexts zurück, sondern nur den nächsten Token.

3.4. Mathematische Beschreibung

Mathematisch gesehen lässt sich die Generierung von Tokens durch das Netzwerk folgendermaßen beschreiben:

$$\begin{aligned} encImg &= CNN(inputImg) \\ encContext &= RNN(context_t) \\ predToken &= softmax(RNN'(encImg, encContext)) \\ context_{t+1} &= context_t + predToken \end{aligned} \tag{1}$$

inputImg Screenshot der Website, $256 * 256$ Pixel

encImg Encoding des CNNs vom Input Bild, durch die abschließende `RepeatVector`-Ebene, sind die Output-Maße `CONTEXT_LENGTH` $* 1024$

$context_t$ Kontext mit Maße `CONTEXT_LENGTH` $*$ `ANZAHL_TOKEN`, Häufig auftretender Fall: $48 * 23$. One-Hot-Encoding der zuletzt generierten Token. Zu Beginn nur mit einem `start`-Token gefüllt

encContext Die Modell-interne Repräsentation des Kontextes

predToken Ausgabe des Netzwerkes. Wird für den nächsten Schritt an den Kontext hinzugefügt. Im Falle der ersten $n <$ `CONTEXT_LENGTH` $- 1$ Token, wird der Token in den Kontext-Vektor an Stelle n eingefügt. Danach wird eine neuer Kontext-Vektor gebildet, der die Token 1 bis $m =$ `CONTEXT_LENGTH` $- 2$ enthält so wie den neu generierten Token an letzter Stelle.

CNN Der Vision-Teil des Modells

RNN Das Sprach-Modell

RNN' Der Encoder-Teil des Modells

Durch die Verkettung der jeweiligen Netzwerke ist die Gesamtarchitektur nach wie vor ableitbar und damit von Anfang bis Ende optimierbar mit einem Gradienten-Abstiegsverfahren.

3.5. Training

Da es für jeden Screenshot eine zugehörige Token-Sequenz mit variabler Länge gibt, müssen die Trainingsdaten mit einem Sliding-Window-Prinzip abgearbeitet werden. Dafür werden aus der Token-Sequenz n Samples gebildet, wobei $n = len(token\_sequence) -$ `CONTEXT_LENGTH`. Das Modell bekommt pro Screenshot n Paare aus Screenshot und dazugehörigem Token-Kontext. Wichtig hierbei ist, dem Modell mitzuteilen, wann eine Sequenz beginnt und wieder endet, dafür gibt es einen `start`-Token sowie einen `end`-Token. Diese werden als pre- bzw. suffix an die Token-Sequenz angefügt, bevor die n Samples gebaut werden. Dies ist eine gängige Methode, um ein Netzwerk für die Generierung von unterschiedlich langen Sequenzen zu verwenden [11]. Zur Veranschaulichung wird ein einfaches Beispiel durchgespielt:

1. $X = (I, T)$ beschreibt den Datensatz bestehend aus Input-Bild I und dazugehöriger Token-Sequenz T.
2. Um ein erstes Trainings-Sample zu bilden, wird ein Vektor S_0 der Länge `CONTEXT_LENGTH` $=$ 6 erstellt.
3. S_0 wird nur mit dem `start`-Token befüllt. $S_0 =$ `[start, none, none, none, none, none]`

4. Für die Erstellung weiterer Trainingssample, wird T mit einem Sliding-Window abgearbeitet.

5. Zunächst wird der erste Token $t_0 = T(0) =$ `header` aus T an erster Stelle des neuen Samples S_1 gesetzt und der `start`-Token rückt an die zweite Stelle:
   $S_1 =$ `[header, start, none, none, none, none]`

6. Das nächste Sample S_2 nimmt nun einen weiteren Token $t_1 = T(1) = \{$ aus T mit auf, und verrückt die beiden schon vorhandenen, um eine Stelle weiter:
   $S_2 =$ `[{, header, start, none, none, none]`

7. Diese Schritten werden so lange wieder holt, bis alle einzelnen Token aus T aufgenommen wurden. Danach wird das letzte Sample erstellt, welches den `end`-Token enthält:
   $S_{last} =$ `[end, }, button, text, headline, {]`

8. Nun trainiert das Netzwerk mit jeden Tupel $x_n = (I, S_n)$ sequentiell.

Die Aufgabe des Netzwerkes ist es nun, aus dem Input x_n, den nächsten Token y_{n+1} zu klassifizieren. So kann während des Trainings mit Backpropagation ein klassischer Multi-Class-Loss optimiert werden.

$$L(I, X) = -\sum_{t=1}^{T} x_{t+1} log(y_t) \tag{2}$$

Hierbei ist x_{t+1} der erwartete Token und y_t der berechnete Token. Das Modell ist in einer Gesamtheit ableitbar, dies bedeutet, der CNN-Teil kann zusammen mit den beiden RNNs optimiert werden. Wie im Original Paper wurde mit RMSProp-Algorithmus trainiert mit einer Learning Rate von $0,0001$. Zur Erhöhung der numerischen Stabilität wurden die Output Gradienten in das Intervall $[-1, 1]$ getrimmt. Es wird eine Dropout-Regulation genutzt als Maßnahme gegen Overfitting. Im CNN nach den Max-Pooling Ebenen mit 25% und nach den Fully-Connected Ebenen mit 30%.

3.6. Sampling

Für das Sampling von Bildern wird ein ähnlicher Prozess wie während dem Training in Abschnitt 3.5 durchlaufen:

1. Das Test-Bild I wird mit einem Kontext Vektor C_0, der nur den `start`-Token enthält, an das Netzwerk angelegt.
   Dabei gilt: $C_0 =$ `[start, none, none, none, none, none]`

2. Mit der erste Prediction $y_0 = header$ des Netzwerkes wird genauso, wie während des Trainings, ein neuer Kontext-Vektor $C_1 =$ `[header, start, none, none, none, none]` erstellt.

3. Jede weitere Prediction des Netzwerkes wird so wieder als Input verwendet bis der Fall $y_{last} =$ `end` auftritt.

Nachdem der `end`-Token erstellt wurde, kann probiert werden, aus der Token-Sequenz eine Website zu kompilieren.

4. Daten Synthese

Da im Zuge dieser Arbeit eine Erweiterung der DSL des Original Papers implementiert wurde, ist es erforderlich, neue Trainingsdaten zu synthetisieren. Ein Beispiel der neu synthetisierten Daten ist in der Abbildung 7 zu sehen. Für die Arbeit wurden insgesamt 4 verschiedene Datensets synthetisiert, mit jeweils ca. 3000 bis 4000 Bildern.

Das `DataCreationTool` ist eine Sammlung von Python-Skripten, die nach vorgegebenen Regeln einen Token-Baum erzeugen. Jeder Token-Baum hat einen `body`-Token als Wurzel und der gesamte Inhalt liegt als dessen Kinder vor. Dafür wurde eine Helferklasse geschrieben, die ein Element des Token Baums abbildet. Diese kann zum einen Parameter wie den Token-Namen, den Inhalt und der Kinder speichern, zum anderen enthält sie Funktionen zum Konvertieren des Baumes zu einer String-Repräsentation sowie zum Rendering nach HTML/CSS.

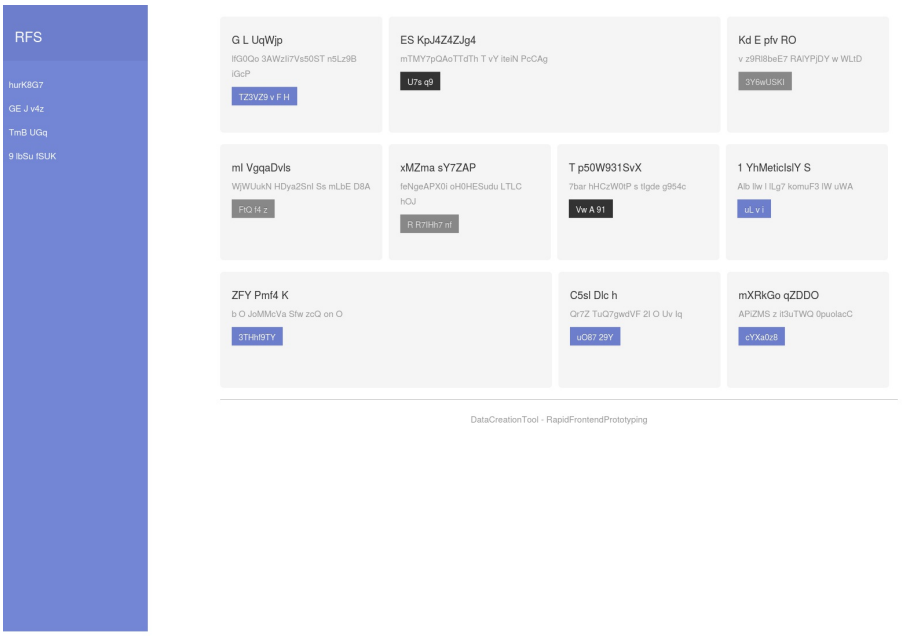

Abbildung 7: Ein Beispiel Bild aus dem Datenset

4.1. Generieren der Token-Bäume

Die Token-Bäume werden in der Datei `dateCreationTool/createAllTokens.py` generiert. Diese Datei erzeugt alle möglichen Token-Kombinationen anhand der folgenden Regeln:

4.1.1. Grammatik

$$start \rightarrow [H, C] \tag{3}$$

$$H \rightarrow [Ml|Mr|S] \tag{4}$$

$$Ml \rightarrow [logoLeft, buttonWhite|logoLeft, buttonWhite, buttonWhite|...] \tag{5}$$

$$Mr \rightarrow [buttonWhite, logoRight|buttonWhite, buttonWhite, logoRight|...] \tag{6}$$

$$S \rightarrow [sidebarHeader, sidebarItem|sidebarHeader, sidebarItem, sidebarItem|...] \tag{7}$$

$$C \rightarrow [R|R, R|R, R, R] \tag{8}$$

$$R \rightarrow [S|D, D, |Q, Q, D|Q, D, Q|D, Q, Q|Q, Q, Q, Q] \tag{9}$$

$$S, D, Q \rightarrow [smallTitle, text, contentButton] \tag{10}$$

$$contentButton \rightarrow [buttonBlue, buttonGrey, buttonBlack] \tag{11}$$

Regeln 3 - 5 sind gekürzt. Es können bis zu 5 Buttons auftreten.

