Rapid Frontend Prototyping with Deep Learning

Simon Deussen

17. Oktober 2018

Generierung von HTML/CSS Code anhand von pixelbasierten Screenshots und -designs. Aufbauend auf dem Pix2Code Paper [6], wird diese Arbeit eine eigene Implementation einer ausführlicheren DSL erstellen. Durch automatisches Erstellen von Frontend-Code können Anwendungen wie diese rapide Entwicklungszyklen realisieren.

Inhaltsverzeichnis

1 Einleitung

Viele Client-basierte Anwendungen brauchen ein schönes Frontend. Dieses soll gleichzeitig funktional und übersichtlich sein, sowie die Firma durch gutes Design repräsentieren. Seit Apple & Co ist es unglaublich wichtig gutes Design im Frontend zu haben, da sich die Kunden sonst schnell eine Alternative suchen. Um nun ein großartiges Frontend zu realisieren benötigt man ein enge Kooperation von Designern und Entwicklern, da hier zwei, sich kaum überschneidende, Skillsets gebraucht werden. Im der klassischen Entwicklung sieht diese Zusammenarbeit folgendermaßen aus:
Ein Designer macht einen grafischen Entwurf, dieser wird vom Kunde abgenommen, dann

geht er zu dem Entwickler, der nun zu aller erst Markup für den Content und anschließend das Design und die richtige Darstellung nach bauen 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. 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 entweder nicht besonders gut die Designs zu erstellen oder darin den Markup zu exportieren.

Eine Abhilfe soll diese Arbeit liefern: Sie ermöglicht, dass der Designer mit seinem bevorzugten Tools das Design baut 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.

2 Ähnliche Arbeiten

Diese Arbeit basiert auf dem Pix2Code Paper von Tony Beltramelli [6] . Er war der erste der Code anhand von visuellen Input generieren kann. Anderen Ansätze wie DeepCoder [5] benötigen komplizierte DSL als Input und schränken so die Benutzbarkeit stark ein. Visuelle Versuche mit Android-GUIs von Nguyen [12] benötigt ebenfalls unpraktische von Experten erstellte Heuristiken. Pix2Code ist so das erste Paper das einen allgemeinen Input hat, und daraus momentan drei verschiedene Targetsprachen hat. Zum einen kann es HTMl/CSS Code erstellen, zum anderen aber auch Android- und iOS-Markup. Siehe Original Code auf Github [2]

3 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 bestehende Software-Teile zu verbinden. Diesen Trend, der für die Reduktion des Schreibens, mehr in die Richtung des Konfigurieren geht, sehe in Zukunft noch viel umfassender und allgegenwärtig. Es wird so weit gehen, dass man nicht nur wie hier in dieser Arbeit das Markup generiert sondern das aus einer Skizze sofort eine fertige Website gebaut wird, und man mit ein paar Klicks das nötige Verhalten einfach hinzufügen kann. Wobei dieser Trend nicht nur auf das Web bezogen ist. Ich denke das sich die Webtechnologien auch in der Desktop Umgebung durchsetzen. Da Plattform unabhängig und sehr stark optimiert. Sehr einfach zu lernen, weit verbreitet. Zum Beispiel Electron [1] ermöglicht den einfachen Einsatz durch einen eingebettet Browser. Daher kommt die Motivation diese Arbeit zu verfassen: Automation ist unumgänglich, deswegen sollte man diese am besten selber bauen und damit endlosen Entwicklern das Leben leichter machen.

4 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 aber im Abschnitt Überblick genauer beleuchtet.

4.1 Neuronale Netzwerke

Neuronale Netzwerke sind einfach zu benutzende Modelle, welche nicht-lineare Abhängigkeiten mit vielen latenten Variablen stochastisch abbilden können [13]. Im einfachen Sinne, sind sie gerichtete Graphen, deren Knoten oder Nodes aus ihren Inputs Werte errechnen und die an die folgenden Nodes weitergeben. Hierbei werden zwischen 3 verschiedenen Arten von Nodes unterschieden:

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

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

Output Nodes 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 der 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 sehr großen Werten gegen 1 und bei sehr negativen Werten gegen 0 (logistische Funktion) oder -1 (Tangens-Hyperbolicus Funktion). Diese Funktion bietet den Vorteil das sie das Aktivitätlevel begrenzt.

Jede der Nodes hat eine bestimmte Anzahl an Verbindungen, diese hängt von der Art der Nodes und deren Zweck ab. Wichtig ist jedoch, das jede Node mit mehreren anderen Nodes verbunden ist, dies soll heißen, 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 multipliziert wird bevor es als Input weiter verwendet wird. TODO BIAS

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 gutes Modell des Problems gelernt hat. Die Stärke von Neuronalen Netzen liegt darin, aus großen Mengen von Daten Gesetzmäßigkeiten oder Patterns zu erkennen. Ein einfaches Beispiel ist die Objekterkennung. Wenn ein Netzwerk Alltagsgegenstände erkennen soll, lernt es die Pixelgruppen welche ein Tisch von einem Bett unterscheiden. Damit dies funktioniert braucht man eine große Menge an Daten. Zunächst wird das Training in zwei verschieden Arten unterteilt:

Supervised learning Innerhalb des Trainingsdatensets, hat jeder Datensatz einen vorgegeben 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.

