TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Analyse eines Forschungsthemas

Texterkennung in topographischen Karten: Untersuchungen mit dem Deep-Learning-Framework Keras in einer HPC-Systemumgebung

Jan Stephan
(Mat.-Nr.: 3755136)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Dr. Peter Winkler

Dresden, 28. September 2018

Inhaltsverzeichnis

Abkürzungsverzeichnis

CRF *conditional random field*

CTC *connectionist temporal classifier*

GPU *graphics processing unit*

HMM *hidden Markov model*

LSTM *long short-term memory*

OCR *optical character recognition*

RNN *recurrent neural network*

Abbildung 1: Kartenbeispiele

1 Einleitung

1.1 Motivation

Die Analyse historischer topographischer Karten ist eine bedeutende Aufgabe der Raum- und Landschaftsplanung, um die historische Entwicklung von Siedlungen und Naturräumen nachvollziehen zu können. Besondere Aufmerksamkeit kommt dabei der automatisierten Texterkennung zu, um digitalisiertes Kartenmaterial schnell katalogisieren, sortieren oder durchsuchen zu können.
Die kombinierte Texterkennung in und -extraktion aus Bildern ist ein Standardproblem der Bildanalyse sowohl im Bereich der klassischen *Computer Vision* (d.h. Objekterkennung durch spezielle Algorithmen) als auch des seit einigen Jahren populärer gewordenen Gebiet des maschinellen Lernens (siehe auch Abschnitt 1.2). Dieser *optical character recognition* (OCR) genannte Prozess kann in zahlreichen Anwendungsdomänen eingesetzt werden, wie etwa der automatisierten Erkennung von Kennzeichen, der Digitalisierung von Büchern oder der Informationsextraktion aus nicht digitalisierten Verträgen oder Business-Dokumenten.
Alle beispielhaft aufgeführten Fälle haben jedoch gemeinsam, dass die Texterkennung aufgrund des starken Kontrastes zwischen Text und Hintergrund und des Mangels an störenden Artefakten relativ einfach möglich ist. Die Anwendung auf historische topographische Karten gestaltet sich aufgrund einiger Faktoren vergleichsweise schwierig. So ist das Farbspektrum trotz der Fülle an Informationen in der Regel auf die Farbe des (vergilbten) Papiers und eine einzige Tintenfarbe eingeschränkt. Das führt dazu, dass der zu extrahierende Text sich farblich nicht von weiteren Markierungen auf der Karte, wie etwa Straßen, Bäumen oder Höhenzügen, unterscheiden lässt und somit zahlreiche störende Objekte bei der Erkennung ignoriert werden müssen. Darüber hinaus ist die Linienstärke der Buchstaben historischer Schriftarten stellenweise sehr dünn, sodass dickere Linien des Bildhintergrundes weitaus dominanter erscheinen als der eigentlich vordergründige Text (siehe Abbildung 1).
Diese Schwierigkeiten machen die Texterkennung in historischen topographischen Karten zu einer besonders herausfordernden und interessanten Aufgabe im Bereich des maschinellen Lernens.

1.2 Forschungsstand

Die Idee der maschinellen OCR beschäftigt Ingenieure, Informatiker und Erfinder seit mehr als einem Jahrhundert. Bereits 1914 stellte *Edmund Fournier d'Albe* ein *Optophon* vor, das nach Erkennung eines Buchstabens einen bestimmten Ton abspielte und auf diese Weise Blinden das Lesen ermöglichen sollte (vgl. [Fd14]). Wenige Jahre später meldete der in Dresden bei der Firma *Zeiss Ikon* tätige *Emanuel Goldberg* ein Patent für die von ihm entwickelte „statistische Maschine“ an, mit deren Hilfe die in Form von Buchstaben- und Ziffernkombinationen vorliegenden Metadaten auf Filmrollen durchsucht werden konnten (vgl. [Gol31]).
In jüngerer Zeit ist die OCR im Bereich des maschinellen Lernens häufiger untersucht worden. Eines der bekanntesten Projekte in diesem Bereich ist die zunächst von der Firma *Hewlett-Packard*, dann von der Firma *Google* entwickelte *Tesseract OCR Engine* (vgl. [Smi07]), die mittlerweile neben algorithmusbasierter Texterkennung den Einsatz eines neuronalen Netzes ermöglicht (vgl. [TE17]).
Jaderberg et al. präsentierten 2016 ein mehrstufiges Verfahren für die Texterkennung in Fotografien. Bei diesem Verfahren wird zunächst die mögliche Existenz eines Texts in einem Bild *entdeckt*. Alle Entdeckungen eines Bildes werden der nächsten Stufe als Vorschläge präsentiert. Diese versucht dann, innerhalb der Vorschläge Text zu *erkennen*. Grundlage des Netzes ist eine Hierarchie mehrerer mit einander verbundener *convolutional layer*, an deren Ende ein *fully connected layer* steht, welches die erkannten Buchstaben ausgibt (*aktiviert*) (vgl. [JSVZ16]).
Shi et al. stellten 2017 ein Netzwerk vor, welches ebenfalls der Texterkennung in Fotografien dienen sollte. Wie *Jaderberg et al.* setzen sie zunächst auf eine Hierarchie mehrerer verbundener *convolutional layer*, fügen jedoch vor der Buchstabenaktivierung *long short-term memory* (LSTM) *layer* ein, deren Ausgabe später durch einen handgeschriebenen Algorithmus dekodiert wird. (vgl. [SBY17])

1.3 Zielstellung

Ziel dieser Arbeit ist der Entwurf eines neuronalen Netzes, welches Texte auf topographischen Karten erkennen und extrahieren kann. Dieses Netz soll auf dem zur TU Dresden gehörigen HPC-System *Taurus* zum Einsatz kommen und die dort vorhandenen GPUs zur Beschleunigung verwenden.
Ferner soll das Skalierungsverhalten des Netzwerks untersucht werden, indem die Erkennungsrate des Netzes in Abhängigkeit von der Größe der Trainingsdaten betrachtet wird.