4.1.2. Zeichenerklärung

H Header der Website, enthält eins der folgenden Elemente:

- **Ml** Menue mit Logo auf der linken Seite
- **Mr** Menue mit Logo auf der rechten Seite
- **S** Sidebar

C Content der Website, besteht aus ein bis drei Wiederholungen dieses Elements:

- **R** Row, die aus einem oder mehreren Row Elementen bestehen kann:
  - **S** Single Row Element - die ganze Row ist mit diesem ausgefüllt.
  - **D** Double Row Element - ist so breit wie eine Hälfte der Row
  - **Q** Quadruble Row Element - ist so breit wie ein Viertel der Row

  Jedes dieser Elemente enthält den gleichen Inhalt:
- **smallTitle** Überschrift
- **text** Text-Inhalt
- **contentButton** Ein Button der entweder Blau, Grau oder Schwarz ist

4.1.3. Token Generierung

Die Generierung erfolgt mit einfachen, verschachtelten Loops und einem Kartesischen Produkt, um alle möglichen Kombinationen abzudecken. Dadurch werden 4128 verschiedene Layout Kombinationen möglich. Die Layout-Kombinationen haben eine durchschnittliche Länge von 62.47 Tokens (Arithmetisches Mittel), bei einen Median von 65 Tokens. Außerdem ist die maximale Länge 92, die minimale Token Anzahl ist 16.

```
menu_or_sidebar = [True, False]
logo_left_or_right = [True, False]
possible_num_of_menu_button = [1, 2, 3, 4]
possible_num_of_rows = [1,2,3]
possible_row_type = [0,1,2,3,4]

row_count_layout_combinations = []

for i in possible_num_of_rows:
    row_count_layout_combinations.extend(
        list(itertools.product(possible_row_type, repeat=i))
    )

for i in range(len(row_count_layout_combinations)):
    row_count_layout_combinations[i] = list(row_count_layout_combinations[i])

complete_layouts =  []

for menu_flag in menu_or_sidebar:
    for logo_flag in logo_left_or_right:
        for num_of_menue_button in possible_num_of_menu_button:
            for row_count_layout in row_count_layout_combinations:

                    root = Element("body", "")

                    if menu_flag:
                        menu = tokenBuilder.createMenu(logo_flag, num_of_menue_button)
                        root.addChildren(menu)
                    else:
                        sidebar = tokenBuilder.createSidebar(num_of_menue_button)
                        root.addChildren(sidebar)

                    for i in range(len(row_count_layout)):
                        row = tokenBuilder.createRow(row_count_layout[i])
                        root.addChildren(row)
```

```
complete_layouts.append(root)
```

In den ersten fünf Zeilen werden die jeweiligen Konfigurations-Möglichkeiten in Listen geschrieben. Anschließend wird eine weitere Liste erstellt mit allen möglichen Kombinationen aus Anzahl von Rows und Row Types mit der Funktion `itertool.product()`. Um dies zu erreichen, hätte man auch je nach Anzahl von Rows verschachtelte `For`-Loops benutzen können. Allerdings wäre dieser Ansatz zu unflexibel, falls in Zukunft noch mehr Rows hinzukommen würden. Als letzter Schritt wird eine Loop pro Liste benutzt um über den gesamten Raum der möglichen Kombinationen zu iterieren. Dann wird mit der jeweiligen Kombination ein Token-Baum gebildet und der Liste aller Token-Bäume angehängt. Im weiteren Code Verlauf wird diese Liste auf mehrerer Threads verteilt und parallel in eigenen Files gespeichert. Bei der Speicherung wird wie im Fall des Renderings eine rekursive Funktion auf Element-Ebene ausgeführt, die den Tag-Name jedes Elements und der Kinder in einen String zusammenführt.

4.2. DSL Mapping

Zu jedem dieser Token, existiert ein Mapping nach HTML/CSS. In einer eigenen Datei `dsl-mapping.json` ist dies abgebildet:

```
"opening-tag": "{",
"closing-tag": "}",
"body": "<!DOCTYPE html>\n <head>\n <meta charset=\"utf-8\">\n ...
"header": "<nav class=\"menue\">\n    <ul class=\"nav nav-pills...
"btn-active": "<li class=\"active\"><a href=\"#\">[]</a></li>\n...
"btn-inactive-blue": "<button type=\"button\" class=\"btn btn-p...
"btn-inactive-black": "<button type=\"button\" class=\"btn btn-...
"btn-inactive-white": "<button type=\"button\" class=\"btn btn-...
"btn-inactive-grey": "<button type=\"button\" class=\"btn btn-p...
"row": "<div class=\"container\"><div class=\"row\">{}</div></d...
"single": "<div class=\"col-lg-12\">\n{}\n</div>\n"
"double": "<div class=\"col-lg-6\">\n{}\n</div>\n",
"quadruple": "<div class=\"col-lg-3\">\n{}\n</div>\n",
"big-title": "<h2>[]</h2>",
"small-title": "<h4>[]</h4>",
"text": "<p>[]</p>\n",
"logo-left": "<a class=\"logo-left\">RFP</a>\n",
"logo-right": "<a class=\"logo-right\">RFP</a>\n",
"sidebar": "<div class=\"wrapper\">\n    <div id=\"sidebar\">\...
"sidebar-element": "<li><a href=\"#\">[]</a><li>"
```

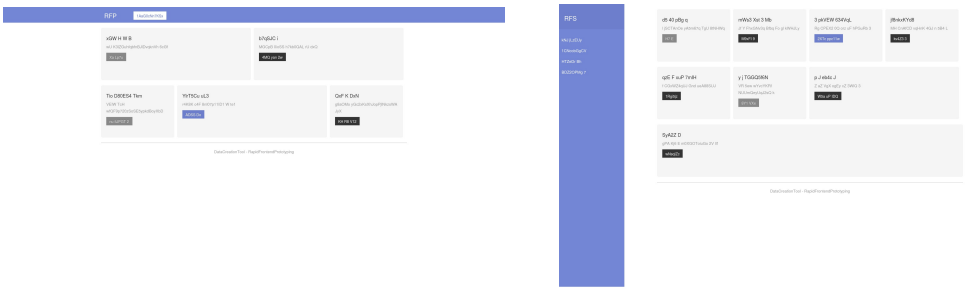

Abbildung 8: Beispiel der beiden Websiten Typen

Um somit aus einem Token-Baum HTML/CSS Markup zu erzeugen, startet der Wurzelknoten eine rekursive Rendering-Funktion. Diese nutzt dem zu den Knoten gehörigen Code und traversiert den gesamten Baum. Jeder Mapping String eines Tokens enthält einen Platzhalter, hier `{}`, mit dem signalisiert wird, wo der Code des Kinderknoten hingehört. Ähnlich gibt es ebenso einen Platzhalter für Text-Content, nämlich die Zeichen: `[]`. Für den Text-Content werden im Zuge der Arbeit nur zufällige Zeichenfolgen genommen, damit das neuronale Netzwerk lernt nicht auf den Text zu achten.

Weitere Punkte im Zusammenhang mit der Erstellung der Trainingsdaten sind:

Screenshot Erstellung Nachdem der HTML/CSS String als Datei abgespeichert wurde, kann mit dem Tool `imgkit` ein `.png` oder `.jpg` erzeugt werden.

Teilen der Trainingsdaten Die Gesamten Trainingsdaten werden in einem Trainingsset mit einem Anteil von 70%, einem Testset mit 20% und einem Validierungsset mit 10% der Bilder abgespeichert.

4.3. Erweiterung der Sprache

Der große Unterschied zu dem Original Paper besteht darin, dass es in dem hier erstellten Datenset, zwei unterschiedliche Website-Typen gibt. Zum einen, die Website mit `header`-Element, zum anderen die Website mit der `sidebar`. Das Ziel der Netzwerke ist nun, nicht nur die eine Website zu lernen, sondern die unterschiedlichen Typen zu erfassen und den Content richtig einzufügen. Diese beiden Typen sind in Abbildung 8 zu sehen.

5. Validierung der Ergebnisse

Um die Qualität der Ergebnisse zu vergleichen, wurde eine Metrik erstellt, welche anhand der Wichtigkeit der jeweiligen Elemente einen Score bildet. Auf Token-Ebene einfach nach Fehlern zu suchen und jeden falschen Token gleich zu gewichten, würde zu nicht aussage-kräftigen Scores führen. Zum Beispiel ist ein Menü-Item zu viel oder zu wenig sehr viel weniger schlimm als eine falsche Kategorisierung des Headers der Website. Um eine ständig gleich bleibende Bewertung der trainierten Netzwerke zu schaffen, ist ein Jupyter Notebook `dataCreationTool/testScoring.ipynb` implementiert worden. Dieses analysiert die generierten Token-Sequenzen der getesteten Netzwerke. Dazu werden pro Datensatz mehrere Tests gemacht. Der erste Test überprüft ob der Header korrekt ist. Dieser ist einer der am stärksten gewichteten Tests. Danach wird festgestellt, ob die Anzahl der Menü-Item korrekt ist. Anschließend wird der Rest des Contents geprüft und zwar die Anzahl an Rows, der richtige Row-Type pro Row und die Anzahl der korrekten sowie falschen Buttons. Pro Datensatz wird so ein `error_object` erstellt. Hier ein Beispiel:

```
'countCorrectButtons': 1,
'countCorrectRowType': 0,
'countWrongButtons': 5,
'countWrongRowType': 3,
'differenceButtonCount': 3,
'differenceMenuButtons': 0,
'differenceRowCount': 0,
'differenceTokenCount': 16,
'isHeaderCorrect': True,
'predictedFileName': 'pred.gui',
'trueFileName': 'true.gui',
'trueHeaderType': 'sidebar',
'true_token_count': 55
```

Hier hat das Modell insgesamt 16 Tokens zu viel generiert, diese kommen aus drei zu viel generierten Buttons, sowie den falschen Row Types. Jede generierte Row hat hier den falschen Row Type, zu sehen an `'countCorrectRowType': 0`. Richtig generiert wurde der Header Type, hier wurde die korrekte `sidebar` verwendet.

Aus diesen Metriken wird nun folgendermaßen ein Score berechnet:

$$score = cWB + dBC + dMB + 5 * cWRT + 5 * dRC + iHC * 10 \tag{12}$$

cWB `countWrongButtons`: Die Anzahl Buttons mit falscher Farbe

dBC `differenceButtonCount`: Der Unterschied in der Anzahl an Buttons

dMB `differenceMenuButtons`: Unterschied der erkannten Menü-Buttons

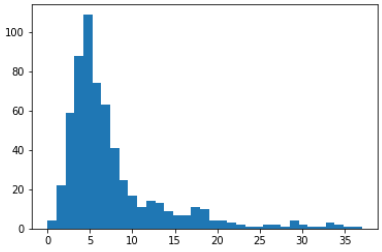

Abbildung 9: 1. Histogramm

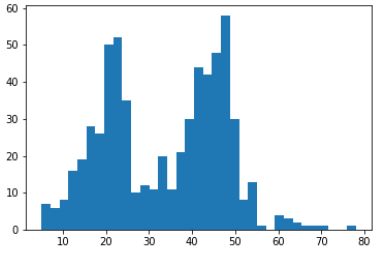

Abbildung 10: 2. Histogramm

cWRT `countWrongRowType`: Gibt an, wie viele falsche Row-Types gefunden wurden

dRC `differenceRowCount`: Ist die Differenz der Anzahl der Rows

iHC `isHeaderCorrect`: Gibt an, ob der Header korrekt ist. Kann entweder den Wert 0 oder 1 annehmen.

Nachdem für jedes Testbild so ein Score berechnet wurde, kann ausgehend von diesem die Performance des Modells genauer bestimmt werden. Um die Verteilung der Scores pro Modell besser verstehen zu können, wird die Verteilung der Fehler anhand mehrerer Lageparameter beschrieben. Dazugehören verschiedene Durchschnitte und Quintile:

```
'median': 6,
'mean': 7.47172859450727,
'g_mean': 6.192757554950546,
'h_mean': 5.272351955758953,
'quintiles':
    'p20': 4,
    'p40': 5,
    'p60': 7,
    'p80': 9
'most_common_error_count': 5,
'count_most_common_error_count': 110
'count_no_erros': 1,
```

Ein weiteres wichtiges Werkzeug hierbei ist ein Histogramm über die Verteilung der Fehlerscores. Bei Vergleich von zwei Histogrammen, Abbildung 9 und Abbildung 10, erkennt man, dass bei Abbildung 9 die Anzahl an niedrigen Scores sehr viel höher ist, als bei Abbildung 10.

Außerdem findet ein Vergleich der Fehler-Typ-Verteilungen aller Predictions gegenüber den 20% schlechtesten Predictions statt.

6. Experimente

In diesem Abschnitt wird beschrieben, welche Trainingsversuche durchgeführt wurden um ein bestmögliches Ergebnis zu erhalten. Die Beschreibungen für jeden Trainingsversuch sind jeweils in fünf Abschnitte unterteilt. Dazu gehört eine Einführung, das Datenset, eine Auflistung der veränderten Parameter, die Ergebnisse und ein Fazit. Die finale Zusammenfassung und Auswertung ist im Kapitel 7 zu finden.

Architektur

Bevor die Beschreibung der jeweiligen Experimente beginnt, hier eine Auflistung der Standard Parameter der Modelle.

```
Layer (type)                 Output Shape              Param #     Connected to

==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 20)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 48, 128)           209920      input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 48, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
lstm_3 (LSTM)                (None, 48, 512)           3409920     concatenate_1[0][0]
----------------------------------------------------------------------------------
lstm_4 (LSTM)                (None, 512)               2099200     lstm_3[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 20)                12312       lstm_4[0][0]
```

Wichtig hierbei sind folgende drei Ebenen des Modells; `sequential_1`, `sequential_2` und `concatenate_1`.