Unupervised learning Hier hat das Trainingsdatenset keine Label. Die Gewichtsveränderungen erfolgen im Bezug zur 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.

4.2 Convolutional Neural Network - CNN

CNN sind Tiefe Neuronale Netzwerke mit einer bestimmten 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 Rechenoperationen verringert und somit das Netz effektiver macht. Um aus Bildern, Informationen zu gewinnen, müssen die Ebenen des Netzwerkes nicht vollständig verbunden sein. Stattdessen werden Filter (Convolutions) und Sub-Sampling genutzt. Filter sind kleine Matrizen, die bestimmte Features entdecken, zum Beispiel Kanten mit bestimmter Ausrichtung. Durch das Erlernen der Filter im Training kann das Netzwerk aus den Pixel Werten, schrittweise abstraktere Features errechnen. Diese gehen von einfachen Kanten, zu komplexeren Umrissen, und schließlich zu vollständigen Teil-Objekten. Zum Beispiel werden aus vielen Kanten ein Kreis, dann kommen noch mehr dazu bis eine Feautere Map ein Auge abbildet. Aus dem Auge und der biologischen Signal Verarbeitung ist diese Architektur inspiriert [?]Hubel68). Einzelne Neuronen des celebralen Kortex reagieren auf Reize nur in einem beschränkten Bereich. Da diese Bereiche leicht überlappen können so diese Neuronen den gesamten Sichtbereich erkennen.

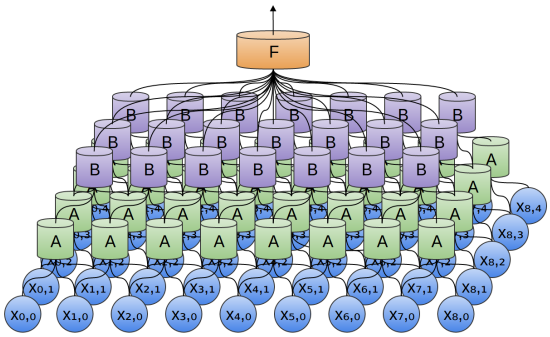

Abbildung 1: CNN von `http://colah.github.io/`

4.3 Recurrent Neural Network - RNN

RNNs sind eine Erweiterung der klassischen Feedforward Netze, die im Stande sind, Input-Sequenzen mit verschiedener Länge zu verarbeiten [8]. Die RNN schaffen dies, in dem sie einen inneren Zustand haben, dessen Aktivierungen von vorherigen Inputs abhängig sind. RNNs haben Verbindungen zu Neuronen der selben oder vorhergehenden Schichten und können dadurch Inputs durch das gesamte Netz durchgeben.

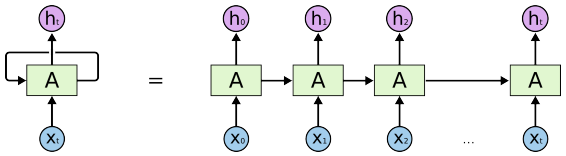

Abbildung 2: RNN von `http://colah.github.io/`

Wie auf der Abbildung zu sehen ist, ähnelt der Verbund der Rekurrenten Einheiten einer Liste. Damit ist diese Art Neuraler Netzwerke besonders gut geeignet sequentielle Daten abzuarbeiten [4]. Listen, Sprachsynthese und Maschinelle Übersetzungen sind Bereiche wofür RNNs gut geignet sind.

4.3.1 Long short-term Memory - LSTM

Ein LSTM ist eine bestimmte Form der RNNs. Ein RNN kann durch dessen starre Struktur immer nur eine bestimmte Anzahl von Schritten abspeichern. Die Schrittanzahl hängt davon ab, wie viele RNN-Units mit einander verkettet sind. Sobald mal jedoch einen sehr weiten Kontext benötigt, zum Beispiel bei der Analyse von Texten oder der maschinellen Übersetzung, stößt diese starre Struktur an ihre Grenzen. Ein LSTM ist so gebaut, dass es die einzelnen Units selber entscheiden können, welche Daten weiter beibehalten oder vergessen werden. Durch die dynamische Speicherverwaltung, kann es auf der einen Seite sich bestimmte Informationen länger speichern, auf der anderen, kurzfristig benötigte

Daten verarbeiten und dann wieder vergessen. 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 auswerten. Fast jeder Erfolg, der mit RNNs erzielt wurde, ist auf LSTMs zurückzuführen [4].

Eingeführt wurde das LSTM von Hochreiter und Schmidhuber im Jahr 1997 [10]. Seit dem wurden viele Variationen davon getestet, verworfen und verbessert. In dieser Arbeit, wird mit dem Begriff LSTM diese Original Spezifikation gemeint.

