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*

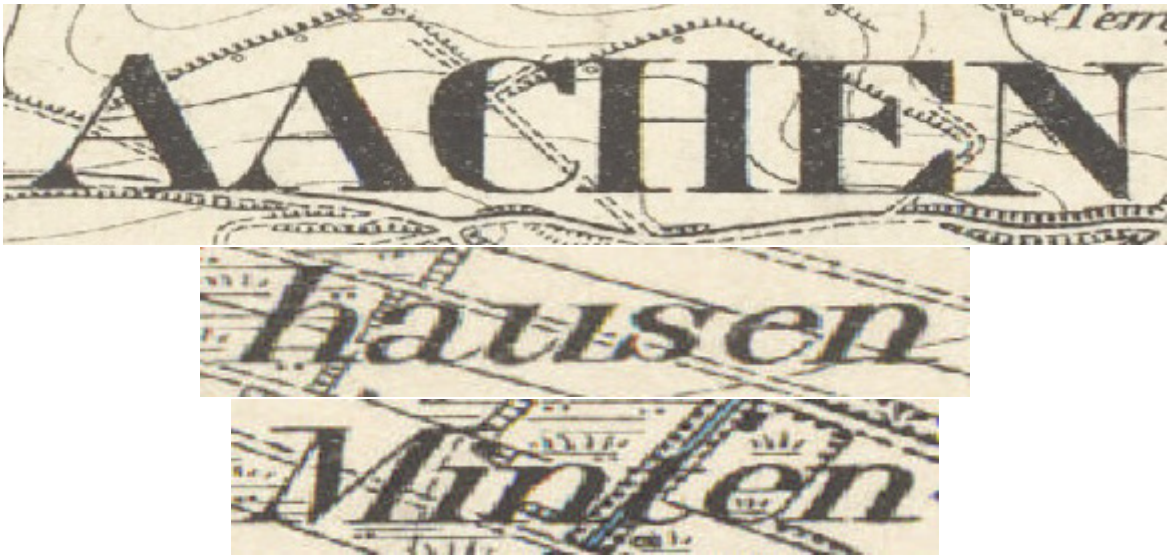

Abbildung 1: Kartenbeispiele

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.

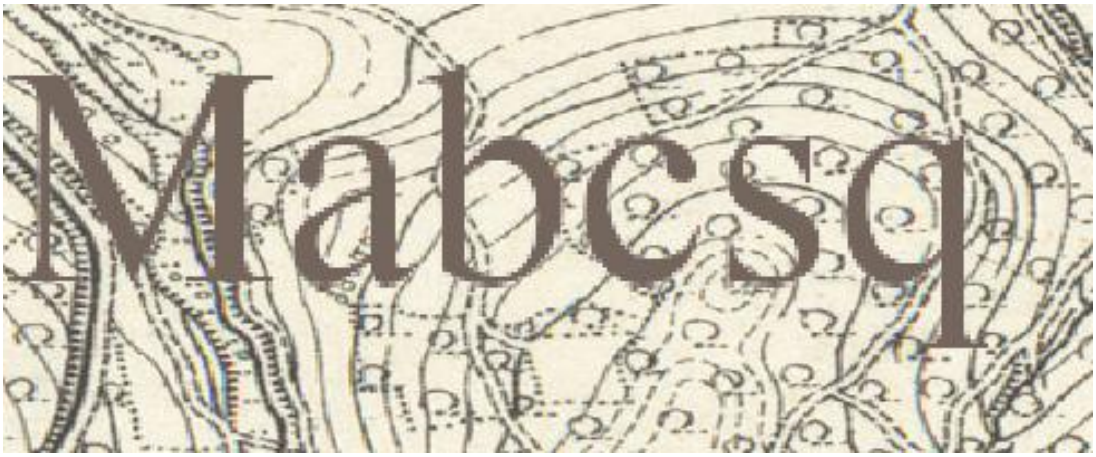

Abbildung 2: Beispiel für künstlich erzeugte Daten

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* 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 der 2006 von Graves et al. entwickelten *connectionist temporal classifier* (CTC).
Die CTC-Methode wurde für die Erkennung nicht-segmentierter sequentieller Daten konzipiert, wie sie beispielsweise bei digitalisierten Handschriften, Sprach- oder Gestenaufnahmen vorliegen. Zum damali-

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

gen Zeitpunkt war für diese 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.