Abbildung 2: Beispiel für künstlich erzeugte Daten

2 Daten und Methoden

2.1 Reale und künstliche Daten

Bevor ein neuronales Netz zum Einsatz gebracht werden kann, muss es mittels realer oder künstlich erzeugter Daten, die dem realen Problem möglichst nahe kommen, *trainiert* werden. Im vorliegenden Fall der topographischen Karten wäre die Erzeugung einer ausreichend großen, das heißt dem *Overfitting* (vgl. Abschnitt 3.2) vorbeugenden, realen Datenmenge sehr aufwendig, da dies das händische Markieren und Ausschneiden tausender Beispiele erforderte. Es kommen daher für das *Training* künstlich erzeugte Daten zum Einsatz.
Innerhalb des ScaDS-Projektes lag zur Erzeugung dieser Daten bereits ein Generator vor (vgl. [Sch17]). Dieser erzeugt auf leeren Kartenhintergründen mit topographischen Informationen (Bäume, Flüsse, Höhenlinien, etc.) zufällige Buchstabenkombinationen in verschiedenen Schriftarten (siehe Abbildung 2).

2.2 Bilderkennung

Die mit dem Datengenerator erzeugten Bilder werden vor der Eingabe in das Netzwerk zunächst auf eine einheitliche Größe (Breite: Wortlänge L multipliziert mit 32, Höhe 64) skaliert. Anschließend werden sie in ein Graubild des Typs `float32` umgewandelt und auf Werte zwischen 0 und 1 normalisiert.
Das Bilderkennungsverfahren folgt dem von *Shi et al.* vorgestellten Ansatz (vgl. [SBY17]). Es wird eine Hierarchie mehrerer miteinander verbundener *convolutional layer*, *max-pooling layer*, *batch normalization layer* und schlussendlich eines bidirektionalen LSTM *layers* erzeugt, an das sich ein *fully-connected layer* für die Buchstabenaktivierung anschließt (siehe Tabelle 1).
Bis zu diesem Punkt entspricht das *Trainings*-Netzwerk dem *Inferenz*-Netzwerk.

2.3 Textklassifizierung

Im Unterschied zur Methode von *Shi et al.* kommt für die Textklassifizierung bzw. -dekodierung kein spezieller Algorithmus zum Einsatz, sondern das Verfahren des 2006 von *Graves et al.* entwickelten *connectionist temporal classifier* (CTC).

Tabelle 1: Netzwerkkonfiguration des Bilderkennungsteils

Typ	**Konfiguration**
Input	$(L * 32) * 64$ *Grey-scale*-Bilder
Convolution	64 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
MaxPooling	(2, 2)-Fenster, (2, 2)-Stride
Convolution	128 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
MaxPooling	(2, 2)-Fenster, (2, 2)-Stride
Convolution	256 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
Convolution	256 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
MaxPooling	(1, 2)-Fenster, (2, 2)-Stride
Convolution	512 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
BatchNormalization	-
Convolution	512 Filter, (3, 3)-Kernel, (1, 1)-Stride, same-Padding, relu-Aktivierung
MaxPooling	(1, 2)-Fenster, (2, 2)-Stride
Convolution	512 Filter, (2, 2)-Kernel, (1, 1)-Stride, valid-Padding, relu-Aktivierung
Reshape	-
LSTM	Bidirektional, 512 Einheiten, alle Sequenzen, Vereinigung: Summe
Dense	53 Einheiten, he_normal-Initialisierung, softmax-Aktivierung

Die CTC-Methode wurde für die Erkennung nicht-segmentierter sequentieller Daten konzipiert. In dieser Form liegen beispielsweise digitalisierte Handschriften, Sprach- oder Gestenaufnahmen vor, deren einzelne Sequenzschritte sich nicht immer eindeutig voneinander abgrenzen lassen: Wann endet bei der Aussprache eines Wortes ein Buchstabe? Wann beginnt der nächste? In Schreibschriften, die Wörter in einer einzigen Handbewegung produzieren, lassen sich Anfang und Ende konkreter Buchstaben häufig ebenfalls nur schwierig bestimmen.

Zum Zeitpunkt der Vorstellung der CTC-Methode war für diese Art von Aufgaben die Nutzung eines *hidden Markov model* (HMM) oder *conditional random field* (CRF) üblich, die jedoch erhebliches Vorwissen über das zu lösende Problem voraussetzten und die explizite Definition von Abhängigkeiten verlangten (vgl. [GFGS06], S. 369).