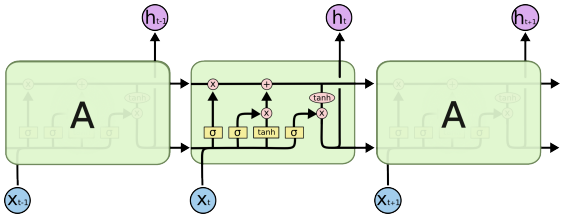

Abbildung 3: LSTM von `http://colah.github.io/`

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.

4.3.2 Gated Recurrent Unit - GRU

Die GRU, ist eine Abwandlung des klassischen LSTM. Die Architektur wurde 2014 von Kyunghyun Cho et al. eingeführt [7]. Was diese Variante ausmacht, ist eine Zussammenführung des `forget`- und `add`-Gates, sowie das Weglassen des Cell-States. Dadurch hat es weniger Parameter als die klassische Variante. Trotz der geringeren Anzahl an Para-

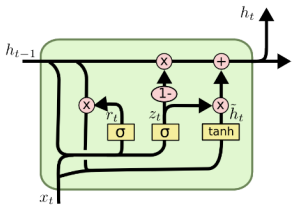

Abbildung 4: GRU von `http://colah.github.io/`

meter, ist die Performance ähnlich der klassische Variante[9], trainiert aber schneller, da weniger optimiert werden muss.

4.4 Training

Als Training wird der Prozess bezeichnet, während dem ein Neurales Netzwerk Wissen aus vorliegenden Daten extrahiert. Genau dieser Vorgang sorgt für den großen Erfolg von Neuralen 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 bestimmt wird, wie weit das genutzte Modell von der Optimalen Lösung entfernt ist. Anhand diesem Abstand können die Parameter des gewählten Modells angepasst werden um dem Optimum näher zu kommen.

4.4.1 Overfitting

Overfitting tritt ein, wenn das Modell zu Komplex für den gewählten Task ist, und deswegen die Trainingsdaten auswendig lernt. Somit wird es sehr gute Ergebnisse auf dem Trainingsset zeigen, sobald es aber Daten bekommt, die es nicht kennt, wird die Performance abstürzen da es nicht Verallgemeinern kann.

4.4.2 RMSProp

Eine Variante des Gradienten Abstiegs, die sich besonders gut eignet um RNNs zu optimieren [3].

4.4.3 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 [14].

4.5 Sampling

Während dem Sampling, wird ein fertig trainiertes Neurales Netzwerk genutzt um aus Daten Schlussfolgerungen abzuleiten. In dem Fall dieser Arbeit, wird probiert aus einem Screenshot, eine deutungsvolle DSL-Sequenz zu erstellen.

4.6 Keras

Um die in dieser Arbeit genutzten Tiefen Neuronalen Netzwerke zu definieren wir das Framework Keras genutzt. Es ist eine Library die es sehr einfach macht Neuronale Netzwerke zu erstellen und verwalten. Es selber verwaltet nur die Definition von Modellen, die eigentlichen Berechnungen werden im Backend getätigt, von Theano, Tensorflow oder CNTK. So ermöglich es schnelles experimentieren mit wenig Overhead und keinen Boilerplate-Code.

4.7 Domain-specific language - DSL

Eine Programmiersprache, die auf ein einzelne Problem-Domäne spezialisiert ist, wird DSL genannt. Im Gegensatz zur DSL steht die General Purpose Language, welches ein Programmiersprache ist, die sehr breit, für viele verschiedene Anwendungen, benutzt werden kann. Die Trennung zwischen DSL und GPL ist nicht immer klar, es kann zum Beispiel Teile einer Sprache geben die hoch spezialisiert für eine bestimmte Aufgabe sind, aber andere Teile von ihr können allgemeinere Aufgaben lösen. Auch historisch bedingt kann sich die Einordnung einer Sprache ändern. JavaScript wurde ursprünglich für ganz einfache Steuerung 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 hochspezialisierte, eigens erstellte Sprache aus Token, die eine Kombination aus HTML und CSS sind, benutzt.

4.7.1 Hypertext Markup Language - HTML

HTML ist die Standard Programmiersprache um Websites zu erstellen. Mit einzelnen HTML Elementen beschreibt es den semantischen Zusammenhang des Contents von Websites.

4.7.2 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.

5 Rapid Frontend Prototyping - Überblick

RFP ist ein Tool. welches aus einfachen GUI-Screenshots HTML/CSS Markup erzeugen kann. Dafür lernt ein System aus Neuralen Netzwerken eine einfache Token Sprache /

DSL. Jedes Wort dieser Sprache hat zugehörigen HTML/CSS Markup, wodurch es nach Erstellung der Tokens einfach in diese Sprache kompiliert werden kann.

