

## **Zweites Kriterium Autoencoder und automatisches Transferlernen**

Sebastian Hoch

### **MASTERARBEIT**

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Studiengang Informatik Master

Fakultät Elektrotechnik, Medizintechnik und Informatik  
Hochschule für Technik, Wirtschaft und Medien Offenburg

2 Juni 2020

Durchgeführt bei der PSIORI GmbH

Betreuer

Prof. Dr.-Ing. Janis Keuper, Hochschule Offenburg  
Dr. rer. nat. Sascha Lange, PSIORI GmbH

**Hoch, Sebastian:**

Zweites Kriterium Autoencoder und automatisches Transferlernen / Sebastian Hoch. –  
MASTERARBEIT, Offenburg: Hochschule für Technik, Wirtschaft und Medien Offenburg,  
2020. 27 Seiten.

**Hoch, Sebastian:**

Second criterion autoencoder and automatic transfer learning / Sebastian Hoch. –  
MASTER THESIS, Offenburg: Offenburg University, 2020. 27 pages.

## **Vorwort**

Die vorliegende Masterarbeit, mit dem Titel Zweites Kriterium Autoencoder und automatisches Transferlernen, habe ich als Abschlussarbeit meines Studiums der Informatik an der Hochschule Offenburg und meines Praktikums bei der PSIORI GmbH geschrieben. Ziel war es, neue Werkzeuge zum Transferlernen zu erzeugen und zu evaluieren. Anfang bis Mitte 2020 habe ich mich intensiv mit der Entwicklung und dem Schreiben der Masterarbeit beschäftigt.

Die Idee und die Fragestellung der Abschlussarbeit habe ich zusammen mit meinem Betreuer Dr. Sascha Lange entwickelt. Durch seine Fachentnisse im Bereich der Data Science konnte ich wichtige Einblicke in die Materie gewinnen.

Während meiner Arbeiten waren meine Betreuer, Prof. Dr.-Ing. Janis Keuper und Dr. rer. nat. Sascha Lange, und der Begleiter meines Praktikums, Flemming Biegert immer erreichbar. Sie beantworteten meine entwicklungstechnischen Fragen und gaben wertvollen Input für die methodische Vorgehensweise, sodass ich meine Masterarbeit erfolgreich durchführen konnte.

Ich wünsche Ihnen viel Spaß beim Lesen dieser Arbeit.

Sebastian Hoch

Waldkirch, Juni 2020

## **Eidesstattliche Erklärung**

Hiermit versichere ich eidesstattlich, dass die vorliegende Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Die Arbeit lag in gleicher oder ähnlicher Fassung noch keiner Prüfungsbehörde vor und wurde bisher nicht veröffentlicht. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Offenburg öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Offenburg, 2 Juni 2020

Sebastian Hoch

## Zusammenfassung

### ***Zweites Kriterium Autoencoder und automatisches Transferlernen***

Im Rahmen dieser Arbeit wurden drei Werkzeuge erstellt, um Merkmalextraktion, Transferlernen und AutoMI zu kombinieren. Das erste Werkzeug gleicht eine Schwäche eines Autoencoders aus. Beim Training eines Autoencoders wird die Rekonstruktion, also der Output des Modelles und nicht direkt die Einbettung als Bewertungskriterium herangezogen. Um diese Schwäche zu kompensieren, wurde der SCAE erstellt. Dieses Werkzeug ist ein Autoencoder mit weiterem Ausgang. Die Datenrepräsentation wird durch ein zweites Kriterium gestärkt. Das zweite Werkzeug nutzt die Datenrepräsentation, um eine Aufgabe mittels Transferlernen durchzuführen. Das zweite Kriterium wird durch ein neues Kriterium ersetzt. Als drittes Werkzeug wurde der TCSCAE um Funktionen des AutoMI erweitert. Die besten Hyperparameter werden automatisch gefunden. Die Werkzeuge wurden anhand von echten Datensätzen getestet und validiert. Dabei hat sich gezeigt, dass mit den Werkzeugen eine ähnlich gute Leistung wie auf dem herkömmlichen Weg erreicht werden kann und das durch das Transferlern sogar aufwand reduziert werden kann.

Beschreibung der Tools weniger konkret?

