

Rapid Frontend Prototyping with Deep Learning

Simon Deussen

11. Oktober 2018

Generierung von HTML/CSS Code anhand von pixelbasierten Screenshots und -designs. Aufbauend auf dem Pix2Code Paper [4], 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	2
2	Ähnliche Arbeiten	3
3	Motivation	3
4	Benutzte Technologien	3
4.1	Neuronale Netzwerke	4
4.2	Convolutional Neural Network - CNN	5
4.3	Recurrent Neural Network - RNN	5
4.3.1	Long short-term Memory - LSTM	6
4.4	Training	6
4.5	Sampling	6
4.6	Keras	6
4.7	Domain-specific language - DSL	7
4.7.1	Hypertext Markup Language - HTML	7
4.7.2	Cascading Style Sheets - CSS	7
5	Rapid Frontend Prototyping - Überblick	7

6	Daten Synthese	8
6.1	Generieren der Token-Bäume	8
6.1.1	Gramatik	8
6.1.2	Zeichenerklärung	9
6.1.3	Token Generierung	9
6.2	DSL Mapping	10
6.3	Screenshot Erstellung	11
6.4	Teilen der Trainingsdaten	11
7	Validierung der Ergebnisse	12
8	Vergleich zu dem Original	13
9	Experimente	13
9.0.1	1. Trainingsversuch	13
9.1	2. Trainingsversuch	15
9.2	3. Trainingsversuch	15
9.3	4. Trainingsversuch	17
9.4	5. Trainingsversuch	18
9.5	6. Trainingsversuch	18
9.6	7. Trainingsversuch	18
9.7	8. Trainingsversuch	19
10	Zusammenfassung	19
11	Fazit	19

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 eine enge Kooperation von Designern und Entwicklern, da hier zwei, sich kaum überschneidende, Skillsets gebraucht werden. In 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 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. 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 [4]. Er war der erste der Code anhand von visuellen Input generieren kann. Anderen Ansätze wie DeepCoder [3] benötigen komplizierte DSL als Input und schränken so die Benutzbarkeit stark ein. Visuelle Versuche mit Android-GUIs von Nguyen [6] 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 genierte 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 [7]. 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 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, 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 vorangegangenen 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 verschiedenen 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.

Unsupervised 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 Feature Map ein Auge abbildet. Aus dem Auge und der biologischen Signal Verarbeitung ist diese Architektur inspiriert [?]Hubel68). Einzelne Neuronen des cerebralen Kortex reagieren auf Reize nur in einem beschränkten Bereich. Da diese Bereiche leicht überlappen können so diese Neuronen den gesamten Sichtbereich erkennen.

4.3 Recurrent Neural Network - RNN

Als RNN bezeichnet man neuronale Netze, welches Verbindungen, im Gegensatz zu FeedForward-Netzen, zu Neuronen der selben oder vorhergehenden Schichten besitzt. Dadurch kann es zeitliche Abhängigkeiten in den Input Daten detektieren. Diese Art von

Netzen wird in der Spracherkennung, Maschinellem Übersetzung und auch Handschrifterkennung eingesetzt.

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. Zum Beispiel bei Videoanalysen jeweils die letzten 5 Frames. Das LSTM kann sich dynamisch Daten speichern, wodurch es irrelevante Daten aus vorherigen Schritten verwirft, relevantere jedoch länger abspeichert. 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.

4.4 Training

Als Training wird der Prozess bezeichnet, während 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 der bestimmt wird, wie weit das genutzte Modell von der Optimalen Lösung entfernt ist. Anhand dieses Abstands können die Parameter des gewählten Modells angepasst werden um dem Optimum näher zu kommen.

4.5 Sampling

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

4.6 Keras

Um die in dieser Arbeit genutzten Tiefen Neuronalen Netzwerke zu definieren wird das Framework Keras genutzt.

4.7 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, welches eine 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 ist, musste sowohl das CNN als auch die LSTMs verändert werden.

Das Training des Systems erfolgt über ein Sliding-Window-Prinzip, jede Token Sequenz wird mit einem Fenster einer bestimmten Context-länge n aufgeteilt, und jeder einzelne Token wird mit seinen jeweiligen n Nachbarn sowie dem Trainingsbild als Input genutzt.

Dadurch entsteht eine gewaltige Anzahl an Trainingsdaten, da jedes Trainingsbild t mal genutzt wird. t ist hierbei die Anzahl der Token für das jeweilige Bild.

Bei einer Kontext Länge von 96, gibt es insgesamt 377.457 Bild Token-Sequenz Paare. CHECK DAS AUS OB ES ÜBERAHUTP VON DER LÄNGE ABHÄNGT. Diese werden mit Mini-Batches a 64 Bild-Token-Sequenz Paare abgearbeitet.