Um den Code für die Website Screenshots zu generieren, wird ein Modell bestehend aus 2 LSTMs und einem CNN erstellt. Das hier benutzte Modell ist eine Abwandlung des im Original Paper verwendeten Modells. Da die neue DSL komplexer als die im Original Paper[6] ist, musste sowohl das CNN als auch die LSTMs verändert werden.

Von einem Computer abgespeicherte Bilder sind ein denkbar schlechtes Format um aus den Rohdaten auf Inhalte zuschließen. Zu Grunde liegen ist jedes Bild ein oder mehrere Matrizen deren einzelne Elemente einem Pixel zugeordnet werden. Um aus diesem Zahlenwerten Schlussfolgerungen zu schließen, müssen diese in mehreren Schritten abstrahiert werden um zu einem bedeutungsvollem Datenmodell zu kommen. Hierfür wird ein CNN genutzt. Dieses verarbeitet das gegebene Input-Bild und erstellt eine niedrigdimensionalere Repräsentation. Die Architektur des CNN ist recht simple und angelehnt an VGGNet von Simonyan and Zisserman.

6 Daten Synthese

Da im Zuge dieser Arbeit eine Erweiterung der DSL des Original Papers implementiert wurde, ist es erforderlich, neue Trainingsdaten zu synthetisieren. Das DataCreationTool ist ein Programm, welches nach vorgegebenen Regeln einen Token-Baum erzeugt und diesen anschließend abspeichert. Dieser Token-Baum hat immer einen body-Token als Wurzel und der gesamten Inhalt liegt als dessen Kinder vor. Dafür wurde eine Helfer Klasse geschrieben, die ein Element in dem Token Baum abbildet. Diese kann zum einen Parameter wie den Token-Name, Inhalt und Kinder speichern, zum anderen enthält sie Funktionen zum Konvertieren des Baumes zu einer String-Repräsentation sowie zum Rendering nach HTML/CSS.

6.1 Generieren der Token-Bäume

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

6.1.1 Gramatik

$$start \rightarrow [H, C] \tag{1}$$

$$H \rightarrow [Ml|Mr|S] \tag{2}$$

$$Ml \rightarrow [logoLeft, buttonWhite|logoLeft, buttonWhite, buttonWhite|...] \tag{3}$$

$$Mr \rightarrow [buttonWhite, logoRight|buttonWhite, buttonWhite, logoRight|...] \quad (4)$$

$$S \rightarrow [sidebarHeader, sidebarItem|sidebarHeader, sidebarItem, sidebarItem|...] \quad (5)$$

$$C \rightarrow [R|R, R|R, R, R] \quad (6)$$

$$R \rightarrow [S|D, D, |Q, Q, D|Q, D, Q|D, Q, Q|Q, Q, Q, Q] \quad (7)$$

$$S, D, Q \rightarrow [smallTitle, text, contentButton] \quad (8)$$

$$contentButton \rightarrow [buttonBlue, buttonGrey, buttonBlack] \quad (9)$$

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

6.1.2 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

6.1.3 Token Generierung

Die Generierung erfolgt mit einfachen, verschachtelten Loops und einem Kartesischen Produkt, um alle möglichen Kombinationen abdecken. Dadurch werden 4128 verschiedene Layout Kombinationen möglich. Die Layout Kombinationen haben eine durchschnittliche Länge von 62.47 Token (Arithmetisches Mittel), bei einen Median von 65 Token. 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, re

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 Type mit der Funktion itertool.product(). Um dies zu erreichen hätte man auch je nach Anzahl an Rows verschachtelte For-Loops benutzten können, dieser Ansatz wäre jedoch 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 jedes Elements und der Kinder in einen String zusammenführt.

6.2 DSL Mapping

Zu jedem dieser Token, existiert ein Mapping nach HTML/CSS. In einer extra 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>"
}
```

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

6.3 Screenshot Erstellung

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

6.4 Teilen der Trainingsdaten

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

7 Validierung der Ergebnisse

Um die Qualität der Ergebnisse zu vergleichen, stelle ich eine Metrik auf, die Anhand der Wichtigkeit der jeweiligen Elemente einen Score erstellt. Auf Token-Ebene einfach nach Fehlern zu suchen und jeden Falschen Token gleich zu gewichten, würde zu nicht aussage-kräftigen Scores leiten. Zum Beispiel ist ein Menue-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 eine Datei `evaluateModel.py` implementiert worden. Diese wird nach jedem Trainieren einen Score mit allen Testdaten errechnen. Dazu werden pro Datensatz mehrere Tests gemacht. Der erste Test überprüft ob der Header korrekt ist. Dieser ist einer der am stärksten gewichteten. Danach wird überprüft ob die Anzahl der Menue Item korrekt. Anschließend wird der Rest des Content überprüft, 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': './generatedMarkup/second.gui',
 'trueFileName': './generatedMarkup/SECOND_complete_generation_4_04.10.2018_1538655331945.
 'trueHeaderType': 'sidebar',
 'true_token_count': 55
}
```

Hier hat das Modell insgesamt 16 Tokens zu viel generiert, diese kommen aus zu 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 folgender Maßen ein Score berechnet:

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

cWB `countWrongButtons`: Die Anzahl Buttons mit falscher Farbe.