Die Verwendung eines *recurrent neural network* (RNN) (wie etwa eines LSTM) zu diesem Zweck erfordert – abgesehen von der Definition der Ein- und Ausgangsformate – kein weiteres Wissen über das Problem. RNN eignen sich darüber hinaus gut für die Modellierung von Sequenzen, da sie über einen internen Zustand verfügen. Die damals bekannten auf dem Gebiet des Machine Learning verwendeten Zielfunktionen (Loss-Funktionen) mussten jedoch für jeden Punkt der Sequenz bzw. Zeitschritt separat definiert werden. Dies hatte zur Folge, dass RNN nur darauf trainiert werden konnten, voneinander gänzlich unabhängige Klassifizierungen vorzunehmen. Die *Trainings*-Daten mussten daher manuell segmentiert und die Ausgabe nachbearbeitet werden (vgl. [GFGS06], S. 369).

Ein damals weit verbreiteter Ansatz bestand darin, RNN mit HMM zu kombinieren: HMM können die Sequenzen während des *Trainings* automatisch segmentieren sowie die Ausgabe der RNN verarbeiten. Dieses *hybride* System erbte jedoch die oben erwähnten Nachteile von HMM (vgl. [GFGS06], S. 369 - 370).

Dagegen ermöglicht die CTC-Methode die Zuordnung von *Labels* zu nicht-segmentierten Eingabesequenzen. Dadurch ist sie für Anwendungsfälle, die eine solche Segmentierung nicht brauchen (wie etwa der Erkennung von Handschriften, bei der die Abgrenzung zwischen Buchstaben mitunter schwierig ist)

besonders gut geeignet; jedoch ist sie für Probleme, die eine solche Segmentierung voraussetzen (z.B. die Voraussage von Proteinstrukturen) weniger geeignet. (vgl. [GFGS06])
Im vorliegenden Problem der topographischen Karten wurde die CTC-Methode vor allem deshalb ausgewählt, um dem mitunter schreibschriftartigen Charakter historischer Schriftarten zu begegnen. *Keras* selbst bietet keine Implementierung der CTC-Methode, jedoch ist sie über das *TensorFlow*-Backend nutzbar.
Ab diesem Punkt besteht ein Unterschied zwischen dem *Trainings*- und dem *Inferenz*-Netzwerk. An den in Abschnitt 2.2 beschriebenen Bildverarbeitungsteil schließt sich im *Trainings*-Netzwerk *TensorFlows* CTC-Loss-Funktion (`ctc_batch_cost`) an, die mittels eines *Keras*-Lambdas als eigene Schicht in das Netzwerk integriert werden kann; hier ersetzt sie die sonst über *Keras* spezifizierte Loss-Funktion (siehe Quelltext 2.1).
Bei der *Inferenz* ist das Vorhandensein der CTC-Loss-Funktion nicht notwendig. Stattdessen lässt sich die Ausgabe der LSTM-Schicht mit *TensorFlows* Funktion `ctc_decode` direkt dekodieren (siehe Quelltext 2.2).

```
from keras import backend as K
from keras.layers import Input, Lambda
from keras.models import Model

def ctc_lambda_func(args):
    y_pred, labels, time_steps, label_length = args
    return K.ctc_batch_cost(labels, y_pred, time_steps,
                            label_length)

# Modelldefinition - Bilderkennungsanteil
# Dann:
labels       = Input(name = "labels", shape = [word_length],
                     dtype = "float32")
time_steps   = Input(name = "time_steps", shape = [1],
                     dtype = "int64")
label_length = Input(name "label_length", shape = [1],
                     dtype = "int64")

# prediction ist der letzte Dense-Layer der Bilderkennung
loss = Lambda(ctc_lambda_func, output_shape = (1, ), name = "CTC")
             ([prediction, labels, time_steps, label_length])

# CTC-Loss mit Bilderkennung kombinieren
model = Model(inputs = [input_data, labels,
                        time_steps, label_length],
              outputs = [loss])

# Loss wird in obigem Lambda berechnet -> hier nur Dummyfunktion
model.compile(loss = {"CTC": lambda y_true, y_pred: y_pred},
              optimizer = sgd, metrics = {...})

# Model kann jetzt für Training verwendet werden
```

Quelltext 2.1: Nutzung der CTC-Loss-Funktion beim *Training*

```
import numpy as np

from keras import backend as K

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

int_to_char = dict((i, c) for i, c in enumerate(alphabet))

def ctc_decode(y_preds):
    results = []
    y_pred_len = np.ones(y_preds.shape[0]) * y_preds.shape[1]

    # Dekodierung der LSTM-Ausgabe
    # greedy = True führt eine Best-Path-Suche durch,
    # andernfalls müsste vorher ein Wörterbuch definiert werden
    decoded, _ = K.ctc_decode(y_preds, input_length = y_pred_len,
                              greedy = True)

    # tf.keras.backend.ctc_decode gibt ein Tupel von
    # einelementigen Listen zurück. Das eine Element ist der
    # kodierte Buchstabe, die Vereinigung der Listen somit das
    # kodierte Wort.
    for label in K.get_value(decoded[0]):
        result = []
        for i in label:
            if i == -1:
                continue # überspringe blanks
            # Umwandlung von labels in Buchstaben
            result.append(int_to_char[i])
        results.append(''.join(result)) # dekodierter String
    return results

# Modelldefinition - Bilderkennungsanteil; Gewichte und Bilder
# laden. Dann:

# hier noch kodiert und mit Blanks
predictions = model.predict(images)

# hier dekodierter String
decodeds = ctc_decode(predictions)
```