Abstract schreiben

## Abstract

## ***Second criterion autoencoder and automatic transfer learning***

et ins Englische  
zen

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
1.1 Zielsetzung . . . . .	3
1.2 Vorgehen . . . . .	4
<b>2 Grundlagen</b>	<b>5</b>
2.1 Stacked Convolutional Autoencoder . . . . .	5
2.1.1 Layerwise Pretrain . . . . .	5
2.2 Transferlernen . . . . .	5
2.3 Automatisiertes maschinelles Lernen . . . . .	6
2.3.1 Hyperparameter-Optimierung . . . . .	6
2.3.2 Meta-Learning . . . . .	6
2.3.3 Neural Architecture Search . . . . .	6
2.3.4 Gittersuche . . . . .	7
2.3.5 Zufallssuche . . . . .	7
2.3.6 Bayesian optimization . . . . .	7
2.3.7 HyperBand . . . . .	7
2.3.8 BOHB . . . . .	7
2.4 Bibliotheken und Werkzeuge . . . . .	7
2.5 Einordnung und bestehende Systeme . . . . .	10
2.6 Datenverständnis . . . . .	12
2.7 Datenvorbereitung . . . . .	14
<b>3 Werkzeuge</b>	<b>17</b>
3.1 ConvolutionalSecondCriterionAutoenocder . . . . .	17
3.2 TransferSecondCriterionAutoenocder . . . . .	19
3.3 AutoTransferSecondCriterionAutoenocder . . . . .	21
<b>4 Experimente</b>	<b>25</b>
4.1 Basislinie . . . . .	25
4.2 ConvolutionalSecondCriterionAutoenocder . . . . .	25
4.3 TransferSecondCriterionAutoenocder . . . . .	25
4.4 AutoTransferSecondCriterionAutoenocder . . . . .	26

## Inhaltsverzeichnis

---

<b>5 Fazit</b>	<b>27</b>
5.1 Zusammenfassung . . . . .	27
5.2 Kritische Reflexion . . . . .	27
5.3 Ausblick . . . . .	27
<b>Abkürzungsverzeichnis</b>	<b>i</b>
<b>Tabellenverzeichnis</b>	<b>iii</b>
<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Quellcodeverzeichnis</b>	<b>vii</b>
<b>Literatur</b>	<b>ix</b>

# **Todo list**

Abstract schreiben . . . . .	iii
Abstract ins Englische übersetzen . . . . .	iv
Einleitung schreiben . . . . .	3
Zielsetzung schreiben . . . . .	3
Vorgehen schreiben, CRISP-DM + Werkezugerstellung . . . . .	4
Quellen . . . . .	5
sec:StackedConvolutionalAutoencoder . . . . .	5
sec:Transferlernen . . . . .	5
Quelle Autoencoder pretrain . . . . .	9
Quelle fit Keras . . . . .	9
weight durch Verhältnis w1/w2 anpassen. . . . .	22
Worker Budget berücksichtigen / notieren . . . . .	22



# 1. Einleitung

Einleitung schre

Das Finden geeigneter Datenrepräsentationen ist ein bekanntes Problem im Feld der DataScience. Dabei ist bekannt, dass Datenrepräsentation maßgeblich für die Leistungsfähigkeit von maschinellem Lernen verantwortlich ist. Insbesondere hochdimensionale Daten, wie Bilder, haben mit dem Fluch der Dimensionalität[] zu kämpfen. Der Einsatz von Autoencoder[] erlaubt es komprimierte Datenrepräsentationen für Bilder zu finden. Sind dabei ein unüberwachtes Lernverfahren, brauchen also keine beschrifteten Daten. In vielen Anwendungsfällen ist es teuer oder schwierig beschriftete Daten für neue Anwendungsfälle zu beschaffen. Transferlernen adressiert diese Problematik. Es hat das Ziel, gute Modelle in einer neuen Domäne basierend von Wissen in einer anderen Domäne zu erstellen.

Die PSIORI GmbH [PS20] ist ein Unternehmen, welches Projekte im Bereich der künstlichen Intelligenz für Kunden durchführt. PSIORI kann dabei auf einen mehrjährigen Erfahrungsschatz im Bereich des maschinellem Lernen zurückgreifen und setzt dabei häufig Autoencoder und das Transferlernen zum Lösen von Aufgabenstellungen ein. Oft ist der Ansatz des Transferlernenes kostengünstiger um ein neues Modell in einer bestehenden Domäne zu erstellen.

(AutoMI auch noch Motivieren)

## 1.1 Zielsetzung

Zielsetzung schre

Ziel dieser Arbeit ist es, Werkzeuge zur Kombination der Ansatzes des Autoencoders und des Transferlernenes zu erstellen und an Hand von einem echten Datensatz zu evaluieren.