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

dMB `differenceMenuButtons`: Unterschied der erkannten Menü-Buttons.

$cWRT$ `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 Test Bild 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:

```
analysed_scores {
    'median': 6,
    'mean': 7.47172859450727,
    'g_mean': 6.192757554950546,
    'h_mean': 5.272351955758953,
    'quintiles': {
        'p20': 4,
        'p40': 5,
        'p60': 7,
        'p80': 9
```

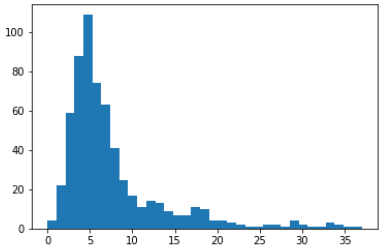

Abbildung 5: A figure

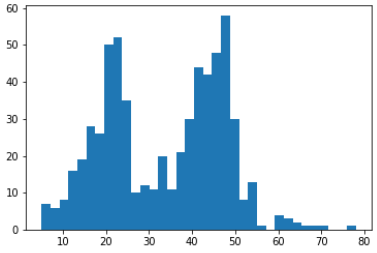

Abbildung 6: Another figure

```
},
    '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, Plot 5 und Plot 6, erkennt man, dass bei Plot 5 die Anzahl an niedrigen Scores sehr viel höher ist, als bei Plot 6.

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

```
'all': {
    'total_wrong_row_count': 99,
    'total_wrong_headers': 0,
    'total_count': 619,
    'total_wrong_row_types': 131,
    'total_wrong_menue_buttons': 244,
    'total_wrong_buttons': 2821
    },
'p80': {
    'total_wrong_row_count': 95,
    'total_wrong_headers': 0,
    'total_count': 150,
    'total_wrong_row_types': 128,
    'total_wrong_menue_buttons': 75,
    'total_wrong_buttons': 652
    }
```

8 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.

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

`model.summary()`:

```
_________________________________________________________________________________
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 bekommen.

`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:

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

`CONTEXT_LENGTH` Die `CONTEXT_LENGTH` 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.

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 dazugehörigen 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 je als pre- bzw. suffix an die Token-Sequenz angefügt bevor die n Samples gebaut werden.

So kann während dem Training mit Backpropagation ein klassischer Multi-Class-Loss optimiert werden.

$$L(I, X) = -\sum_{t=1}^{T} x_{t+1} log(y_t) \tag{11}$$

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

8.1 1. Trainingsversuch

Einführung

Um eine Baseline aller weiteren Ergebnisse zu erstellen, wird das Original Modell aus dem pix2code Paper mit meinen neuen Daten 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 mussen, 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)`. Ebense 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 das selbe 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.

8.2 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, verdoppel 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 bei ersten Versuch ergibt dieses Training wieder der gleichen Output. Egal welches Bild als Input genommen wird. 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.

8.3 3. Trainingsversuch

Einführung

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

Datenset

Gleichbleibend zum ersten Versuch.

Veränderte Parameter

Hier hab 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]
==================================================================================
Total params: 115,398,456
Trainable params: 115,398,456
Non-trainable params: 0


__________________________________________________________________________________
```

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

Ergebnis

Exakt gleiches Ergebnis zu den beiden ersten Versuchen.

Fazit

Da inzwischen immer noch der gleiche Output generiert wird, probiere ich nun einen Fehler im Programm-Code zu finden. 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 dem Preprocessing werden die Trainingsbilder in `numpy`-Arrays mit `shape(256,256,3` umgewandelt. Anschließend werden die Arrays und `gui`-Files zusammen abgespeichert.

Array Konvertierung Beweis Bild + 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 in Falle des `menue` zwei Möglichkeiten für die Positionierung des Logos gibt, wurden diese durch iteriert in der Erstellung der Daten. Da es bei der `sidebar` diese beiden Möglichkeiten aber nicht gab, wurden so 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`, in denen aber nur halb so viele einmalige enthalten sind.

8.4 4. Trainingsversuch

Einführung

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

Datenset

Ei neues Datenset mit 3096 Bildern wurde erstell. 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-titletextbtn-inactive-black
}
quadruple{
small-titletextbtn-inactive-black
}
double{
small-titletextbtn-inactive-black
}
}
row{
quadruple{
small-titletextbtn-inactive-black
}
quadruple{
small-titletextbtn-inactive-black
}
double{
small-titletextbtn-inactive-black
}
}
}
```

Fazit

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

8.5 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. bin8

Ergebnis

Es erfolgte keine Verbesserung der Trainingsergebnisse.