Quelltext 2.2: Dekodierung im Zuge der *Inferenz*

2.4 Umsetzung auf dem HPC-System *Taurus*

2.4.1 Verwendete Hardware

Sowohl das *Training* als auch die *Inferenz* wurden stets auf einem der im *Taurus* vorhandenen GPU-Knoten mit vier NVIDIA Tesla K80 und 62 GiB Arbeitsspeicher durchgeführt (Partition `gpu2`).
Der Datengenerator wurde (je nach Verfügbarkeit) auf einem der *Taurus*-CPU-Knoten betrieben, in der Regel auf einem Knoten der `sandy`-Partition mit 16 CPU-Kernen und 30 GiB Arbeitsspeicher.

2.4.2 Verwendete Module

Sowohl die *Trainings*- und *Inferenz*-Skripte als auch der Datengenerator wurden innerhalb der SCS5-Umgebung verwendet.
Die Skripte für das *Training* und die *Inferenz* erfordern neben dem *Keras*-Framework zusätzlich die Bibliotheken *NumPy* (für den Umgang mit großen Arrays) und *OpenCV* (für die Bildvorverarbeitung). Das Modulsystem des *Taurus* stellt alle genannten Abhängigkeiten bereit; zusätzlich sorgt das Laden des *Keras*-Moduls dafür, dass die *NumPy*-Bibliothek als Teil von Python automatisch mitgeladen wird. Für die Ausführung genügt deshalb das Laden der *Keras*- und *OpenCV*-Module. Zum gegenwärtigen Zeitpunkt (20. September 2018) werden *Keras* in der Version `2.2.0-foss-2018a-Python-3.6.4` und *OpenCV* in der Version `3.4.1-foss-2018a-Python-3.6.4` geladen.
Der Datengenerator bringt diverse Abhängigkeiten mit sich, die durch das *Taurus*-Modulsystem nicht vollständig abgedeckt werden können. Aus diesem Grund wurde für seinen Einsatz im Rahmen des ScaDS-Projekts eine virtuelle Python-Umgebung (*venv*) angelegt, in der die benötigten Python-Pakete vorhanden sind. Diese *venv* kam auch bei der Anfertigung dieser Arbeit zum Einsatz. Sie befindet sich auf dem *Taurus* im ScaDS-Projektverzeichnis unter dem folgenden Pfad:
`/projects/p_scads/keras/new_venv2`

3 Ergebnisse

3.1 Erkennungsrate

Für die Bestimmung der Erkennungsrate der *Inferenz* werden in den folgenden Abschnitten zwei *Scores* gebildet: der *Wordscore* W und der *Charscore* C. W ist ein binärer Wert und gibt an, ob ein ganzes Wort korrekt erkannt wurde. C errechnet sich aus der Anzahl der korrekt erkannten Buchstaben geteilt durch die Gesamtzahl der Buchstaben des Wortes. Anders ausgedrückt bedeutet $W = 1$ immer $C = 1$; $C < 1$ impliziert immer $W = 0$.
Beispiele:

- Der Text „Bramel“ wird korrekt als „Bramel“ erkannt. $W = C = 1$
- Der Text „Tannen“ wird als „Taunen“ erkannt. $W = 0, C = 0,8333$

Dieses Verfahren erlaubt eine grobe statistische Auswertung des Lernerfolgs, berücksichtigt jedoch einige Sonderfälle nicht.
Beispiel:

- Der Text „Wehdel“ wird als „NWehdel“ erkannt. $W = C = 0$

Obwohl der Text „Wehdel“ in Gänze richtig erkannt wurde, sorgt das falsch erkannte „N“ am Wortbeginn für einen *Wordscore* von 0. Da der *Charscore* nur Buchstaben an ihrem korrekten Platz berücksichtigt, diese jedoch durch das „N“ um eine Position nach rechts verschoben wurden, ist er in diesem Beispiel ebenfalls 0.

3.2 *Overfitting*

Ein generelles Problem des maschinellen Lernens im Allgemeinen sowie der CTC-Methode im Besonderen (vgl. [GFGS06], S. 376) ist die Frage, wie eine zu große Abhängigkeit des Netzwerks von den *Trainings*-Daten vermieden werden kann. Diese *Overfitting* genannte Sichtverengung des Netzwerks sorgt zwar für exzellente Ergebnisse, wenn die *Inferenz* auf den *Trainings*-Daten durchgeführt wird, verschlechtert jedoch die Erkennungsrate für unbekannte Daten.
Das im Rahmen dieser Arbeit entwickelte Netzwerk ist vor allem bei der Verwendung kleiner *Trainings*-Datensätze von *Overfitting* betroffen, wie die in der Abbildung 3 dargestellten Messergebnisse zeigen. Als *Trainings*-Grundlage dienen hier Datensätze mit 1000 bzw. 10 000 Bildern, der Loss sowie der Validierungs-Loss wurden automatisch durch *Keras* ermittelt. In beiden Fällen ist anhand des zunächst fallenden, dann steigenden Validierungs-Loss zu erkennen, dass das Netzwerk schon nach wenigen Epochen zu *Overfitting* neigt. Im weiteren Verlauf wurde daher nur mit Datensatzgrößen von mehr als 100 000 Bildern gearbeitet (siehe Abschnitt 3.3).