Die Verwendung eines *recurrent neural network* (RNN) (wie etwa eines LSTM) zu diesem Zweck erforderte dagegen eine manuelle Segmentierung der *Trainings*-Daten sowie eine Nachbearbeitung der Ausgabe (wie beispielsweise Shi eti al. sie noch durchführen), da die einzelnen von einem RNN erzeugten Klassifizierungen voneinander unabhängig sind. Vor der Vorstellung der CTC-Methode wurden RNN daher oft in Verbindung mit HMM eingesetzt.

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 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 Fixe und variable Wortlängen

Der in den vorherigen Abschnitten vorgestellte Netzaufbau zur Bilderkennung und Textklassifizierung ist in der beschriebenen Form nur auf Bilder anwendbar, deren enthaltene Texte alle über die gleiche Anzahl an Buchstaben verfügen. Da es unter Realbedingungen äußerst unwahrscheinlich ist, nur auf Orts- und Flurbezeichnungen der selben Wortlänge zu treffen, ist die Anwendbarkeit auf unterschiedliche Wortlängen ein wichtiges Ziel. Zu diesem Zweck muss das vorgestellte Netz an einigen im Folgenden aufgeführten Punkten geändert werden:

- Die zu erkennende Wortlänge wurde auf zwei bis acht Buchstaben festgesetzt. Eine spätere Erweiterung auf längere Wörter ist prinzipiell möglich.
- Die Nutzung von LSTM-Schichten erfordert die vorherige Festsetzung der Dimensionen der eingehenden Daten. Die zu verarbeitenden Bilder werden deshalb auf eine Breite von 256 Pixeln und eine Höhe von 64 Pixeln normiert. Die Umwandlung in `float32`-Greyscale-Bilder bleibt davon unberührt.
- Die CTC-Methode benötigt keine weitere Anpassung, da diese prinzipbedingt mit Sequenzen beliebiger Länge umgehen kann.

2.5 Umsetzung auf dem HPC-System *Taurus*

Zur Ausführung der Python-Skripte für *Training* und *Inferenz* auf dem zur TU Dresden gehörigen HPC-System *Taurus* ist das Laden der im Abschnitt 2.5.2 genannten Module nötig. Die Nutzung mehrerer GPUs erfordert kleinere Anpassungen am Quelltext der Skripte, die in Abschnitt 2.5.3 aufgeführt werden.

2.5.1 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.

2.5.2 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`

2.5.3 Nutzung mehrerer GPUs

Die Dauer des *Trainings* kann – abhängig von der konkreten Netzwerkkonfiguration und der Datenmenge – mitunter sehr viel Zeit in Anspruch nehmen, bis hin zu mehreren Tagen. Es liegt deshalb nahe, den benötigten Zeitaufwand durch die Erhöhung des Parallelisierungsgrades zu verringern.
Das *Keras*-Framework unterstützt in Verbindung mit dem *TensorFlow*-Backend die parallele Nutzung mehrerer GPUs. Dazu wird das Netz zuerst im Arbeitsspeicher der CPU angelegt und dann auf die vorhandenen GPUs verteilt. Das so entstandene `multi_gpu_model` kann in der Folge wie ein normales *Keras*-Modell verwendet werden (siehe Quelltext 2.3).

```
import tensorflow as tf
# ...

from keras.utils.multi_gpu_utils import multi_gpu_model
# ...

# Keras-Beschränkung: gpu_num muss ein Vielfaches von 2 sein
if (gpu_num >= 2) and (num_gpu % 2 == 0):
    with tf.device('/cpu:0'):
        cpu_model = Model(inputs = [input_data, labels,
                                    time_steps, label_length],
                          outputs = [loss_out])
    model = multi_gpu_model(cpu_model, gpu_num)
else:
    # Standardfall: gpu_num <= 1 oder ungerade
    model = Model(inputs = [input_data, labels,
                            time_steps, label_length],
                  outputs = [loss_out])

model.compile(...)
model.fit(...)
```