Fazit

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

8.6 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.

```
python train.py ../../data/rfp_data_no_body_3096_11.10.
2018/train/ ../bin9/ 1
```

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 ergibt 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 Token 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 das Bild-Teil des Modells näher in Betracht genommen werden. Ein Großteil der Parameter des Modells sind 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. Deswegen wird zunächst der Sprach- und der Decoder-Teil des Modell untersucht.

8.7 7. Trainingsversuch

Einführung

Wie im Fazit des vorangegangen Versuchs, wird nun daran gearbeit, die LSTMs, welche der sprach analysierende und der decodierende Teil des Modells sind zu verbessern. Nach der Lektüre eines Posts von Colahs Blog, wurden 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-

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 0.13

Abbildung 7: Lageparameter

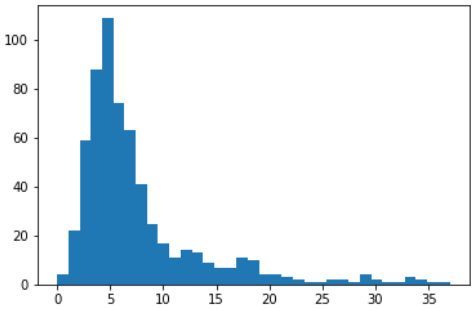

Abbildung 8: Fehlerverteilung

und Forget-Gates gekoppelt sind, hat das Modell jetzt eine Millionen Parameter weniger.

Ergebnis

Anstatt der bereits mehrmals auftretenden Token Abfolge, gelingt es diesem Modell richtige Predictions zu erstellten. Alle der 619 getesteten Files konnten ohne Fehler compiliert werden. Die Analyse der Fehlerverteilung ergibt folgende Lageparameter in Abbildung 7. Die Verteilung der Fehleranzahlen ist in Abbildung 8 gegeben.

Im weiteren wird das letzte Quintil der Fehler analysiert und mit dem Rest verglichen. In der Abbildung 9 ist die Fehlerverteilung aller Testdaten zusehen, in Abbildung 10 die besten 80% und in Abbildung 11 die schlechtesten 20%.

Bei 80% aller Testdaten sind die Fehleruhrsachen zu 92% ausschließlich falsche Buttons! Außerdem ist die Verteilung der Row Types über den Testdaten sehr genau, wie in Abbildung 12 zu sehen ist. Mit einer Perfekten Prediction müsste jeder Row Type mit $16, \frac{2}{3}\%$ vorliegen.

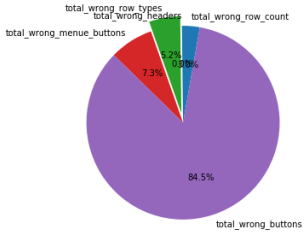

Abbildung 9: Gesamt

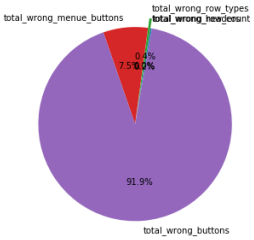

Abbildung 10: Ohne p80

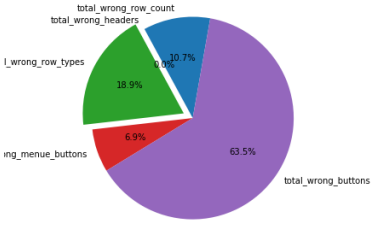

Abbildung 11: Nur p80

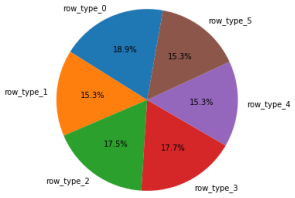

Abbildung 12: Verteilung Row Types

Fazit

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

8.8 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

Jeder 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.

8.9 9. Trainingsversuch

Einführung

Nach dem Testen ob es reicht allein den Decoder durch ein GRU zu ersetzten reicht um das Modell konvergieren zu lassen, wird nun getestet, ob es ausreicht das Sprach-Modell mit dem GRU 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, zusehen 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 das siebte Training 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 13. Die Verteilung der Fehler-Scores ist in Abbildung 14 gegeben.

Der Vergleich zu den Lageparametern 7 des siebten Experiment, performt dieses Modell sehr viel schlechter. Die Median Fehler-Score dieses Experiments ist fünfmal so hoch wie die des siebtens.

Im weiteren wird das letzte Quintil der Fehler analysiert und mit dem Rest verglichen. In der Abbildung 15 ist die Fehlerverteilung aller Testdaten zusehen, in Abbildung 16 die besten 80% und in Abbildung 17 die schlechtesten 20%.

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 18. Hier ist zusehen, dass das Modell es nicht lernen konnten die verschiedenen Typen sicher zu unterscheiden. Ein Großteil wird als Typ 4 erkannt, ca. 60%, und der Rest ist nicht regelmäßig verteilt.