3.3 Trainings- und Validierungsdaten

Für das *Training* wurden insgesamt zwei verschiedene Datensätze verwendet, die mit dem Präfix `wl6` versehen sind. Die `wl6`-Datensätze werden für das Training eines Netzwerks für Texte mit fixer Wortlänge (sechs Buchstaben) verwendet und enthalten Bilder zufälliger Größe (vgl. Tabelle 2).

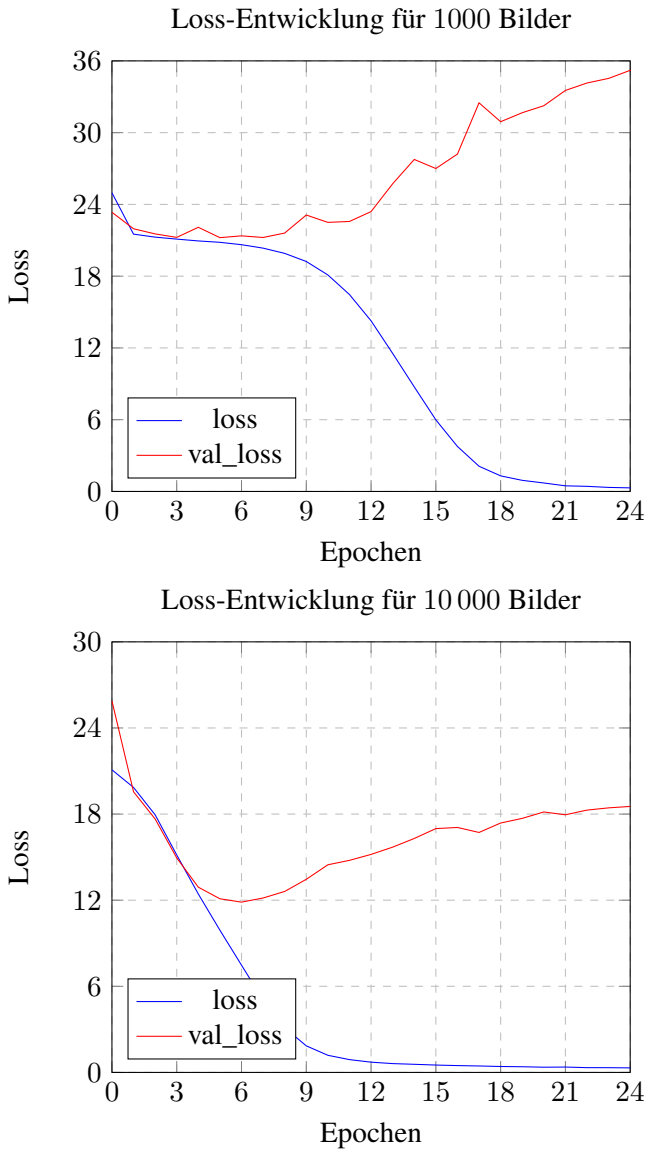

Abbildung 3: Loss und Genauigkeit für das Training mit kleinen Datensätzen

Tabelle 2: Trainingsdatensätze

Name	Bildanzahl	Textlänge	Bildgröße	Datengröße
wl6_120k	120 164	6	variabel	2,3357 GiB
wl6_250k	250 000	6	variabel	4,8841 GiB

Tabelle 3: Validierungsdatensätze

Name	Typ	Bildanzahl	Textlänge	Bildgröße	Datengröße
wl6_1000	generiert	1000	6	variabel	19,4173 MiB
wl6_real	real	27	6	variabel	549,5 KiB

Die Validierung erfolgte durch *Inferenz* in drei Schritten: zunächst auf dem Trainingsdatensatz, dann auf einem generierten, vom Trainingsdatensatz verschiedenen Validierungsdatensatz und schließlich auf realen, aus dem vorliegenden Kartenmaterial ausgeschnittenen Daten. Das Benennungsschema der künstlichen und realen Validierungsdatensätze entspricht dem der *Trainings*-Daten (vgl. Tabelle 3).

3.4 *Scores*, *Losses* und Genauigkeit

Alle der im Folgenden präsentierten Ergebnisse wurden mit Netzwerken erzielt, die auf ihren jeweiligen Datensätzen über 25 Epochen *trainiert* wurden.
Die in Tabelle 4 (visualisiert in Abbildung 4) für die fixe Wortlänge von 6 ermittelten Ergebnisse zeigen, dass das Netzwerk von einer größeren Datenmenge profitiert. Durch den Sprung vom Datensatz `wl6_120k` auf `wl6_250k` werden für alle Validierungsdatensätze deutlich bessere Ergebnisse erzielt.

3.5 Performance

Neben den erzielten Ergebnissen ist für die Nutzung auf einem HPC-System auch die Ausführungsgeschwindigkeit von Interesse. Diese soll in den folgenden Abschnitten näher betrachtet werden.

3.5.1 *Training*

Aufgrund der großen Datenmengen und der Komplexität des Netzwerks dauert das *Training* – abhängig vom konkreten Datensatz – wenige bis viele Stunden. Dabei zeigt sich, dass die Laufzeit proportional zur Größe des Datensatzes ansteigt (siehe Tabelle 5).