Quelltext 2.3: Nutzung mehrerer GPUs mit *Keras*

3 Ergebnisse

3.1 Erfolgskriterien

Für die Erfolgsbewertung der *Inferenz* werden in den folgenden Abschnitten zwei *Scores* gebildet: der *Wordscore* und der *Charscore*. Der *Wordscore* ist ein binärer Wert und gibt an, ob ein ganzes Wort korrekt erkannt wurde. Der *Charscore* wird durch eine Zählung der korrekt erkannten Buchstaben eines Wortes ermittelt und ist ein Wert zwischen 0 und 1, wobei 1 die korrekte Erkennung aller Buchstaben an ihrem erwarteten Platz bedeutet. Anders ausgedrückt bedeutet ein *Wordscore* von 1 immer einen *Charscore* von 1; ein *Charscore* kleiner 1 impliziert immer einen *Wordscore* von 0.
Beispiele:

- Der Text „Bramel“ wird korrekt als „Bramel“ erkannt. *Wordscore* = *Charscore* = 1
- Der Text „Tannen“ wird als „Taunen“ erkannt. *Wordscore* = 0, *Charscore* = 0.8333

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

- Der Text „Wehdel“ wird als „NWehdel“ erkannt. *Wordscore* = *Charscore* = 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 Tabelle **??** aufgeführten Messergebnisse zeigen. Im weiteren Verlauf wurde daher nur mit Datensatzgrößen jenseits von 100.000 Bildern gearbeitet (siehe Abschnitt 3.3).
TODO: Ergebnisse für kleine Datensätze hier einfügen, wenn SLURM wieder funktioniert

3.3 Trainings- und Validierungsdaten

Für das *Training* wurden insgesamt sechs verschiedene Datensätze verwendet, wovon jeweils drei mit dem Präfix `wl6` und drei mit `wl28` versehen sind. Die `wl6`-Datensätze werden für das Training eines Netzwerks für Texte mit fixer Wortlänge (sechs Buchstaben) verwendet und enthalten Bilder zufälliger Größe. Die `wl28`-Datensätze sind für das Training eines Netzwerks erzeugt worden, welches Wörter mit

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
`wl6_500k`	500.000	6	variabel	9,7756 GiB
`wl28_120k`	120.164	variabel	256×64	605,5155 MiB
`wl28_250k`	TODO	variabel	256×64	0 GiB
`wl28_250k`	TODO	variabel	256×64	0 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
`wl28_1000`	generiert	1000	variabel	256×64	0 MiB
`wl28_real`	real	TODO	variabel	256×64	0 KiB

beliebiger Länge zwischen 2 und 8 Buchstaben erkennt. Die Bilder dieser Datensätze haben eine feste Größe von 256×64 Pixeln (vgl. Tabelle 2).
Die Validierung erfolgte durch *Inferenz* auf generierten und realen, aus dem vorliegenden Kartenmaterial ausgeschnittenen Daten. Diese Datensätze wurden nicht für das *Training* verwendet; ihr Benennungsschema 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 für die fixe Wortlänge ermittelten Ergebnisse zeigen, dass das Netzwerk nicht unbegrenzt von einer größeren Datenmenge profitiert. Während der Sprung vom Datensatz `wl6_120k` auf `wl6_250k` für alle Validierungsdatensätze deutlich bessere Ergebnisse liefert, verhält es sich beim Wechsel von `wl6_250k` auf `wl6_500k` genau umgekehrt; tatsächlich liefert letzteres Training die schlechtesten Ergebnisse.
TODO: Variable Länge... Trainings laufen noch
TODO: Liegt es an den Epochen? Messung läuft noch

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 sowie einiger Einschränkungen bei der Nutzung mehrerer GPUs (siehe Abschnitt 3.5.3) dauert das *Training* – abhängig vom konkreten Datensatz – mehrere Stunden bis hin zu fast einem Tag (siehe Tabelle 5).