`sequential_1` Dieser Teil ist ein CNN, welches den Screenshot der Website als Input hat.

`sequential_2` Das Sprach-Modell, ein rekurrentes Netzwerk, dass die DSL erlernt.

`concatenate_1` Hier wird der Output des CNN und des Sprach-Modells zusammen geführt und in ein weiteres rekurrentes Netzwerk gegeben welches aus der Sprache und dem Screenshot die Beschreibung dazu findet.

Weitere Hyper-Parameter:

`CONTEXT_LENGTH` Die `CONTEXT_LENGTH` ist auf 48 Tokens eingestellt. Die `CONTEXT_LENGTH` bestimmt die Größe des Sliding-Windows, mit dem die Input Token Sequenzen abgearbeitet werden.

`IMAGE_SIZE` Die Größe der Input Bilder ist 256 mal 256 Pixel.

`BATCH_SIZE` Trainiert wird in Batches mit jeweils 64 Bildern

`EPOCHS` Es werden 10 Epochen trainiert.

`STEPS_PER_EPOCH` In jeder Epoche weren 72.000 Schritte gemacht.

Insgesamt hat das Netzwerk 109.829.432 trainierbare Parameter und kommt damit auf eine Größe von knapp 420 Mega Byte.

6.1. 1. Trainingsversuch

Einführung

Um eine Baseline aller weiteren Ergebnisse zu erstellen, wird das Original-Modell aus dem pix2code Paper mit meinen neuen Daten von Grunde auf neu trainiert.

Datenset

Das verwendete Datenset enthält 4128 Bilder, von denen 2890 für das Training verwendet werden. Die - das Datenset beschreibende - DSL enthält 24 Token. Jedes Bild wird mit durchschnittlich 62,47 Token (Median: 65) beschrieben, die maximale Anzahl liegt bei 92 Token, die minimale Anzahl bei 16.

Veränderte Parameter

Für diese Baseline wurden alle Parameter unverändert übernommen. Die einzige Anpassung, die gemacht werden muss, sind zwei Ebenen im Netzwerk, dessen Input und Output die Anzahl der Token reflektieren. Dies sind Ebene `input_2` und `dense_3`. Der Shape der beiden Ebenen wurde angepasst, um mit 24 Token arbeiten zu können. Daher ist dieser in der Input-Ebene `(None, 48, 24)` anstatt `(None, 48, 20)`. Ebenso bei der Dense-Ebene `(None, 24)` anstatt `(None, 20)`.

Ergebnis

Nach dem Training des Netzwerkes wurden mehrere zufällige Bilder ($n = 10$) aus dem Test Set ausgewertet. Bei jedem einzelnen der Bilder kommt dasselbe Ergebnis:

```
body {
sidebar {
sidebar-element,sidebar-element,sidebar-element,sidebar-element
},
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
},
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
},
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
}
```

Fazit

Anhand des Ergebnisses kann man sagen, dass entweder die erhöhte Token Anzahl oder die neuen Trainingsbilder zu viel Entropie in die Trainingsdaten bringen. So kann das Modell leider zu keinem sinnvollen Optimum konvergieren.

6.2. 2. Trainingsversuch

Einführung

In diesem Versuch wurde eine vergrößerte Context-Länge erprobt. Da im ersten Versuch das Training nicht funktioniert hat, vermute ich den Fehler in einem zu kleinen Sichtbereich des LSTMs. Da durch die Token-Länge die Sichtbarkeit der Long-Term-Dependencies geregelt wird, verdoppele ich diese, um zu sehen, ob dadurch ein verbessertes Ergebnis zustande kommt.

Datenset

Gleichbleibend zum ersten Versuch.

Veränderte Parameter

Das Modell ist genau gleich wie im ersten Versuch, lediglich die Context-Länge wurde von 48 auf 96 erhöht.

Ergebnis

Wie beim ersten Versuch ergibt dieses Training wieder den gleichen Output, unabhängig vom als Input gewählten Bild. Um sicher zu gehen, dass diese Prediction nicht zufällig von dem gewählten Testbild abhängt, habe ich insgesamt 313 Bilder getestet, um festzustellen, das alle diese Bilder zum gleichen Ergebnis führen. Alle dieser Bilder bekommen den exakt gleichen Output.

Fazit

An einem zu geringen Kontext kann das nicht konvergierende Training nicht liegen. In den folgenden Versuchen werden andere Parameter verändert.

6.3. 3. Trainingsversuch

Einführung

Da im ersten und zweiten Versuch, mit der Original Architektur kein brauchbares Ergebnis herausgekommen ist, wird nun versucht mit einer höheren Komplexität der LSTMs ein Modell zu finden das besser funktioniert.

Datenset

Gleichbleibend zum ersten Versuch.

Veränderte Parameter

Hier habe ich die Anzahl der LSTM `units` von 128 im Language Modell auf 192 erhöht. Außerdem erfolgte eine Veränderung der LSTM `units` von 512 auf 768 im Decoder Modell.

```
__________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
input_1 (InputLayer)            (None, 256, 256, 3)  0
__________________________________________________________________________________
input_2 (InputLayer)            (None, 48, 24)       0
__________________________________________________________________________________
sequential_3 (Sequential)       (None, 48, 1024)     104098080   input_1[0][0]
__________________________________________________________________________________
sequential_4 (Sequential)       (None, 48, 192)      462336      input_2[0][0]
__________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 48, 1216)     0           sequential_3[1][0]
                                                                 sequential_4[1][0]
__________________________________________________________________________________
lstm_3 (LSTM)                   (None, 48, 768)      6097920     concatenate_1[0][0]
__________________________________________________________________________________
lstm_4 (LSTM)                   (None, 768)          4721664     lstm_3[0][0]
__________________________________________________________________________________
dense_3 (Dense)                 (None, 24)           18456       lstm_4[0][0]
==================================================================================
Trainable params: 115,398,456
```

Durch diese Veränderung ergeben sich ca. 8 Millionen mehr Parameter. Die Länge des Kontextes wurde zurück auf 48 gestellt.

Ergebnis

Exakt gleiches Ergebnis wie in den beiden ersten Versuchen.

Fazit

Da inzwischen immer noch der gleiche Output generiert wird, suche ich nun nach einen Fehler im Programm-Code. Folgende Ursachen könnte es geben:

Fehler im Sampler Der Sampler generiert nach dem Training aus Screenshots die Token-Sequenz. Hier könnte zum einen, ein falsches Modell geladen werden, oder es wird fehlerhaft geladen.

- **Falsches Modell** Durch Umbenennung des Modells im Output Folder und Eingabe eines falschen Modellnamens, bricht das Programm jeweils ab.
- **Fehlerhaftes Laden** Die Ausgabe von `model.summary()` nach dem Laden des Modells ist identisch mit der erstellten während des Trainings.

Fehler in den Daten Während des Preprocessings werden die Trainingsbilder in `numpy`-Arrays mit `shape(256,256,3)` umgewandelt. Anschließend werden die Arrays und `gui`-Files zusammen gespeichert.

- **Array Konvertierung** Nach optischer Kontrolle von zufälligen Samples ($n = 10$) musste ich feststellen, dass diese korrekt sind.
- **Zuordnung Files** Nach Kontrolle zufälliger Samples sieht die Zuordnung ebenfalls gut aus.

Fehler in Vocabulary Es könnte auch an der fehlerhaften Abspeicherung der einzelnen Tokens liegen. Vocabulary und One-Hot-Encoding sind ebenfalls richtig.

Fehler im Datenset / -synthese Hier wurde ein Fehler gefunden: Alle Screenshots, welche die `sidebar` anstatt des `menue` haben, sind doppelt enthalten. Der Grund hierfür ist die Positionierung des Menü-Logos. Da es im Falle des `menue` zwei Möglichkeiten für die Positionierung des Logos gibt, wurden diese durchiteriert in der Erstellung der Daten. Da es bei der `sidebar` diese beiden Möglichkeiten aber nicht gibt, wurden alle Bilder mit der `sidebar` doppelt erstellt. Ein Viertel der Trainingsdaten hat ein Menü mit Logo links, ein weiteres Viertel hat das Logo rechts. Die andere Hälfte der Trainingsdaten hat die `sidebar`, bei denen aber jede mögliche Konfiguration doppelt enthalten ist.

6.4. 4. Trainingsversuch

Einführung

Nach dem Finden des Fehlers im vorherigen Versuchsaufbau, wurde ein neues Datenset erstellt.

Datenset

Ein neues Datenset mit 3096 Bildern wurde erstellt. Ein Drittel hat das Feature `menue_logo_left`, ein Drittel `menue_logo_right` und der Rest die `sidebar`.

Veränderte Parameter

In diesem Versuch wurden die gleichen Parameter wie im ersten Versuch genutzt.

Ergebnis

Es erfolgte keine Verbesserung der Trainingsergebnisse. Die Token-Sequenz hat sich dahingehend geändert, dass nun anstatt der `sidebar` ein Menü in allen Ergebnissen ist.

```
body{
header{
logo-right,btn-inactive-white,btn-inactive-white
}
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
},
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
```

```
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
},
row {
quadruple {
small-title,text,btn-inactive-blue
},
quadruple {
small-title,text,btn-inactive-blue
},
double {
small-title,text,btn-inactive-blue
}
}
```

Fazit

Die gefundene Token Sequenz muss wohl die allgemeingültigste sein, da diese immer wieder gefunden wird. Da der Fehler nicht im Programm liegt, muss er entweder in den Daten oder im Modell liegen.

6.5. 5. Trainingsversuch

Einführung

Gleichzeitig zu dem 4. Versuch wurde noch ein Subset der Daten erstellt, das nur 70% der Bilder aufweist. Dieses hat 2166 Bilder im Vergleich zu dem Set vom 4. Trainingsversuch. Der Gedanke hierbei ist, dass im Original Training vom pix2code Paper 1500 Trainingsbilder ausgereicht haben. Unter Umständen konvergiert es mit weniger Daten zu einem anderen Durchschnittswert oder fängt an, die richtigen Ergebnisse herauszufinden.

Datenset

Eine reduzierte Variante des Sets aus dem vierten Versuch.

Veränderte Parameter

Gleiche Parameter wie im ersten Versuch.

Ergebnis

Es erfolgte keine Verbesserung der Trainingsergebnisse.

Fazit

Dieser Ansatz hat gezeigt, dass dieser Weg der falsche ist.

6.6. 6. Trainingsversuch

Einführung

Nach Analyse der Daten des Original Papers, wurde festgestellt, dass dort ein Tag weggelassen wurde. Es wird kein `body`-Tag verwendet. Dieser ist bei jedem der Bildbeschreibenden Token-Sequenzen der root-Knoten. Da er jedes Mal auftritt, hat er aber keine Relevanz zu dem Screenshot und kann daher weggelassen werden.

Datenset

Neues Datenset, ohne `body`-Tags. Dieses hat insgesamt 3096 Bilder und ist wieder 70-20-10% gesplittet in Trainings-, Test- und Validierungsset. Durch das Fehlen des `body`-Tags ergeben sich folgende Verteilungsmaße:

```
Token length analysis:
     mean token length:    59.69
     max token length:     89
     min token length:     14
     median token length:  62
```

Die Median-Token-Länge ist um genau 3 Token geringer geworden. Dies liegt daran, dass zugehörig zum `body`-Tag zwei Klammern benötigt wurden (`{` und `}`), welche anzeigen, dass der Rest der Tokens ausschließlich Kinds-Knoten des `body`-Tags sind.

Veränderte Parameter

Gleiche Parameter wie im ersten Versuch.

Ergebnis

Es kommt die gleiche Ergebnis-Sequenz wie bei den Versuchen davor heraus.

```
header{
logo-right,btn-inactive-white,btn-inactive-white
}
row{
quadruple{
small-title,text,btn-inactive-black
}
quadruple{
small-title,text,btn-inactive-black
}
double{
small-title,text,btn-inactive-black
}
}
row{
quadruple{
small-title,text,btn-inactive-black
}
quadruple{
small-title,text,btn-inactive-black
}
double{
small-title,text,btn-inactive-black
}
}
```

Diesmal jedoch ohne `body`-Tag.