Tabelle 4: Scores

Trainingsdatensatz	Validierungsdatensatz	$\sum$ Wordscore / Gesamtzahl	Charscore (∅)
wl6_120k	wl6_120k	80 824/120 164	0,6819
wl6_120k	wl6_1000	661/1000	0,6685
wl6_120k	wl6_real	1/27	0,3889
wl6_250k	wl6_250k	226 432/250 000	0,9086
wl6_250k	wl6_1000	759/1000	0,8630
wl6_250k	wl6_real	4/27	0,4753

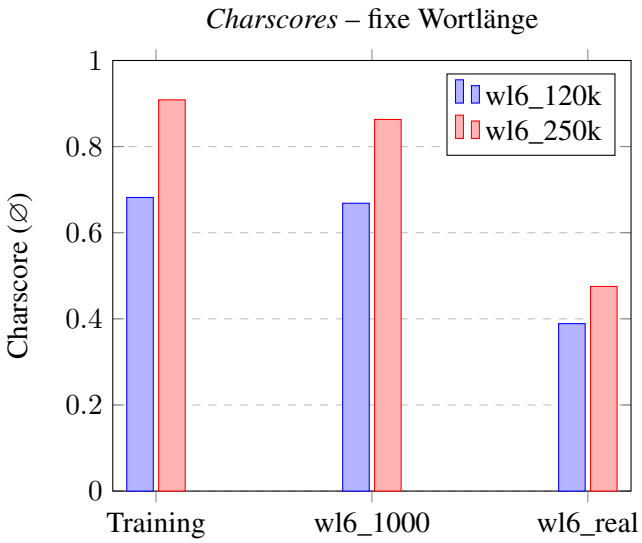

Abbildung 4: Charscores für verschiedene *Trainings*-Datensatzgrößen

Tabelle 5: *Trainings*-Laufzeiten

Trainingsdatensatz	Gesamtlaufzeit	Epochenlaufzeit
`wl6_120k`	4 h 20 min 54 s	10 min 39 s
`wl6_250k`	9 h 3 min 22 s	21 min 44 s

3.5.2 *Inferenz*

Bei der *Inferenz* zeigt sich das gleiche Bild wie beim *Training*. Auch hier steigt die Laufzeit proportional zur Datenmenge an, fällt aber durch den geringeren Rechenaufwand deutlich kleiner aus als beim *Training* (siehe Tabelle 6).

Tabelle 6: *Inferenz*-Laufzeiten

Validierungsdatensatz	Gesamtlaufzeit	Laufzeit pro Bild
`wl6_120k`	19 min 25 s	9,7 ms
`wl6_250k`	40 min 17 s	9,7 ms