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

$$start \rightarrow [H, C] \quad (1)$$

$$H \rightarrow [Ml|Mr|S] \quad (2)$$

$$Ml \rightarrow [logoLeft, buttonWhite|logoLeft, buttonWhite, buttonWhite|...] \quad (3)$$

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

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

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

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

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

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

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:

- MI** 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** Quadruple 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 `itertools.product()`. Um dies zu erreichen hätte man auch je nach Anzahl an Rows verschachtelte For-Loops benutzen 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\">...\n"
"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 aussagekräftigen 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.

8 Vergleich zu dem Original

9 Experimente

9.0.1 1. Trainingsversuch

Um eine Baseline aller weiteren Ergebnisse zu erstellen, wird das Original Modell aus dem pix2code Paper mit meinen neuen Daten neu trainiert. Dazu wurden alle Einstellungen gelassen wie sie waren. Da in dem benutzten Datenset eine andere Anzahl von Token ist, musste dies in 2 Layern angepasst werden:

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 256, 256, 3)	0	
input_2 (InputLayer)	(None, 48, 24)	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, 24)	12312	lstm_4[0][0]
Total params: 109,829,432			
Trainable params: 109,829,432			
Non-trainable params: 0			
CONTEXT_LENGTH 48			
IMAGE_SIZE 256			
BATCH_SIZE 64			
EPOCHS 5			

STEPS_PER_EPOCH 72000

```
input_shape (256, 256, 3)
output_size 24
input_images 377457
```

Diese Anpassung ist in in der Layer `input_2` und `dense_3` sichtbar. `input_2` ist das Language Modell und `dense_3` der Finale Output.

Leider resultierte das Training in keinen guten Ergebniss. Egal welcher Input dem Modell gegeben wurde, es erzeugt immer den gleichen Output. Nämlich diese Abfolge von Token:

```
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
}
}

```

Bei jedem Input, kommt mit ungefähr gleich hohen Wahrscheinlichkeiten immer diese Sequenz heraus. Deswegen wird angenommen, dass diese Abfolge von Token die allgemeingültigste ist. Bei diesem Output kommt der geringste Fehler heraus, obwohl sie nicht mal ein gültiges Dokument darstellt.

9.1 2. Trainingsversuch

In diesem Versuch wurde eine vergrößerte Context-Länge erprobt. Da im ersten Versuch eine 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.

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

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

9.2 3. Trainingsversuch

Da im 1. und 2. Versuch, mit der Original Architektur kein brauchbares Ergebnis heraus kommt, und das Netz ein zu allgemeines Ergebnis liefert, wird nun versucht mit einer höheren Komplexität der LSTMs ein Modell zu finden das besser funktioniert.

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) to	Output Shape	Param #	Connected
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			

```

CONTEXT_LENGTH 48
IMAGE_SIZE 256
BATCH_SIZE 64
EPOCHS 5
STEPS_PER_EPOCH 72000

```

```

input_shape (256, 256, 3)
output_size 24
input_images 377457

```

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 $n = 10$ Samples 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 Menues haben, sind doppelt enthalten. Der Grund hierfür, war die Positionierung des Menü-Logos, welches Einfluss auf die Erstellung des Menüs hatte, aber nicht auf die Sidebar. 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, obwohl sich dort nichts mehr ändert.

9.3 4. Trainingsversuch

Nach dem Finden des Fehlers nach dem dritten Versuch in den synthetisierten Trainingsdaten, wurde dieser behoben und neue Trainingsdaten generiert. Nun sind es nur noch 3096 verschiedene Bilder. Ein Drittel hat das Feature `menu_logo_left`, ein Drittel `menu_logo_right`, und der Rest die `sidebar`. In diesem Versuch werden wieder die Standard Parameter benutzt.

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
}
}
}
}

```

bin7 Nach dem Training wird das trainierte netz weiter trainiert

9.4 5. Trainingsversuch

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.

Es erfolgte keine Verbesserung der Trainingsergebnisse. bin8

9.5 6. Trainingsversuch

Update Training mit Löschung des Body Tags bin9

9.6 7. Trainingsversuch

Versuch mit GRUs [5] bin10

Using TensorFlow backend. Loading data... Generating sparse vectors... Dataset size: 256631 Vocabulary size: 23 Input shape: (256, 256, 3) Output size: 23

9.7 8. Trainingsversuch

Teste irgendwie den image input... entweder trainiere nen autoencoder oder validieren die ergebnisse irgendwie anders

10 Zusammenfassung

11 Fazit

Literatur

- [1] Electron. <https://www.electronjs.org/>.
- [2] pix2code. <https://www.github.com/tonybeltramelli/pix2code>.
- [3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.
- [4] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017.
- [5] Shubham Khandelwal, Benjamin Lecouteux, and Laurent Besacier. COMPARING GRU AND LSTM FOR AUTOMATIC SPEECH RECOGNITION. Research report, LIG, January 2016.
- [6] 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.
- [7] Günter Daniel Rey and Fabian Beck. Neuronale netze - eine einföhrung. http://www.neuronaletesnetz.de/downloads/neuronaletesnetz_de.pdf.