Fazit

Das Scheitern des Trainings wurde nicht durch die Verkleinerung der DSL behoben. Als nächster Schritt sollte jeweils der Sprach- sowie der Vision-Teil des Modells näher in Betracht gezogen werden. Ein Großteil der Parameter des Modells ist bereits im CNN (hier erfolgt die Analyse des Bildes), deswegen wird es die Gesamtperformance weniger stark verbessern, wenn man die gleiche Zahl an neuen Parametern hier hinzufügt. Aus diesem Grund wird zunächst der Sprach- und der Decoder-Teil des Modell untersucht.

6.7. 7. Trainingsversuch

Einführung

Wie im Fazit des vorangegangen Versuchs erwähnt, wird nun daran gearbeitet, die LSTMs, welche der sprach analysierende und der decodierende Teil des Modells sind zu verbessern. Nach der Lektüre eines Posts von Colahs Blog [19], werden testweise die klassischen LSTMs durch GRUs ersetzt.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 48, 128)           157056      input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 48, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
gru_3 (GRU)                  (None, 48, 512)           2557440     concatenate_1[0][0]
----------------------------------------------------------------------------------
gru_4 (GRU)                  (None, 512)               1574400     gru_3[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 23)                11799       gru_4[0][0]
==================================================================================

Trainable params: 108,398,775
```

Die LSTMs in `sequential_2` sowie in Ebene 3 und 4 wurden durch GRUs ersetzt. Die Anzahl der `units` wurde beibehalten (128 bzw. 512). Dadurch, dass im GRU die Input- und Forget-Gates gekoppelt sind, hat das Modell jetzt eine Millionen Parameter weniger.

Arithmetisches Mittel 7.82

Geometrisches Mittel 6.36

Harmonisches Mittel 5.34

Quintile p20: 4, p40: 5, p60: 7, p80: 10

Median 6

Modus 5

Vorkommen Modus 109

Gesamt Fehler Score 4840

Anzahl Token Testset 36751

Anzahl generierter Token 36186

Score pro Token in Testset 0.1317

Abbildung 11: Lageparameter

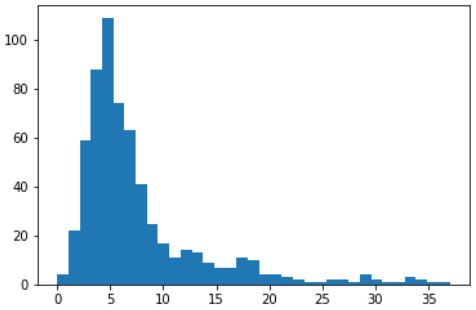

Abbildung 12: Fehlerverteilung

Ergebnis

Anstatt der bereits mehrmals auftretenden Token Abfolge gelingt es diesem Modell richtige Predictions zu erstellen. Alle der 619 getesteten Files konnten ohne Fehler compiliert werden. Die Analyse der Fehlerverteilung ergibt folgende Lageparameter in Abbildung 11. Die Verteilung der Fehleranzahlen ist in Abbildung 12 gegeben.

Im Weiteren wird das letzte Quintil der Fehler analysiert und mit dem Rest verglichen. In der Abbildung 13 ist die Fehlerverteilung aller Testdaten zu sehen, in Abbildung 14 die besten 80% und in Abbildung 15 die schlechtesten 20%.

Bei 80% aller Testdaten sind die Fehlerursachen zu 92% ausschließlich falsche Buttons! Außerdem ist die Verteilung der Row Types über den Testdaten sehr genau, wie in Abbildung 16 zu sehen ist. Mit einer perfekten Prediction müssten, die Verhältnisse der Row Types gegen 16, $\frac{2}{3}$% gehen.

Fazit

Dieses Training war ein voller Erfolg. Es ergibt eine hohe Testgenauigkeit und hat weniger Parameter als das Original Modell aus dem pix2code-Paper.

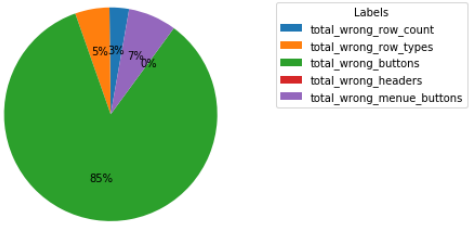

Abbildung 13: Gesamt

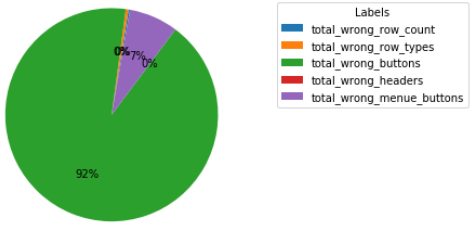

Abbildung 14: Ohne p80

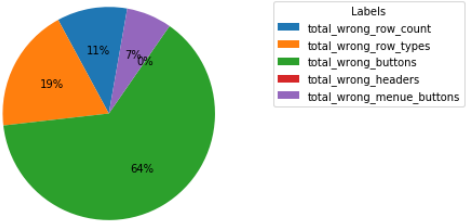

Abbildung 15: Nur p80

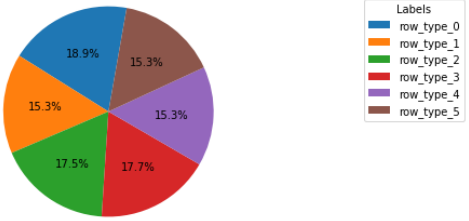

Abbildung 16: Verteilung Row Types

6.8. 8. Trainingsversuch

Einführung

In diesem Versuch wird eine Kombination aus der Original Architektur mit LSTMs und der neuen mit GRUs aus dem siebten Trainingsversuch getestet. Das Sprach-Modell wird wie im Original mit LSTMs gebildet, der Decoder hingegen mit den GRUs.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Wie folgt wurde das Modell verändert:

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 48, 128)           209408      input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 48, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
gru_1 (GRU)                  (None, 48, 512)           2557440     concatenate_1[0][0]
----------------------------------------------------------------------------------
gru_2 (GRU)                  (None, 512)               1574400     gru_1[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 23)                11799       gru_2[0][0]
==================================================================================

Trainable params: 108,451,127
```

Die Ebene `sequential_2` ist durch ein LSTM realisiert, die Ebenen `gru_1` und `gru_2` jeweils durch ein GRU.

Ergebnis

Jede der getesteten Daten erzeugt den selben Output:

```
header{
logo-right,btn-inactive-white,btn-inactive-white
}
row{
quadruple{
small-title,text,btn-inactive-black
}
quadruple{
small-title,text,btn-inactive-black
```

```
}
double{
small-title,text,btn-inactive-black
}
}
row{
quadruple{
small-title,text,btn-inactive-black
}
quadruple{
small-title,text,btn-inactive-black
}
double{
small-title,text,btn-inactive-black
}
}
```

Fazit

Dieses Ergebnis legt nahe, dass das Bottleneck des Modells in dem Sprach-Modell liegt.

6.9. 9. Trainingsversuch

Einführung

Nach dem Versuch, ob es ausreicht allein den Decoder durch ein GRU zu ersetzen, um das Modell konvergieren zu lassen, wird nun das Gegenteil getestet: Ob es auch genügt, das Sprach-Modell mit GRUs zu realisieren.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Wie bereits in dem Original Paper ist hier der Decoder mit LSTMs realisiert, zu sehen in den Ebenen `lstm_1` und `lstm_1`. Die Ebene `sequential_2` ist wie im erfolgreichen siebten Trainingsversuch ein GRU.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 48, 128)           157056      input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 48, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
lstm_1 (LSTM)                (None, 48, 512)           3409920     concatenate_1[0][0]
----------------------------------------------------------------------------------
lstm_2 (LSTM)                (None, 512)               2099200     lstm_1[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 23)                11799       lstm_2[0][0]
==================================================================================

Trainable params: 109,776,055
```

Durch das Benutzen von LSTMs im Decoder, ist die Anzahl der Parameter wieder auf knappe 110 Millionen gewachsen.

Ergebnis

Ähnlich wie im siebten Versuch konnte das Netzwerk konvergieren und einen nicht allgemeinen Output erzeugen. Die Analyse der Testdaten zeigte jedoch, dass insgesamt 143 der 619 Daten nicht kompiliert werden konnten.

Die Analyse der Fehlerverteilung ergibt folgende Lageparameter in Abbildung 17. Die Verteilung der Fehler-Scores ist in Abbildung 18 gegeben.

Im Vergleich zu den Lageparametern 11 des siebten Experiments, performt dieses Modell sehr viel schlechter. Der Median Fehler-Score dieses Experiments ist fünfmal so hoch wie die des siebtens - obwohl sogar 143 Predictions nicht kompiliert wurden und nicht mit in den Score fließen.

Im Weiteren wird das letzte Quintil der Fehler analysiert und mit dem Rest verglichen. In der Abbildung 19 ist die Fehlerverteilung aller Testdaten zu sehen, in Abbildung 20 die besten 80% und in Abbildung 21 die schlechtesten 20%.

Arithmetisches Mittel 33.50

Geometrisches Mittel 30.23

Harmonisches Mittel 26.45

Quintile p20: 20, p40: 26, p60: 41, p80: 46

Median 36

Modus 46

Vorkommen Modus 32

Gesamt Fehler Score 20734

Anzahl Token Testset 36751

Anzahl generierter Token 49012

Score pro Token in Testset 0.56

Abbildung 17: Lageparameter

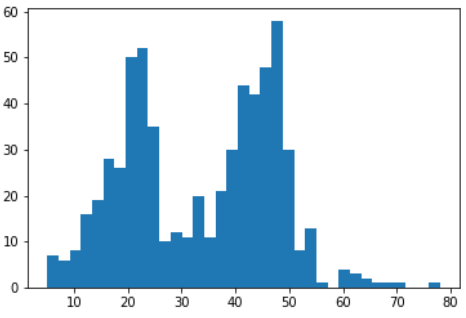

Abbildung 18: Fehlerverteilung

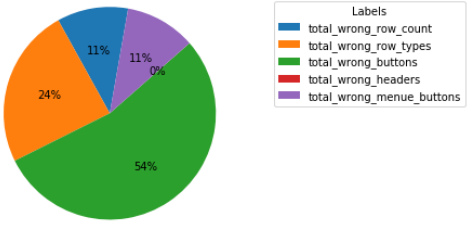

Abbildung 19: Gesamt

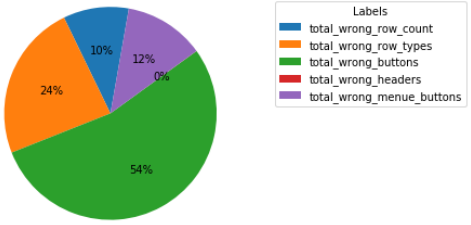

Abbildung 20: Ohne p80

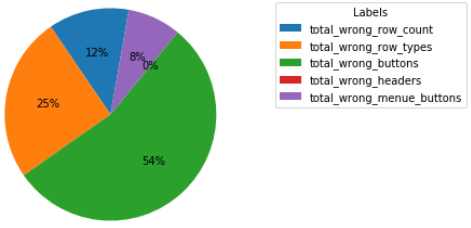

Abbildung 21: Nur p80

Der Unterschied in der Verteilung der Fehlertypen zwischen den ersten 4 Quintilen und dem letzten Quintil ist hier viel geringer als in dem siebten Experiment. Diesem Modell ist es nicht gelungen, gute Ergebnisse aus den Trainingsdaten zu generieren. Dies unterstreicht die Verteilung der generierten Row-Types in Abbildung 22. Hier ist zu sehen, dass das Modell nicht lernen konnte, die verschiedenen Typen sicher zu unterscheiden. Ein Großteil wird als Typ 4 erkannt, ca. 60%, und der Rest ist nicht regelmäßig verteilt.

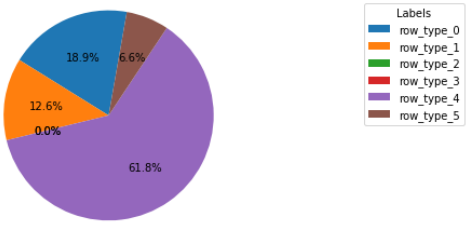

Abbildung 22: Verteilung Row Types

Fazit

Die hier benutzte Konfiguration ist schlechter als die reine GRU-Lösung.

6.10. 10. Trainingsversuch

Einführung

Mit diesem Versuch wird getestet, ob eine LSTM-Lösung mit mehr Parametern funktionieren könnte und das GRU vielleicht doch noch einholt.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Um zu überprüfen, ob es ausreicht, das Sprach-Modell komplexer zu gestalten, wurde die Anzahl der LSTM Units dieses Teils verdoppelt.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)  0
```

```
__________________________________________________________________________________
input_2 (InputLayer)            (None, 48, 23)       0
__________________________________________________________________________________
sequential_1 (Sequential)       (None, 48, 1024)     104098080   input_1[0][0]
__________________________________________________________________________________
sequential_2 (Sequential)       (None, 48, 256)      812032      input_2[0][0]
__________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 48, 1280)     0           sequential_1[1][0]
                                                                 sequential_2[1][0]
__________________________________________________________________________________
lstm_3 (LSTM)                   (None, 48, 512)      3672064     concatenate_1[0][0]
__________________________________________________________________________________
lstm_4 (LSTM)                   (None, 512)          2099200     lstm_3[0][0]
__________________________________________________________________________________
dense_3 (Dense)                 (None, 23)           11799       lstm_4[0][0]
==================================================================================

Total params: 110,693,175
```

In der Ebene `sequential_2`, kommen 256 anstatt 128 Units vor.