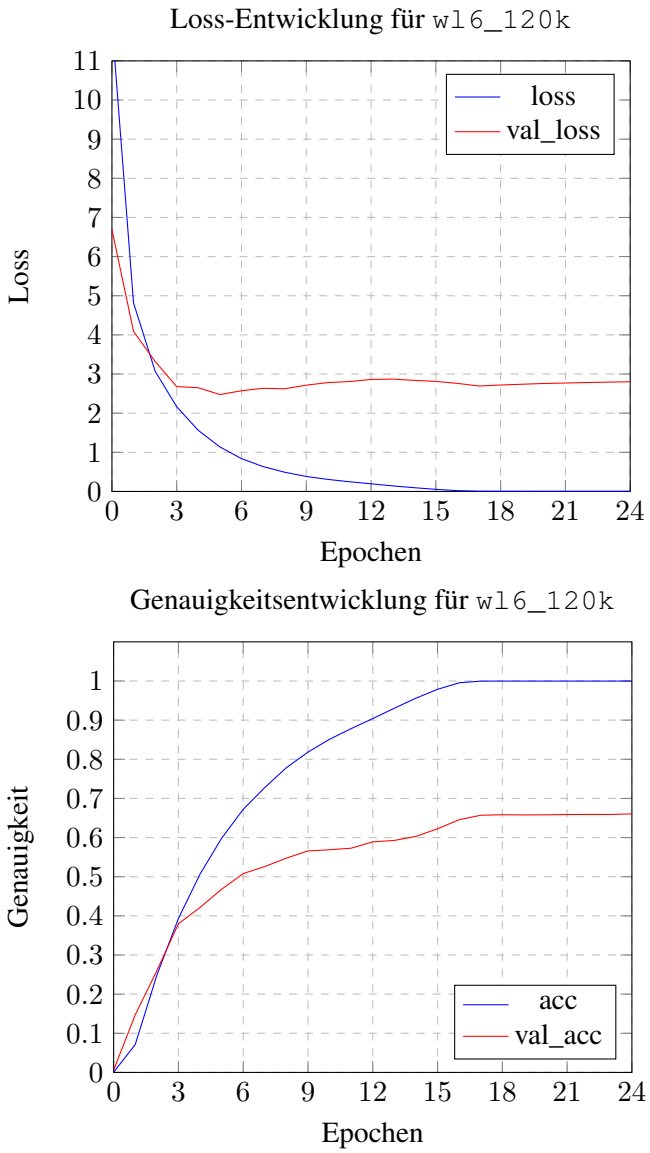

Abbildung 5: Loss und Genauigkeit für den Datensatz `wl6_120k`

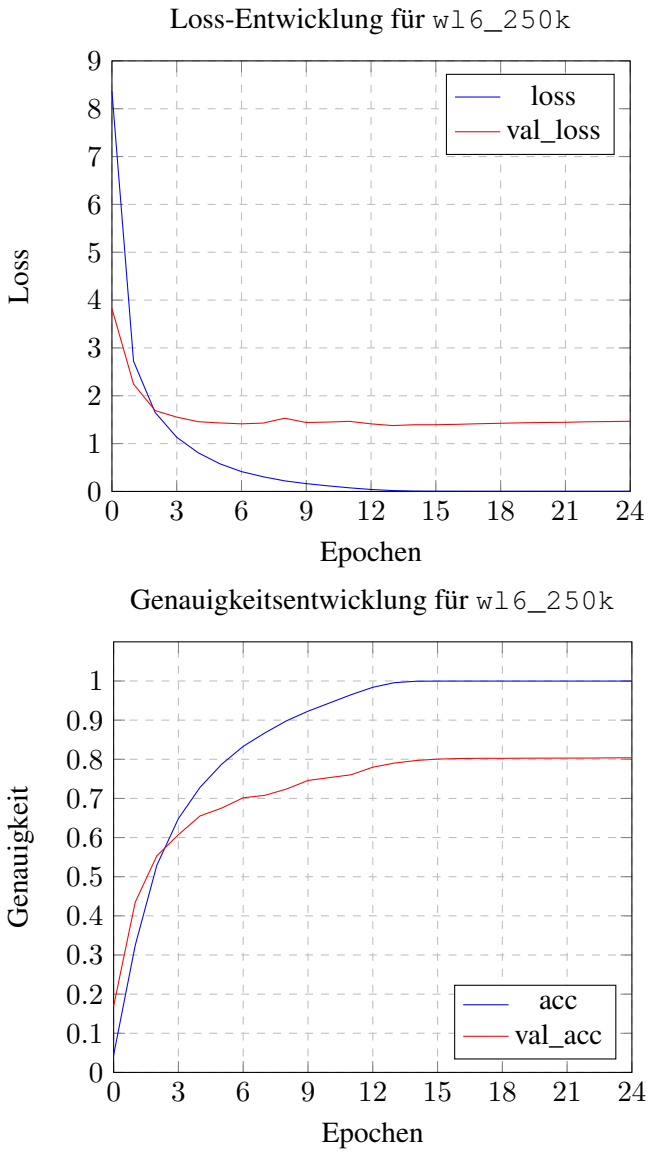

Abbildung 6: Loss und Genauigkeit für den Datensatz `wl6_250k`

4 Fazit

4.1 Entwicklungsstand

Es wurde gezeigt, dass Texterkennung in und -extraktion aus topographischen Karten mit der vorgestellten Netzwerkarchitektur möglich ist und gute Erkennungsraten liefert. Vorbedingung dafür ist das Vorliegen einer genügend großen Datenmenge für das *Training*, da das Netzwerk ansonsten zu *Overfitting* neigt. Künstliche *Trainings*-Daten sind daher für ein funktionierendes Netzwerk essentiell.
Die Nutzung der Frameworks *Keras* und *TensorFlow* ermöglichte eine einfache Umsetzung für die Nutzung der auf dem HPC-System *Taurus* vorhandenen GPUs.

4.2 Ausblick

4.2.1 Qualität der Trainingsdaten und Erkennungsraten

Insbesondere die Gewinnung den historischen Schriftbildern entsprechender *Trainings*-Daten gestaltete sich schwierig. In seiner aktuellen Fassung behilft sich der Datengenerator mit aus dem vorliegenden Kartenmaterial ausgeschnittenen Einzelbuchstaben, die dann aneinandergereiht werden; die Alternative besteht im mühevollen manuellen Ausschneiden der ganzen Wörter, wie es für diese Arbeit für die in Abschnitt 3.3 erwähnten Validierungsdaten getan wurde.
Eine Verbesserung dieser Situation sowie daraus folgend der *Inferenz*-Ergebnisse ließe sich vermutlich durch den Einsatz der Schriftarten *Kursivschrift* (vgl. [Gera]) und *Roemisch* (vgl. [Gerb]) innerhalb des Datengenerators erreichen, da sie den historisch genutzten Schriftarten weitestgehend entsprechen.
Ein weiterer Ansatz zur Verbesserung der Erkennungsrate könnte darin liegen, Vorder- und Hintergrund, das heißt Text und Karte, voneinander zu trennen und getrennt auszuwerten.
Eine genauere Untersuchung erfordert das *Training* mit Datensätzen, die größer als $250\,000$ Bilder sind. So zeigte sich bei der Verwendung eines Datensatzes mit $500\,000$ Bildern ein massiver Einbruch der Erkennungsrate, der noch unter der Rate eines Netzwerks, das mit $120\,000$ Bildern trainiert wurde, lag. Dieses Verhalten konnte bisher noch nicht erklärt werden.

4.2.2 Variable Wortlängen

Das im Rahmen dieser Arbeit entwickelte Netzwerk ist zwar auf Texte mit einer fixen Wortlänge beschränkt, eignet sich prinzipiell aber auch für die Erkennung von Texten mit variabler Wortlänge. Hierzu bedarf es einer Untersuchung der folgenden Punkte:

- Da das Netzwerk auf LSTM-Schichten basiert, ist eine vorherige Festsetzung der Dimensionen der eingehenden Bilder erforderlich. In der Variante mit fixer Wortlänge ergibt sich die Breite eines Bildes aus der Anzahl der vorhandenen Buchstaben (siehe Abschnitt 2.2). Dagegen muss bei der variablen Wortlänge eine gemeinsame Bildbreite für alle möglichen Wortlängen gefunden werden.