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 13: Lageparameter

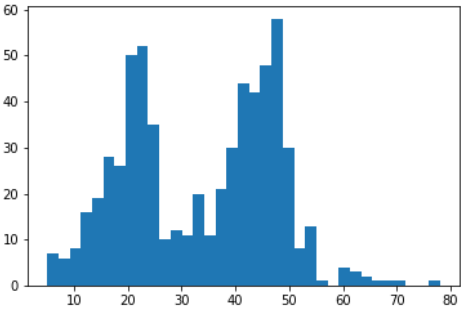

Abbildung 14: Fehlerverteilung

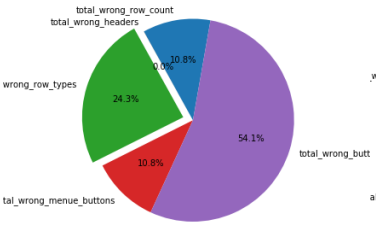

Abbildung 15: Gesamt

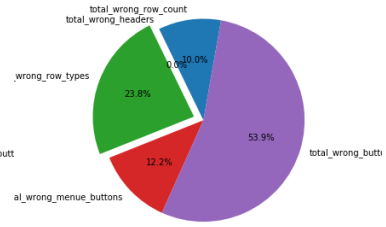

Abbildung 16: Ohne p80

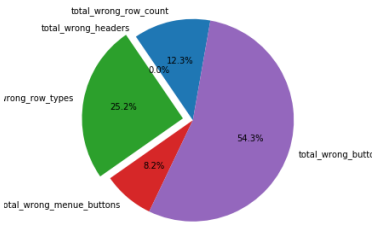

Abbildung 17: Nur p80

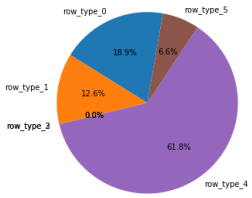

Abbildung 18: Verteilung Row Types

Fazit

8.10 10. Trainingsversuch

Einführung

Mit diesem Versuch, wird getestet, ob eine reine LSTM-Lösung mit mehr Parametern funktionieren könnte.

Datenset

Hier wurde das Datenset aus dem sechsten Trainingsversuch verwendet.

Veränderte Parameter

Um zu überprüfen, ob es reicht 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]
```

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 19: Lageparameter

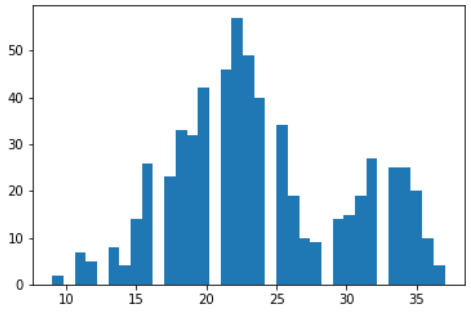

Abbildung 20: Fehlerverteilung

```
_________________________________________________________________
lstm_4 (LSTM)                (None, 512)               2099200
_________________________________________________________________
dense_3 (Dense)              (None, 23)                11799
=================================================================