Ergebnis

Das Training konvergierte, erzeugt aber ein schlechtes Ergebnis. Es konnten zwar alle 619 Testdaten kompiliert werden, die Scorings sind jedoch, nicht Konkurrenzfähig im Vergleich zu dem siebten Versuch - aber besser als die Ergebnisse des Neunten.

Die Analyse der Fehlerverteilung ergibt folgende Lageparameter in Abbildung 23. Die Verteilung der Fehler-Scores ist in Abbildung 24 gegeben.

Auffallend ist hier, dass die Verteilung der Fehler Scores viel gleichmäßiger als in den Versuchen Sieben und Neun ist. Die Abstände der verschiedenen Mittel sowie die der Quintile sind geringer. Im Vergleich zum neunten Versuch ist der Median Score um 10 geringer.

Auffallend ist jedoch in Abbildung 25, dass dieses Modell den bisher höchsten Anteil an falsch zugeordneten `header`-Elementen aufweist. Die anderen konvergierten Modelle konnten diese jeweils perfekt zuordnen.

Schließlich kommt der letzte Punkt der Analyse, die Verteilung der generierten Row Typen. Hier schneidet dieses Modell am aller schlechtesten ab. Es generiert für jede Datei aus dem Testset immer nur einen von zwei Row Types.

Arithmetisches Mittel 23.81

Geometrisches Mittel 22.98

Harmonisches Mittel 22.14

Quintile p20: 19, p40: 22, p60: 24, p80: 31

Median 23

Modus 22

Vorkommen Modus 57

Gesamt Fehler Score 14739

Anzahl Token Testset 36751

Anzahl generierter Token 34664

Score pro Token Testset 0.40

Abbildung 23: Lageparameter

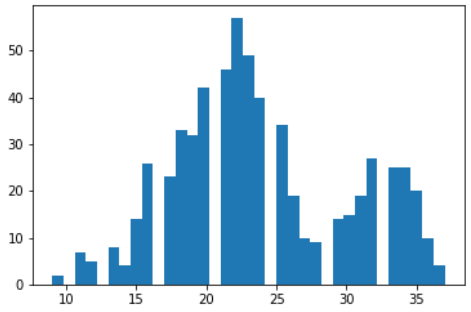

Abbildung 24: Fehlerverteilung

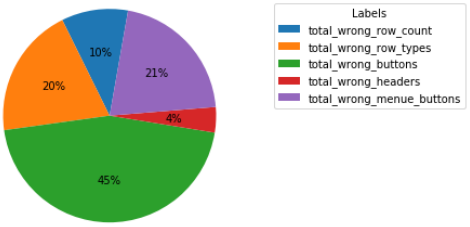

Abbildung 25: Gesamt

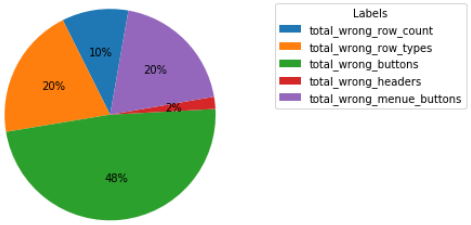

Abbildung 26: Ohne p80

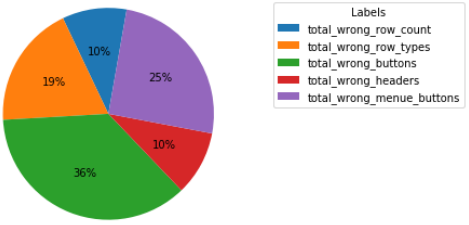

Abbildung 27: Nur p80

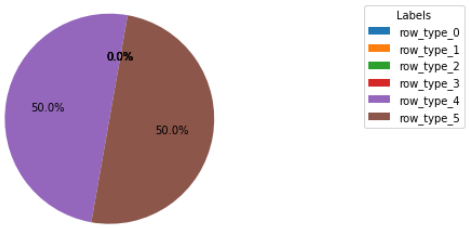

Abbildung 28: Verteilung Row Types

Fazit

Obwohl dieses Modell mit großer LSTM Architektur knappe 2.5 Millionen Parameter mehr hat, sind die Ergebnisse viel schlechter. Besonders die drei fehlenden Row-Types fallen hier ins Gewicht. Zunächst wurde angenommen, dass dieses Modell gleich gut oder besser wie die reine GRU Architektur aus Versuch Sieben sein könnte. Die LSTMs performen bisher schlechter für diesen Task.

6.11. 11. Trainingsversuch

Einführung

In diesem Versuch soll herausgefunden werden, ob eine größere GRU-Architektur ein noch besseres Ergebniss liefern könnte.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Es wurden sowohl die Sprach-Ebene, als auch die Decoder-Ebene vergrößert.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
```

```
sequential_2 (Sequential)       (None, 48, 192)      346176      input_2[0][0]
__________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 48, 1216)     0           sequential_1[1][0]
                                                                 sequential_2[1][0]
__________________________________________________________________________________
gru_3 (GRU)                     (None, 48, 768)      4573440     concatenate_1[0][0]
__________________________________________________________________________________
gru_4 (GRU)                     (None, 768)          3541248     gru_3[0][0]
__________________________________________________________________________________
dense_3 (Dense)                 (None, 23)           17687       gru_4[0][0]
==================================================================================

Total params: 112,576,631
```

Hier kann man sehen, dass in der Ebene `sequential_2` und in der Ebene `gru_3` die Unit-Anzahl vergrößert wurde. Von 128 auf 192, bzw. von 512 auf 786.

Ergebnis

Auch dieses Netz konvergiert, leider ist das Ergebnis gegenüber Versuch Sieben nicht besser geworden.

Der Hauptteil der Fehler-Scores liegt um die Zwanzig, mit ein paar wenigen im niedrigeren oder höheren Bereich. Zu sehen ist das sehr gut in der Abbildung 30.

In den Abbildungen 31- 33 ist zu beobachten, dass die Verteilung der Fehler-Type nach Fehler Anzahl relativ gleichmäßig liegt. Die Daten, welche die meisten Fehler erzeugt haben, verursahcen die gleichen Arten wie die Gesamtmenge. Daher liegt die Fehlerursache nicht in besonders komplexen Screenshots, sondern bei einer groben Klassifizierung.

Schlussendlich ist in Abbildung 34 zu sehen, wie schlecht das Modell bei der richtigen Einordnung von den Row-Types anschneidet. Es kann nicht zwischen den einzelnen Typen unterscheiden.

Fazit

Wieso performt dieses Modell, obwohl es doch größer ist so viel schlechter als Versuch Sieben? Deutlich sichtbar hat man es mit einem klassischen Fall des Overfittings zu tun. Das Modell war hier so komplex, dass es die Beschreibungen aller Trainingsdaten auswendig lernen konnte. Dadurch sah die Konvergenz das Training gut aus, die Tests zeigte aber, dass es nicht qualifiziert verallgemeinern konnte.

Arithmetisches Mittel 20.10

Geometrisches Mittel 19.23

Harmonisches Mittel 18.38

Quintile p20: 16, p40: 19, p60: 21, p80: 22

Median 20

Modus 21

Vorkommen Modus 85

Gesamt Fehler Score 12442

Anzahl Token Testset 36751

Anzahl generierter Token 33680

Score pro Token Testset 0.34

Abbildung 29: Lageparameter

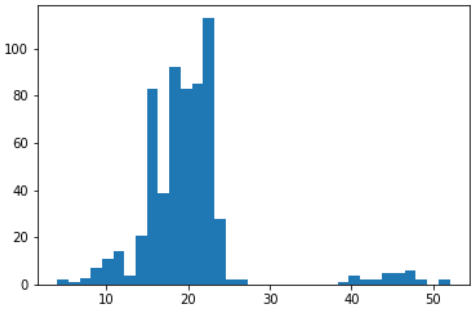

Abbildung 30: Fehlerverteilung

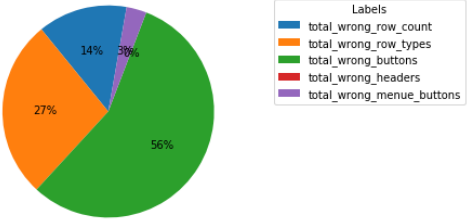

Abbildung 31: Gesamt

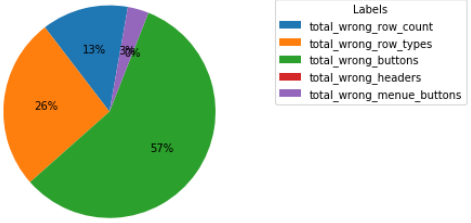

Abbildung 32: Ohne p80

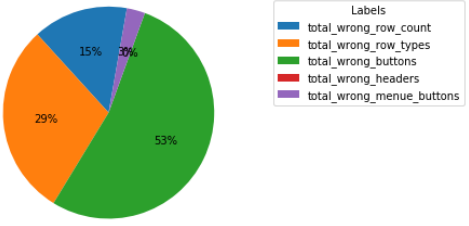

Abbildung 33: Nur p80

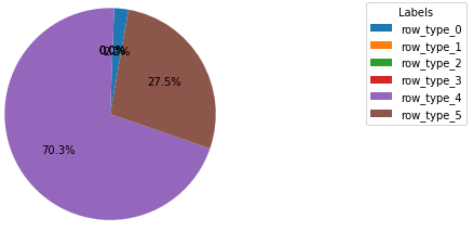

Abbildung 34: Verteilung Row Types

6.12. 12. Trainingsversuch

Einführung

In diesem Versuch wird überprüft, ob eine Vergrößerung der `CONTEXT_LENGTH` das Ergebnis positiv beeinflusst, und wenn ja wie stark.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Die `CONTEXT_LENGTH` wurde von 48 auf 64 (Anmerkung: die Median Token-Sequenz-Länge des Trainingssets ist 62) erhöht. Diese wirkt sich folgendermaßen auf die Architektur aus:

```
_________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
_________________________________________________________________________________
input_2 (InputLayer)         (None, 64, 23)            0
_________________________________________________________________________________
sequential_1 (Sequential)    (None, 64, 1024)          104098080   input_1[0][0]
_________________________________________________________________________________
sequential_2 (Sequential)    (None, 64, 128)           157056      input_2[0][0]
_________________________________________________________________________________
concatenate_1 (Concatenate)  (None, 64, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
_________________________________________________________________________________
gru_3 (GRU)                  (None, 64, 512)           2557440     concatenate_1[0][0]
```

Arithmetisches Mittel 6.44

Geometrisches Mittel 5.78

Harmonisches Mittel 5.07

Quintile p20: 4, p40: 6, p60: 7, p80: 8

Median 6

Modus 6

Vorkommen Modus 116

Gesamt Fehler Score 3989

Anzahl Token Testset 36751

Anzahl generierter Token 36869

Score pro Token Testset 0.1085

Abbildung 35: Lageparameter

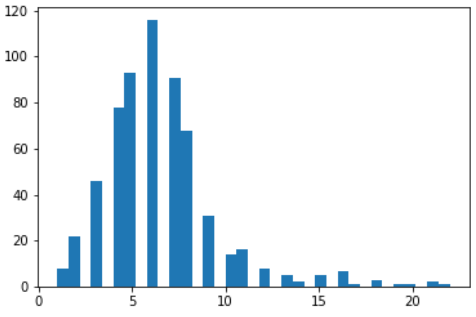

Abbildung 36: Fehlerverteilung

```
_________________________________________________________________________
gru_4 (GRU)                  (None, 512)               1574400     gru_3[0][0]
_________________________________________________________________________
dense_3 (Dense)              (None, 23)                11799       gru_4[0][0]
=========================================================================

Trainable params: 108,398,775
```

Der `Output Shape` der Ebenen `input_2`, `sequential_1`, `sequential_2`, `concatenate_1` und `gru_3` spiegelt jeweils in der ersten Dimension die nun erhöhte `CONTEXT_LENGTH` wieder. Ansonsten ist das Modell gleich wie das des siebten Versuchs.

Ergebnis

Im Vergleich zum Modell des siebten Versuches, performt dieses Modell besser, siehe 11. Der Fehler-Score ist etwas geringer, aber im Modell von Versuch Sieben ist das zweite Quintil, sowie der Modus jeweils um eine Einheit niedriger.

Die Verteilung der Fehler im letzten Quintil ist ebenfalls sehr gut; es gab nur 22 mal das Auftreten von falschen Row-Klassifizierungen. Zum Vergleich: im siebten Versuch gab es 174 falsche Klassifizierungen des Row-Types.

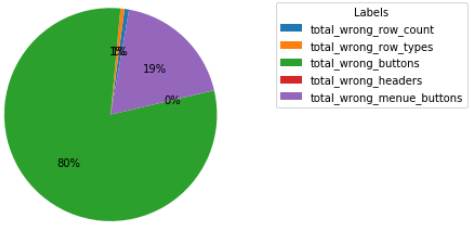

Abbildung 37: Gesamt

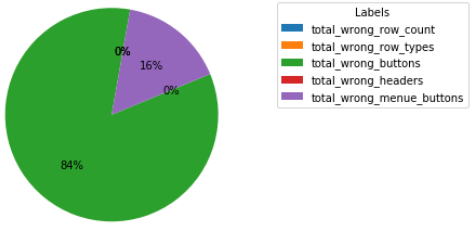

Abbildung 38: Ohne p80

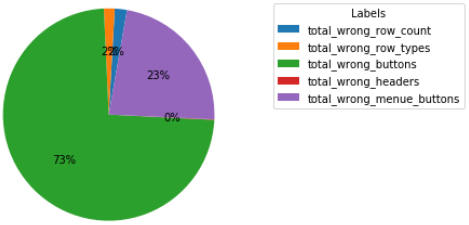

Abbildung 39: Nur p80

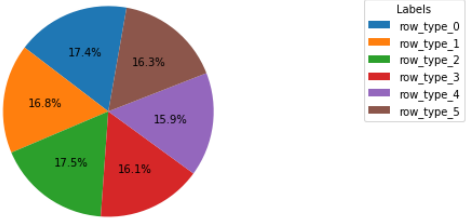

Abbildung 40: Verteilung Row Types

Daher ist auch die Verteilung der Row Types sehr gleichmäßig, zu sehen in Abbildung 40. Der Fehler Score ist gegenüber Versuch Sieben um ca. 18% gesunken. Interessant ist, dass dieses Modell 276% der Fehler des siebten Versuchs in den Menü-Buttons macht. Außerdem hat es einmal einen falschen Header identifiziert. Bei der Anzahl der falschen Buttons gibt es kaum einen Unterschied zum siebten Modell.

Fazit

Aufgrund der längeren `CONTEXT_LENGTH` ist das Ergebnis sehr viel besser geworden. Row-Types werden hier mit am zuverlässigsten erkannt, was wahrscheinlich an der verlängerten `CONTEXT_LENGTH` liegt, da das Modell so auf jeden Fall die aktuelle Row sowie auch die vorangehenden gleichzeitig im Speicher haben kann. Leider gibt es Abstriche in der

Qualität der Menüs, da gab es fast 3 mal so viele Fehler wie im Vergleich. Die ist aber zu verschmerzen, da Performance der Rows so außerordentlich gut ist.

6.13. 13. Trainingsversuch

Einführung

Hier wird eine vereinfachte Variante des siebten Modells genutzt, um zu testen, ob es so eine bessere Generalisierung erreicht wird.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Sowohl der Sprach- als auch der Decoder-Teil des Modells wurden verkleinert. Die Anzahl der rekurrenten `Units` sind von 128 auf 92 im Sprach-Teil und von 512 auf 386 im Decoder-Teil reduziert worden.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 48, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 48, 1024)          104098080   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 48, 92)            83076       input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 48, 1116)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
gru_3 (GRU)                  (None, 48, 386)           1740474     concatenate_1[0][0]
----------------------------------------------------------------------------------
gru_4 (GRU)                  (None, 386)               895134      gru_3[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 23)                8901        gru_4[0][0]


Trainable params: 106,825,665
```

Arithmetisches Mittel 16.71

Geometrisches Mittel 15.25

Harmonisches Mittel 13.62

Quintile p20: 11, p40: 15, p60: 17, p80: 22

Median 16

Modus 16

Vorkommen Modus 54

Gesamt Fehler Score 10342

Anzahl Token Testset 36751

Anzahl generierter Token 36047

Score pro Token Testset 0.28

Abbildung 41: Lageparameter

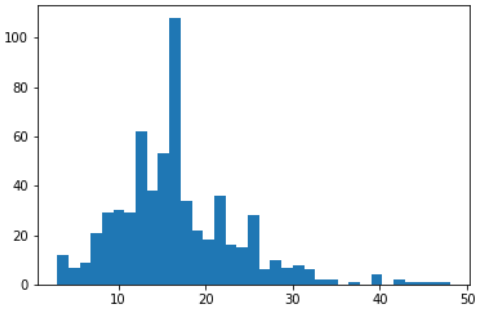

Abbildung 42: Fehlerverteilung

Diese Änderung sieht man in den Ebenen `sequential_2` sowie `gru_2` und `gru_4`. Durch die Verkleinerung der RNNs, ist das Modell um knapp zwei Millionen Parameter kleiner geworden.

Ergebnis

Wie in Abbildung 47 zu sehen ist, sind alle Lageparameter schlechter als die bisherigen guten Ergebnisse.

Im Vergleich der Lageparameter mit dem dem zehnten Trainingsversuch (LSTM mit mehr Parametern), gibt es hier einen geringeren Fehler-Score. Es ist jedoch zu sehen, an Abbildungen 43 bis 45, dass die Fehlerverteilung überall gleich ist, egal in welchem Quintil.

An der Verteilung der generierten Row Types in Abbildung 46 erkennt man, dass diese Modell es nicht gelernt hat, die Typen zuverlässig zu unterscheiden.

Fazit

Durch das Reduzieren der `Units` der RNNs ist das Modell schlicht zu simpel geworden. Obwohl es die richtige Grammatik lernen konnte - es hat keine Token-Sequenzen erzeugt, die nicht kompiliert werden konnte.

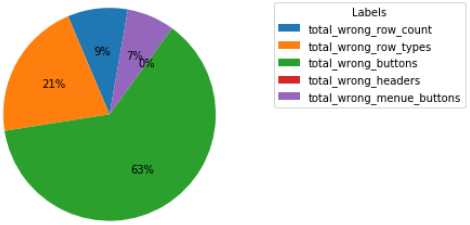

Abbildung 43: Gesamt

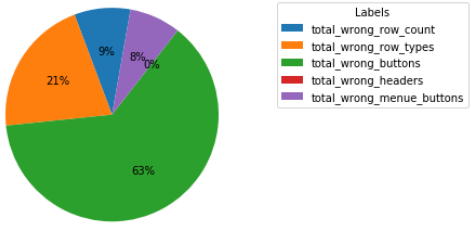

Abbildung 44: Ohne p80

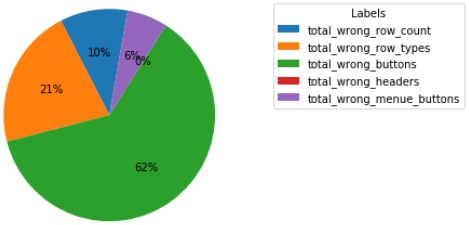

Abbildung 45: Nur p80

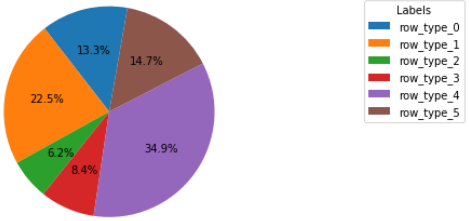

Abbildung 46: Verteilung Row Types

6.14. 14. Trainingsversuch

Einführung

Der Ansatz aus dem 12. Versuch, die vergrößerte `CONTEXT_LENGTH` wurde hier verwendet, in Kombination mit einem etwas größeren Netzwerk. In den bisherigen Versuchen gab es stets viele falsch farbige Buttons, vielleicht kann dieses Problem mit einer leicht vergrößerten Decoder Architektur behoben werden.