- Ausgehend von dem vorherigen Punkt stellt sich die Frage, in welchem Format die künstlichen *Trainings*-Daten vorliegen sollen. Ein erster Ansatz, den Generator ausschließlich Bilder mit fester Breite und Höhe, aber Texten mit verschiedener Schriftgröße generieren zu lassen und das Netzwerk auf diesen Daten zu traineren, lieferte sehr schlechte Ergebnisse ($C < 0,2$ für den Trainingsdatensatz und $C \approx 0,01$ für einen Validierungsdatensatz mit realen Daten). Eine Alternative

könnte sein, das Netzwerk ausschließlich mit künstlichen Daten der selben Wortlänge (z.B. einer vorher festgelegten Maximallänge) zu trainieren. Dieses Netzwerk lässt man dann Worte verschiedener Breite erkennen. Ein erster Versuch mit einem Netzwerk, das auf eine Wortlänge von 6 Buchstaben trainiert wurde, zeigte vielversprechende Ergebnisse für **reale** Daten mit Wortlängen zwischen 2 und 8 Buchstaben ($C \approx 0,25$). Möglicherweise ließe sich auch ein iteratives Lernverfahren umsetzen (das Netzwerk lernt erst Wörter der Länge 2, dann der Länge 3, usw. bis zum Maximum).

4.2.3 Performance-Verbesserungen

Hinsichtlich des Laufzeitverhaltens steht eine genauere Untersuchung der effizienten Nutzung mehrerer GPUs aus. Eine solche Erhöhung des Parallelisierungsgrades ist aufgrund der Dauer des *Trainings* wünschenswert und wird von *Keras* und *TensorFlow* prinzipiell unterstützt (*Keras* bietet hier das `multi_gpu_model`-API). Erste Testläufe dazu haben bereits stattgefunden, es sind jedoch für einen effizienten Multi-GPU-Ansatz weitere Arbeiten notwendig.

4.2.4 Weitere Arbeiten

Zu untersuchen wäre ferner, inwieweit sich ein Netzwerk, das bereits auf ein ähnlich gelagertes Problem angelernt wurde (wie etwa die Texterkennung in Fotografien), im Rahmen von *Transfer Learning* für die Texterkennung in topographischen Karten verwenden lässt.
Schließlich wäre eine verbesserte *Score*-Berechnung wünschenswert, die auch Sonderfälle wie die in Abschnitt 3.1 genannten berücksichtigen kann, beispielsweise mittels einer Substring-Suche.

Quelltextverzeichnis

Literatur

[Fd14] FOURNIER D'ALBE, Edmund E.: On a type-reading optophone. In: *Proceedings of the Royal Society of London* Bd. 90, 1914 (Series A: Containing Papers of a Mathematical and Physical Character), S. 373–375

[Gera] GERMAN CARTOGRAPHIC DESIGN: *Kursivschrift*. `https://www.linotype.com/de/915/kursivschrift-schriftfamilie.html`, . – Online; zuletzt abgerufen am 27. September 2018

[Gerb] GERMAN CARTOGRAPHIC DESIGN: *Roemisch*. `https://www.linotype.com/de/1410/roemisch-schriftfamilie.html`, . – Online; zuletzt abgerufen am 27. September 2018

[GFGS06] GRAVES, Alex ; FERNÁNDEZ, Santiago ; GOMEZ, Faustino ; SCHMIDHUBER, Jürgen: Connectionist Temporal Classification: Labelling Unsegmented Data with Recurrent Neural Networks. In: *Proceedings of the 23rd International Conference on Machine Learning*, 2006, S. 369–376

[Gol31] GOLDBERG, Emanuel: *Statistical Machine*. United States Patent Office, 1931. – Patentnummer 1838389

[JSVZ16] JADERBERG, Max ; SIMONYAN, Karen ; VEDALDI, Andrea ; ZISSERMAN, Andrew: Reading Text in the Wild with Convolutional Networks. In: *International Journal of Computer Vision* 116 (2016), S. 1–20

[SBY17] SHI, Baoguang ; BAI, Xiang ; YAO, Cong: An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (2017), November, S. 2298–2304

[Sch17] SCHÖLZEL, Eric: *genSet3.py*. 2017. – Unveröffentlicht

[Smi07] SMITH, Ray: An Overview of the Tesseract OCR Engine. In: *Proceedings of the Ninth International Conference on Document Analysis and Recognition* Bd. 2, 2007, S. 629–633

[TE17] TESSERACT-ENTWICKLER: *4.0 with LSTM*. `https://github.com/tesseract-ocr/tesseract/wiki/4.0-with-LSTM`, September 2017. – Online; zuletzt abgerufen am 24. September 2018

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Für die tatkräftige Hilfe bei der manuellen Extraktion der Realdaten aus dem Kartenmaterial bedanke ich mich herzlich bei Herrn Niklas Werner vom Zentrum für Informationsdienste und Hochleistungsrechnen.