Tabelle 4: Scores

Trainingsdatensatz	**Validierungsdatensatz**	$\sum$ **Wordscore** / Gesamtzahl	**Charscore** ($\varnothing$)
`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`	$759/1000$	0,863
`wl6_250k`	`wl6_real`	$4/27$	0,4753
`wl6_500k`	`wl6_500k`	TODO	TODO
`wl6_500k`	`wl6_1000`	$338/1001$	0,4335
`wl6_500k`	`wl6_real`	$3/27$	0,3641
`wl28_120k`	`wl28_120k`	TODO	TODO
`wl28_120k`	`wl28_1000`	TODO	TODO
`wl28_120k`	`wl28_real`	TODO	TODO
`wl28_250k`	`wl28_250k`	TODO	TODO
`wl28_250k`	`wl28_1000`	TODO	TODO
`wl28_250k`	`wl28_real`	TODO	TODO
`wl28_500k`	`wl28_500k`	TODO	TODO
`wl28_500k`	`wl28_1000`	TODO	TODO
`wl28_500k`	`wl28_real`	TODO	TODO

Tabelle 5: *Trainings*-Laufzeiten

Trainingsdatensatz	**Gesamtlaufzeit**	**Epochenlaufzeit**

3.5.2 *Inferenz*

3.5.3 Nutzung mehrerer GPUs

Der beabsichtigte Geschwindigkeitszuwachs beim *Training* durch die Nutzung der für die Verwendung mehrerer GPUs vorgesehenen *Keras*-Befehle (wie in Abschnitt 2.5.3 beschrieben) fiel deutlich kleiner aus als erwartet. So dauerte das *Training* bei der Nutzung einer einzigen GPU $669\,\mathrm{s}$ bei einer Datensatzgröße von 90123 Bildern ($7\,\mathrm{ms}$ pro Schritt). Die Dauer der Verarbeitung des selben Datensatzes mit vier GPUs verkürzte sich auf lediglich $515\,\mathrm{s}$ ($6\,\mathrm{ms}$ pro Schritt). Das entspricht einem Speedup S von $1{,}299$ sowie einer parallelen Effizienz E von $0{,}325$ und liegt somit weit unter den theoretisch erreichbaren Werten von $S = 4$ bzw. $E = 1$.
Wie existierende Benchmarks zeigen, ist dies kein inhärentes Problem von Deep-Learning-Netzwerken, der implementierten Algorithmen oder des *TensorFlow*-Backends (vgl. [Ten]). Wahrscheinlich sind daher zwei Ursachen ausschlaggebend:

1. *Keras'* `multi_gpu_model` könnte laut einiger Hinweise der Nutzergemeinschaft nicht sehr gut implementiert sein - unter anderem wird mangelnde Überlappung zwischen Speichertransfers und Berechnungen auf der GPU beklagt. Dieses Problem soll sich durch direkten Zugriff auf die *TensorFlow*-API umgehen lassen (vgl. [Zá17]).

2. *TensorFlows* CTC-Implementierung kann in Ermangelung einer entsprechenden Implementierung nicht von der Beschleunigung durch GPUs profitieren; eine zeitnahe Umsetzung ist derzeit (26.

September 2018) nicht in Sicht (vgl. [Li16]).

Der Einfluss dieser Punkte auf Speedup und parallele Effizienz bedürfen zukünftig weiterer Untersuchung.

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.
Die Nutzung der Frameworks *Keras* und *TensorFlow* ermöglichte eine einfache Umsetzung für die Nutzung der auf dem HPC-System *Taurus* vorhandenen GPUs, auch wenn die Nutzung mehrerer GPUs zur Zeit noch nicht optimal funktioniert.

4.2 Ausblick

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 **??** 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.
Hinsichtlich der Performance im Hinblick auf die GPU-Nutzung steht eine genauere Untersuchung der effizienten Nutzung mehrerer GPUs sowie der Beschleunigung durch eine künftige CTC-Implementierung für GPUs aus.
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.

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

[Li16] LI, Xibai: *CTC GPU Support*. https://github.com/tensorflow/tensorflow/issues/2146, 2016. – Online; zuletzt abgerufen am 26. September 2018

[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

[Ten] TENSORFLOW: *Benchmarks*. https://www.tensorflow.org/performance/benchmarks, . – Online; zuletzt abgerufen am 23. September 2018

[Zá17] ZÁMEČNÍK, Bohumír: *Towards Efficient Multi-GPU Training in Keras with TensorFlow*. https://medium.com/rossum/towards-efficient-multi-gpu-

training-in-keras-with-tensorflow-8a0091074fb2, Oktober 2017. – Online; zuletzt abgerufen am 23. 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.