Datenset

Hier wurde das Datenset auf dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Die Anzahl der GRUs in dem Decoder-Teil des Netzwerkes wurde von 512 auf 564 erhöht.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
----------------------------------------------------------------------------------
input_2 (InputLayer)         (None, 64, 23)            0
----------------------------------------------------------------------------------
sequential_1 (Sequential)    (None, 64, 1024)          104097600   input_1[0][0]
----------------------------------------------------------------------------------
sequential_2 (Sequential)    (None, 64, 128)           157056      input_2[0][0]
----------------------------------------------------------------------------------
concatenate_1 (Concatenate)  (None, 64, 1152)          0           sequential_1[1][0]
                                                                   sequential_2[1][0]
----------------------------------------------------------------------------------
gru_3 (GRU)                  (None, 64, 564)           2905164     concatenate_1[0][0]
----------------------------------------------------------------------------------
gru_4 (GRU)                  (None, 564)               1910268     gru_3[0][0]
----------------------------------------------------------------------------------
dense_3 (Dense)              (None, 23)                12995       gru_4[0][0]
==================================================================================
Trainable params: 109,083,083
```

Sichtbar ist dies in den Ebenen `gru_3` und `gru_4`.

Ergebnis

Diese Model performt sogar noch besser als das Modell des Siebten und Zwölften Versuches.

Im Vergleich zum Versuch Zwölf, macht dieses Modell sehr viel weniger Fehler bei Bestimmung der richten Anzahl der Menü Buttons (223 gegenüber 780) aber bei Bestimmung der richtigen Buttons im Content ist Performance ein wenig schlechter (2963 gegenüber 2751). Das aktuelle Modell bestimmt mehr falsche Row Types, 32 gegenüber 13 in Versuch Zwölf, aber dafür generiert es weniger oft die falsche Row Anzahl (33 gegenüber 45).

Arithmetisches Mittel 5.85

Geometrisches Mittel 5.06

Harmonisches Mittel 4.43

Quintile p20: 3, p40: 5, p60: 6, p80: 7

Median 5

Modus 6

Vorkommen Modus 116

Gesamt Fehler Score 3621

Anzahl Token Testset 36751

Anzahl generierter Token 36879

Score pro Token Testset 0.0985

Abbildung 47: Lageparameter

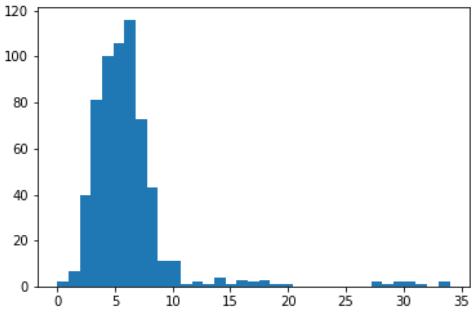

Abbildung 48: Fehlerverteilung

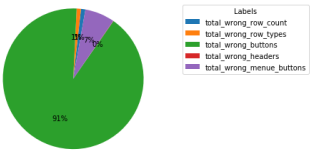

Abbildung 49: Gesamt

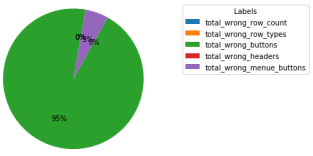

Abbildung 50: Ohne p80

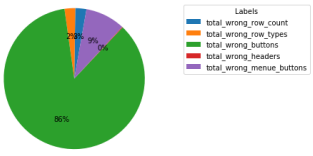

Abbildung 51: Nur p80

Fazit

Der Fehler Score ist etwas geringer als im Versuch 12, aber im Grunde ist die Performance der beiden Modelle ziemlich ähnlich. Die geringe Vergrößerung des Decoder-Teils, hat so nicht zu einer signifikanten Verbesserung beigetragen. Da das Netzwerk bei einer stärkeren Vergrößerung zum Overfitting neigt (vergleiche mit Versuch 11, Abschnitt 6.11, ist diese Konfiguration des Netzwerkes die wohl Effektivste.

Versuch	7	12	14
Fehler Score	4840	3989	3621
Fehler/Token	0.1317	0.1085	0.0985
Anzahl Parameter	108.398.775	108.398.775	109.083.083

Tabelle 1: Für jede der Metriken gilt: Je kleiner, desto besser

7. Zusammenfassung der Versuchsergebnisse

Trotz der vielen Fehlschläge, gelang es eine Architektur zu finden, welche die Ansprüche der DSL und der Komplexität der Screenshots abbilden konnte. Viele der Versuche sind nicht konvergiert oder hatten die Tendenz die Trainingsdaten auswendig zu lernen. Schließlich wurde eine Lösung gefunden, die viele Probleme gelöst hat: Das Ersetzen der LSTMs zu GRUs zeigte einen erstaunlichen Performance-Schub. Außerdem konnte mit einer Erhöhung der Kontext Länge die Architektur aus Versuch 7 und 12 noch mal verbessert werden, um schließlich die Konfiguration des 14. Versuches zu finden.

In Tabelle 1 sind die drei Modelle mit der besten Performance aufgeführt. Eine kleine Vergrößerung der GRU-Architektur in Kombination mit der leicht erhöhten Kontext Länge im Versuch 14 hat das beste Ergebnis erzielt. Dieses Modell hat immer noch ca. 800.000 Parameter weniger als die Originale LSTM Architektur, performt aber um ein vielfaches besser auf den hier verwendeten Daten.

In vielen Bereichen der Sequenz Modellierung performen LSTMs und GRUs auf einer Ebene ohne einen signifikanten Gewinner auf einer der Seiten. Hier, im Bereich der Sprach-Modellierung in Verbindung mit der Analyse eines Screenshots gibt es aber einen klaren Gewinner: Die **Gated Recurrent Unit**.

Verschiedene Publikationen [10], [12], [19] deuteten stets auf eine ungefähre Ausgewogenheit der Fähigkeiten der beiden RNN Varianten, aber auch darauf, dass, je nach spezieller Domäne eines Problems, eine Variante besser als die andere sein kann.

Die Annahme, die im Rahmen dieser Arbeit getroffen wird, ist, dass die innere Struktur der GRU, siehe Abbildung 4, besonders gut die Eltern/Kind Beziehungen der verwendeten DSL abbilden kann. Durch die, in GRUs verwendete, Kopplung der `forget`- und `add`-Gates kann diese Beziehung vielleicht besser abgebildet werden als in dem Cell-State des LSTMs. Vielleicht haben Fehler während des Löschens von Informationen aus dem Cell-State oder während dem Hinzufügen von Informationen zu demselben, bei LSTMs das Halten der Input-Historie zu kompliziert gemacht und verfälscht. Denn wenn der Zustand der Zellen nicht gut genug verwaltet wird, können die Langzeit-Beziehungen innerhalb der Daten nicht richtig modelliert werden, insbesondere die Eltern/Kind Beziehungen der DSL. Dies sieht man besonders bei Versuch zehn, wo auch eine besonders große LSTM-Architektur ein nur dürftiges Ergebnis produziert. In Abbildung 6.10 erkennt man das wirklich schlechte Modellieren der Rows in diesem Ansatz. In Versuch

Sieben und Zwölf auf der anderen Seite, sind diese Rows stets korrekt modelliert, woraus geschlossen werden könnte, dass diese GRUs besser in der Lage sind, Eltern/Kind Beziehungen zu modellieren.

So konnte auf der Suche nach den richtigen Parametern ein Modell gefunden werden, welches den Vorteil der GRUs nutzt und dadurch in der Lage war zu lernen, richtigen Code aus den Input-Bildern zu erstellen. Bei über 80% der getesteten Bilder waren die einzigen Fehler, die das Netzwerk gemacht hat, eine falsche Button-Farbe, größere und wichtigerere Elemente konnten stets korrekt vorhergesagt werden.

Dies führt zu einem Ende der Zusammenfassung der getätigten Experimente. Es wird die Architektur des 14. Versuches präsentiert, um einen Teil zu der Automatisierung der Frontend-Entwicklung bei zu tragen.

Zusätzlich zu den Experimenten im Kapitel 6 wurden im Anhang zwei weitere Arten von Versuchen durchgeführt. Der erste Versuch, siehe Abschnitt A, evaluiert, ob eine anderes Vision-Modell besser abschneidet als die im Original Paper benutzte Architektur. Hier konnte jedoch keine Verbesserung festgestellt werden. Der zweite Versuch, siehe Abschnitt B, testet ob die verwendeten Architekturen auch lernen können, mit einer Skizze einer Website den dazugehörigen Code zu generieren. Dort werden einige Ansätze und Versuche beschrieben, die in einer ausgebauten Form bestimmt ebenso gute Ergebnisse erzielen könnten wie die Code-Generierung aus den Screenshots. Leider ist noch sehr viel Arbeit nötig um an diesen Punkt zu gelangen.

8. Fazit

Das Ziel dieser Arbeit war, folgender Frage auf den Grund zu gehen: Ist es möglich auch komplexere Websites mit dem pix2code Modell abzubilden?

Nein, das ist es nicht. Um komplexere Websites abzubilden, wird ein erweitertes Sprach-Vokabular benötigt und die Original Architektur scheiterte daran, diese zu verstehen. Die zugrunde liegende Original Architektur musste angepasst werden, damit die neuronalen Netzwerke zu einer akzeptablen Lösung konvergieren konnten. Dies wurde durch das Verwenden einer anderen Art von rekurrenten Netzwerken erreicht. Das Ersetzen der LSTMs zu GRUs hatte die Folge, dass das Modell den Zusammenhang zwischen DSL und Input-Bilder viel genauer abbilden konnte. So ist ein Beitrag dieser Arbeit, die verbesserte Architektur, aber auch eine Evaluation, welche Art von rekurrenten Netzwerken besser ist, um eine HTML/CSS Markup beschreibende DSL zu modellieren. So kann ein neue, auf dem Original aufbauende, Architektur veröffentlicht werden, um Code aus Screenshots zu erstellen und die Empfehlung ausgesprochen werden, GRUs an Stelle von LSTMs für diesen Task mit dem hier genutzten Datenset zu verwenden.

Ein weiterer Beitrag ist ein synthetisiertes Datenset, aus Website-Screenshots und dazugehörigen Token-Sequenzen. Mit diesem Datenset können viele weitere Experimente durchgeführt werden, um herauszufinden wie das Modellieren von Websites aus Screenshots noch besser funktionieren kann. Dieses Datenset enthält zwei verschiedene Ausführungen, nämlich die Screenshots die für das Training der Experimente genutzt wurden, aber auch ein zweites Set, welches Websites aus besteht, die so aussehen als ob diese skizziert sind. Damit können neuronale Netzwerke trainiert werden, um Websites aus Skizzen zu erstellen.

Die Ergebnisse dieser Arbeit sind keinesfalls ausreichend, um die Frontend-Entwicklung von einem Tag auf den nächsten vollständig zu automatisieren. Sie ist viel mehr einer von vielen Schritten, die auf dem Weg dorthin gemacht werden müssen.

Einer der nächsten Ansätze wäre es, auf den Zwischenschritt der Token-Sprache zu verzichten und ein System zu erstellen, was eine Trennung von Content und Style ermöglicht. So könnte ein erstes Netzwerk rohen HTML-Code erstellen, ein zweites CSS-Styles einfügen, um den Content richtig zu positionieren und zu stylen. Außerdem können die beiden Netzwerk-Teile unabhängig von einander optimiert werden, um so zu einer jeweils höher spezialisierten Lösung zu gelangen.

Ebenfalls wäre es möglich und praktikabel, für das Training des Vision-Teils des Netzwerkes eine Autoencoder-Architektur mit einzubauen. So kann wahrscheinlich eine bessere Repräsentation des Input-Bildes gefunden werden.

Literatur

[1] Electron. https://www.electronjs.org/.

[2] Gimp dokumentation. https://docs.gimp.org/2.8/en/plug-in-convmatrix.html.

[3] pix2code. https://www.github.com/tonybeltramelli/pix2code.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[5] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017.

[6] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[7] François Chollet et al. Keras. https://keras.io, 2015.

[8] François Chollet et al. Keras, vergleich der modelle, 2015. https://keras.io/applications/#documentation-for-individual-models.

[9] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[10] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[11] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[12] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[13] Geoffrey Hinton. Rmsprob. http://www.cs.toronto.edu/%7Etijmen/csc321/slides/lecture_slides_lec6.pdf.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation, 9(8)*, page 1735–1780, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf.

[15] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, page 215–243, 1968. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1557912.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[17] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259, 2015.

[18] Christopher Olah. Colah's blog cnn, 2014. `http://colah.github.io/posts/2014-07-Understanding-Convolutions/`.

[19] Christopher Olah. Understanding lstm networks, 2015. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[20] Günter Daniel Rey and Fabian Beck. Neuronale netze - eine einführung. `http://www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf`.

[21] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[23] Chen Xing, Devansh Arpit, Christos Tsirigotis, and Yoshua Bengi. A walk with sgd. *stat.ML*, 2018.

A. Experiment: Kann ein besseres Vision-Modell die Performance verbessern?

Einführung

Bei diesem Experiment wird ein anderes Vision-Modell getestet. Anstatt einer auf VGGNet bestehenden Architektur, wird eine aus dem Xception [9] Modell erstellt. Diese Modell performt auf dem ImageNet Dataset um ca. 8% besser als VGGNet bei ca. einem Sechstel der Parameter [8]. Außerdem kann dieses Modell mit trainierten Parametern direkt aus Keras geladen werde, so können die Vorteile von Transferlernen ausgenutzt werden.

Datenset Hier wurde das Datenset aus dem sechsten Versuch genutzt.

Veränderte Parameter