Total params: 110,693,175
```

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

Ergebnis

Das Training konvergierte, erzeugt ein sehr schlechtes Ergebnis. Es konnten zwar alle 619 Testdaten kompiliert werden, die Scorings sind aber nicht Konkurrenzfähig zu den siebten Versuch - aber besser als die Ergebnisse des Neuntens.

Die Analyse der Fehlerverteilung ergibt folgende Lageparameter in Abbildung 19. Die Verteilung der Fehler-Scores ist in Abbildung 20 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

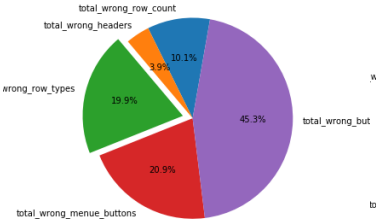

Abbildung 21: Gesamt

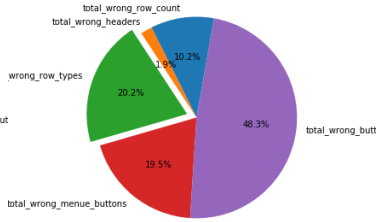

Abbildung 22: Ohne p80

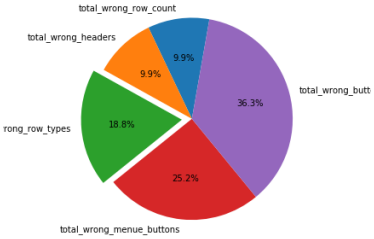

Abbildung 23: Nur p80

Quintile sind geringer. Im Vergleich zum neunten Versuch ist der Median Score um 10 geringer.

Auffallend ist jedoch in Abbildung 21, dass dieses Modell den bisher höchsten Anteil an falsch zugeordneten `header`-Elementen hat. Die bisherigen 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 in für Datei aus dem Testset immer nur einen von zwei Row Types.

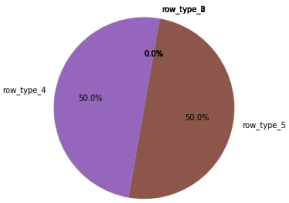

Abbildung 24: Verteilung Row Types

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.

8.11 11. Trainingsversuch

Einführung

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

Datenset

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` in der Ebene `gru_3` die Unit-Anzahl vergrößert wurde. Von 128 auf 192, bzw. von 512 auf 786.

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 25: Lageparameter

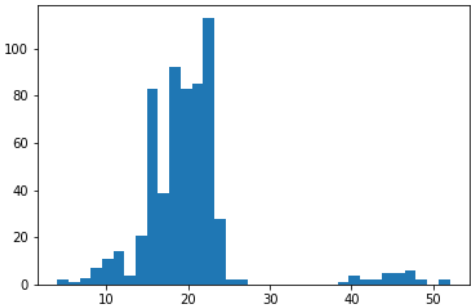

Abbildung 26: Fehlerverteilung

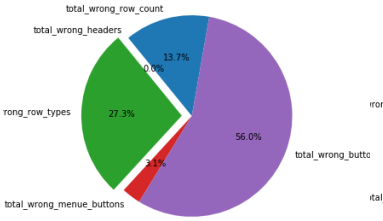

Abbildung 27: Gesamt

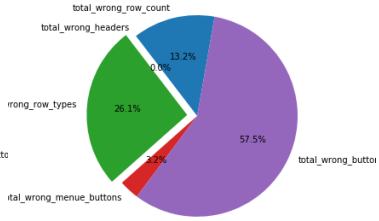

Abbildung 28: Ohne p80

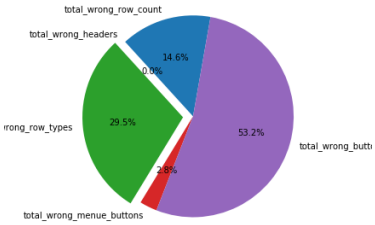

Abbildung 29: Nur p80

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 tiereren oder höheren Bereich. Zu sehen ist das sehr gut in der Abbildung 26.

In Abbildungen 27- 29 ist zu sehen, dass die Verteilung der Fehler-Type nach Fehler Anzahl relativ gleichmäßig liegt. Die Daten, welche die Meisten Fehler erzeugt haben, erzeugen die gleichen Arten wie der Gesamtmenge. Daher liegt die Fehlerursache nicht in besonders komplexen Screenshots sondern bei einer schlechten Klassifizierung.

Schlussendlich ist zu sehen, wie schlecht das Modell bei der richtigen Einordnung von den Row Types anschneidet in Abbildung 30. Es kann nicht zwischen den einzelnen Typen unterscheiden.

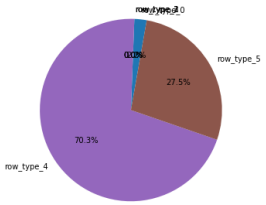

Abbildung 30: Verteilung Row Types

Fazit

Wieso performt dieses Modell, obwohl es doch größer ist so viel schlechter als Versuch Sieben? Hier 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 das Training gut aus, das Testen hingegen zeigt, dass es nicht gut verallgemeinern konnte.

8.12 12. Trainingsversuch

Einführung

Datenset

Veränderte Parameter

Ergebnis

Fazit

8.13 13. Trainingsversuch

Language: GRU 92 units Decoder GRU 386 units

bin16

Einführung

Datenset

Veränderte Parameter

Ergebnis

Fazit

8.14 Einschub - Funktioniert das original überhaupt?

Training insegesamt 20 Epochen

```
python train.py ../datasets/web/training_features/ ../b
in_orginal/ 1 ../bin_orginal/pix2code.h5
Epoch 1/10
291/290 [==============================] - 130s 445ms/step - loss: 0.2733
Epoch 2/10
291/290 [==============================] - 126s 434ms/step - loss: 0.2302
Epoch 3/10
291/290 [==============================] - 126s 431ms/step - loss: 0.2251
Epoch 4/10
291/290 [==============================] - 125s 431ms/step - loss: 0.2269
Epoch 5/10
291/290 [==============================] - 125s 431ms/step - loss: 0.2099
Epoch 6/10
291/290 [==============================] - 125s 431ms/step - loss: 0.2140
Epoch 7/10
291/290 [==============================] - 125s 431ms/step - loss: 0.1891
Epoch 8/10
291/290 [==============================] - 125s 431ms/step - loss: 0.1865
Epoch 9/10
291/290 [==============================] - 126s 432ms/step - loss: 0.1880
Epoch 10/10
291/290 [==============================] - 126s 431ms/step - loss: 0.1780
```

9 Zusammenfassung

10 Fazit

Literatur

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

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

[3] Rmsprob. https://keras.io/optimizers/#rmsprop.

[4] Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

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

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

[7] 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.

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

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

[10] 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.

[11] Shubham Khandelwal, Benjamin Lecouteux, and Laurent Besacier. COMPARING GRU AND LSTM FOR AUTOMATIC SPEECH RECOGNITION. Research report, LIG, January 2016.

[12] 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.

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

[14] 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.