```
__________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
input_15 (InputLayer)           (None, 256, 256, 3)  0
__________________________________________________________________________________
input_16 (InputLayer)           (None, 48, 23)       0
__________________________________________________________________________________
sequential_10 (Sequential)      (None, 48, 1024)     21886504    input_15[0][0]
__________________________________________________________________________________
sequential_11 (Sequential)      (None, 48, 128)      157056      input_16[0][0]
__________________________________________________________________________________
concatenate_3 (Concatenate)     (None, 48, 1152)     0           sequential_10[1][0]
                                                                 sequential_11[1][0]
__________________________________________________________________________________
gru_11 (GRU)                    (None, 48, 564)      2905164     concatenate_3[0][0]
__________________________________________________________________________________
gru_12 (GRU)                    (None, 564)          1910268     gru_11[0][0]
__________________________________________________________________________________
dense_11 (Dense)                (None, 23)           12995       gru_12[0][0]
==================================================================================
Total params: 26,871,987
Trainable params: 20,868,771
Non-trainable params: 6,003,216
```

Die Ebene `sequential_10` enthält statt der VGGNet-ähnlichen Architektur nun ein Xception Modell. Das Xception Modell enthält ca. 21 Millionen Parameter, von denen ungefähr 5 Millionen eingefroren wurden. Da das Modell schon vortrainiert ist, wurden die ersten 6 von insgesamt 12 Xception Module eingefroren. Die in diesen Modulen gelernten Features werden beibehalten, nur die höheren werden während dem Training optimiert. Dies spart Zeit und erhöht die Genauigkeit des Netzwerkes. Nochmal zum Vergleich: Das Netzwerk aus dem der Original Architektur hatte 109.829.432 trainierbare Parameter.

Ergebnis

Wie in den Abbildungen 52 bis 57 zu sehen ist, performt diese Netzwerk nicht besonders gut. Die Fehler Score von 12442 über das Test ist nicht konkurrenzfähig zu den anderen Architekturen.

Arithmetisches Mittel 20.10

Geometrisches Mittel 19.23

Harmonisches Mittel 18.38

Quintile p20: 16, p40: 19, p60: 21, p80: 22

Median 20

Modus 21

Vorkommen Modus 85

Gesamt Fehler Score 12442

Anzahl Token Testset 36751

Anzahl generierter Token 33680

Score pro Token Testset 0.34

Abbildung 52: Lageparameter

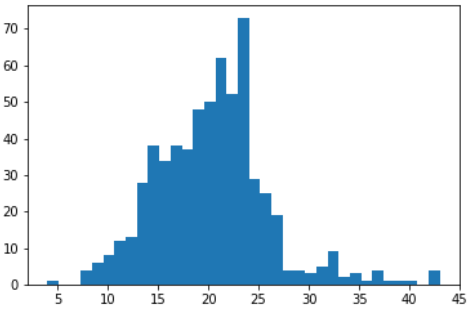

Abbildung 53: Fehlerverteilung

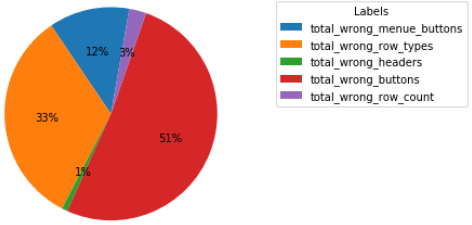

Abbildung 54: Gesamt

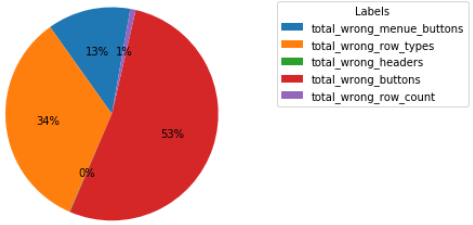

Abbildung 55: Ohne p80

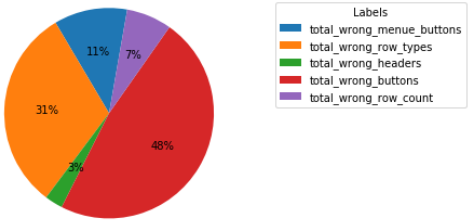

Abbildung 56: Nur p80

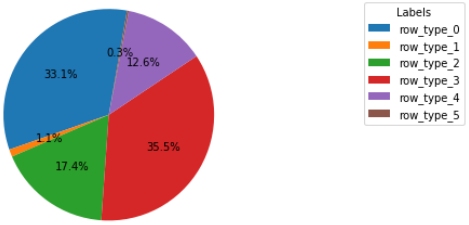

Abbildung 57: Verteilung Row Types

Fazit

Da das Xception-Modell, in vielen anderen Domänen, besser performt als das VGGNet-Modell wurde angenommen, dass es auch hier besser sein könnte. Der Ansatz nur das Vision-Modell zu ersetzten, schlug leider fehl. Aufgrund der modularen Bauweise des Netzwerkes hätte es funktionieren können, aber es werden noch mehr Anpassungen benötigt, um die Architektur auf ein Xception-Netzwerk umzustellen. Weiter in diese Richtung zu gehen, würde den Rahmen dieser Arbeit sprengen.

B. Bonus Experiment: Ist das Modell komplex genug um aus einer Skizze eine Website zu erstellen?

Ein Website aus einen Screenshot eines Designs zu erstellen, ist das eine Problem, aber könnte das selbe Modell auch aus einen groben Skizze vollständiges HTML zu erlernen?

B.1. 1. Trainingsversuch

Einführung

Zunächst muss ein neues Datenset erstellt werden, mit Skizzen von Websiten anstelle von schön designten. Danach wird eins der Modelle mit guter Erkennungsrate genommen und dieses neu trainiert mit skizzierten Websiten.

Datenset

Hier wurden die Token-Sequenzen aus dem sechsten Trainingsversuch genommen, aber das Styling der Websiten wurde mit anderen CSS-Regeln verändert, damit diese Skizziert aussehen. Im ersten Schritt wurden alle Farben und großflächige Hintergründe durch Weiß ersetzt. Anschließen bekamen alle Elemente eine leicht unregelmäßige Outline. Die Ergebnisse des ersten Schritt sind zu sehen in Abbildung 58.

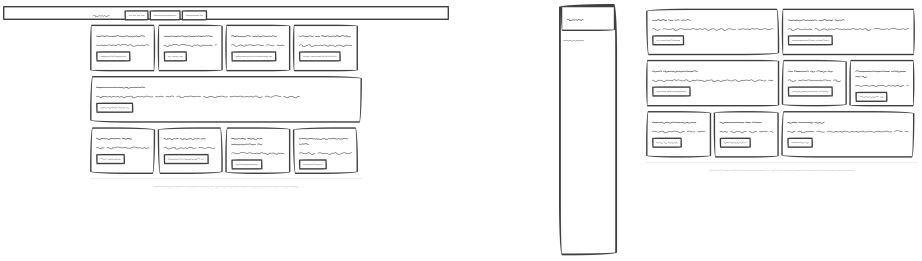

Abbildung 58: Beispiele der Skizzen

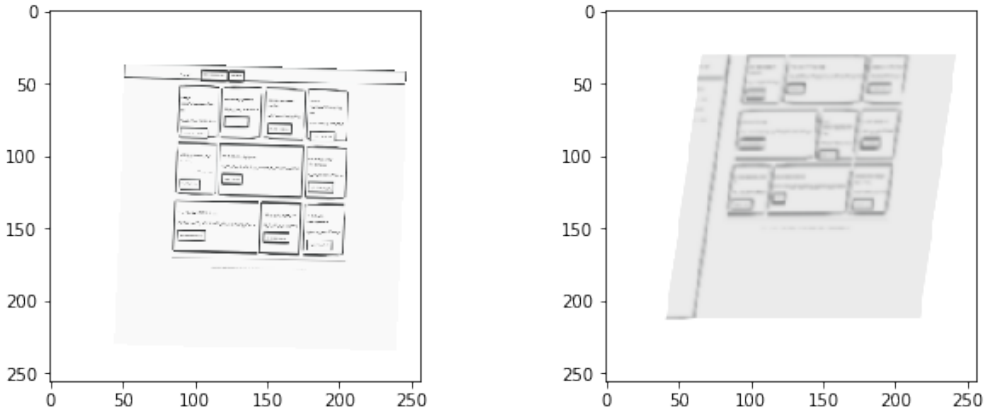

Abbildung 59: Beispiele der augmentierten Trainingsdaten

Anschließend wurden in einem Zweiten Schritt die Skizzen mit der Python Library `imagug` noch weiter verzerrt und augmentiert. Für jedes Screenshot wurden 3 unterschiedliche verzerrte Bilder erstellt und auf die Input-Größe des Netzwerkes (256 mal 256 Pixel) skaliert, siehe Abbildung 59.

Während dem Augmentieren wurden folgende Schritte ausgeführt:

Skalierung In X- und Y-Ausrichtung auf jeweils 70 bis 80% der original Größe.

Verschiebung Verschieben des Bildes um bis zu ± 10% auf beiden Achsen.

Rotation Um jeweils bis zu ±10°

Scherung Scherung des Bildes um bis zu ±8° auf beiden Achsen.

Helligkeitsreduktion Abdunkelung des Bildes bei 50% aller Bilder um bis zu 15%

Weichzeichnung 50% der Bilder wurden mit einem Sigma zwischen 0 und 1.2 weichgezeichnet.

All diese Schritte haben zum Ziel, dass das Modell mit Fotos einer Skizze arbeiten kann. Da es viel zu viel Aufwand ist, echte Skizzen als Trainingsdaten zu verwenden, wird getestet ob man diese so nachbilden kann.

Veränderte Parameter

Das Modell des letzten Versuch wurde genutzt.

Ergebnis

Leider führte das Training zu keiner Konvergenz des Netzwerkes. Siehe Abbildung 60.

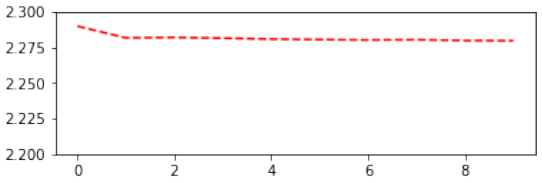

Abbildung 60: Loss des 1. Extra-Training

Fazit

Da das Datenset aus komplexeren Bildern bestand (Weichzeichnung, Rotation, veränderte Belichtung), wird angenommen, dass das Vision-Modell auf die neuen Daten underfittet. An den Sprach- und Decodier-Teil kann dies nicht liegen, da hier die Daten gleich sind. Nun muss der Vision-Teil lernen, die wichtigen Features in den viel unterschiedlicheren Daten zu erkenn, wofür es leider nicht ausreichend komplex gewesen ist.

B.2. 2. Traininsversuch

Einleitung

Hier wird ebenfalls das Xception-Netzwerk für das Visions-Modell getestet.

Datenset

Wie bei dem ersten Versuch.

Veränderte Parameter

Das Vision-Modell wurde ausgetauscht und stattdessen ein Xception-Netzwerk verwendet. Siehe die Detail im Abschnitt A

Ergebnis

Leider sind die Predictions des Netzwerkes stets eine zufällig erscheinende Abfolge von Token:

```
<START>sidebarsidebar<START>sidebar<START>sidebarsidebar
<START>sidebarsidebar<START>sidebar<START>sidebarsidebar
<START><START>sidebarsidebar<START>sidebar<START>sidebarsidebar
<START><START>sidebarsidebar<START>sidebar<START>sidebarsidebar
<START><START>sidebar<END>
```

Da eine derartige Folge ebenso erscheint, wenn ein Bild aus den Trainingsdaten gesamplet wird, stellt sich die Frage wieso der Trainings-Loss einen Verlauf wie in Abbildung 61.

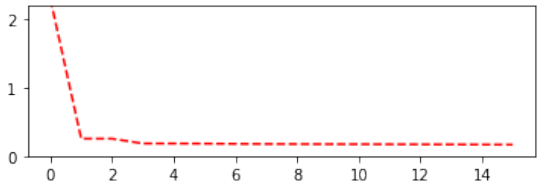

Abbildung 61: Loss des 2. Extra-Training

Fazit

Das Training dieses Netzwerkes über 15 Epochen hat insgesamt 45 Stunden gedauert. Aus diesem Grund, wird es keine weiteren Experimente mit der augmentierten Datenset von Skizzen geben. Es ist wirklich schade diese Experimente einstellen zu müssen, im Rahmen dieser Arbeit, ist aber kein Platz mehr für weitere Ausführungen. Besonders im Zusammenhang mit dem Experiment in Abschnitt A ist zu beobachten, dass die neue Vision-Architektur noch ausgiebig angepasst und verbessert werden muss.