**SecondCriterionAutoencoder** Der SecondCriterionAutoencoder (SCAE) ist ein Werkzeug welches beim Erstellen eines Autoencoder mit weiterer Verlustfunktion unterstützt.

**TransferSecondCriterionAutoencoder** Der TransferSecondCriterionAutoencoder (TSCAE) ersetzt die Verlustfunktion eines SCAE.

**AutoTransferSecondCriterionAutoencoder** Der AutoTransferSecondCriterionAutoencoder (AutoTSCAE) erweitert den TSCAE um eine automatische Hyperparametersuche.

## 1.2 Vorgehen

Werkzeuge erstellen Mnist + Emnist 1. 2.

Modelle Iterativ verbessern 1. fit 2. fitgenerator 3. 80.000 3. AutoML

[MNIST] [EMNIST]

## 2. Grundlagen

Quellen

### 2.1 Stacked Convolutional Autoencoder

CNN [Le99] CAE [Ma11]

sec:StackedConv

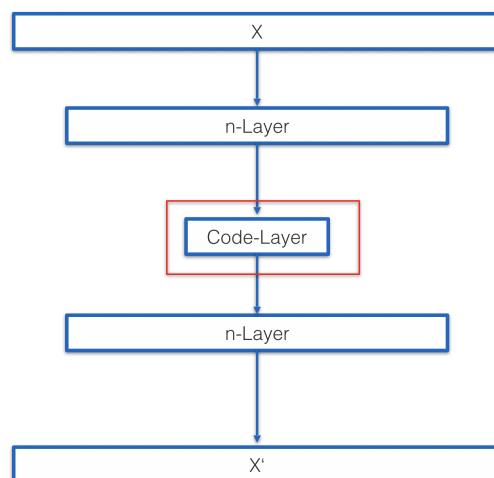


Abbildung 2.1: Schema Autoencoder

#### 2.1.1 Layerwise Pretrain

[Be07]

## 2.2 Transferlernen

sec:Transferlerne

- Was ist das - oberste Schichten allgemeiner als untere → todo papaer verweisen s

## 2.3 Automatisiertes maschinelles Lernen

Das automatisierte maschinelles Lernen kurz AutoML<sup>1</sup> hat das Ziel alle Aspekte des maschinellen Lernens und der Datenanalyse-Pipeline zu automatisieren. Die vollständige Automatisierung erlaubt es auch Nutzern ohne oder mit geringen Kenntnissen von ML-Techniken die Erstellung von ML-Systemen. Die volle Automatisierung ist ein langfristiges Ziel. Aktuelle Systeme sind halbautomatisch und zielen darauf ab, Personenaufwände bei Bedarf nach und nach durch Rechenvorgänge zu reduzieren. Trotz der steigenden Rechenleistung, können sind AutoML-Methoden sehr rechenintensiv. AutoML wird in drei Methoden eingeordnet. Das Meta-Learning, Neural Architecture Search kurz NAS<sup>2</sup> und Hyperparameter-Optimierung kurz HPO<sup>3</sup>. [HKV19] s

### 2.3.1 Hyperparameter-Optimierung

Hyperparameter-Optimierung oder Hyperparameter-Tuning wird in jedem HPO-Algorithmus zur Steuerung des Trainings eingesetzt.

[KJ]s

[FH19]

Datenänderungen (führen zu es funktioniert nicht) Kohavi, R., John, G.: Automatic Parameter Selection by Minimizing Estimated Error. In: Prieditis, A., Russell, S. (eds.) Proceedings of the Twelfth International Conference on Machine Learning, pp. 304–312. Morgan Kaufmann Publishers (1995)

### 2.3.2 Meta-Learning

[Jo18] [Va19]

### 2.3.3 Neural Architecture Search

[EMH19]

---

<sup>1</sup>automatisiertes maschinelles Lernen

<sup>2</sup>Neural Architecture Search

<sup>3</sup>Hyperparameter-Optimierung

Die AutoML-Methoden nutzen dabei Optimierungsmethoden wie Gittersuche, Zufallssuche, Bayesian optimization, HyperBand und Mischungen davon wie BOHB.

#### **2.3.4 Gittersuche**

[Mi18]

#### **2.3.5 Zufallssuche**

[Be12] [Mi18]

#### **2.3.6 Bayesian optimization**

[Mi18]

#### **2.3.7 HyperBand**

[Li17]

#### **2.3.8 BOHB**

[SAF18]

### **2.4 Bibliotheken und Werkzeuge**

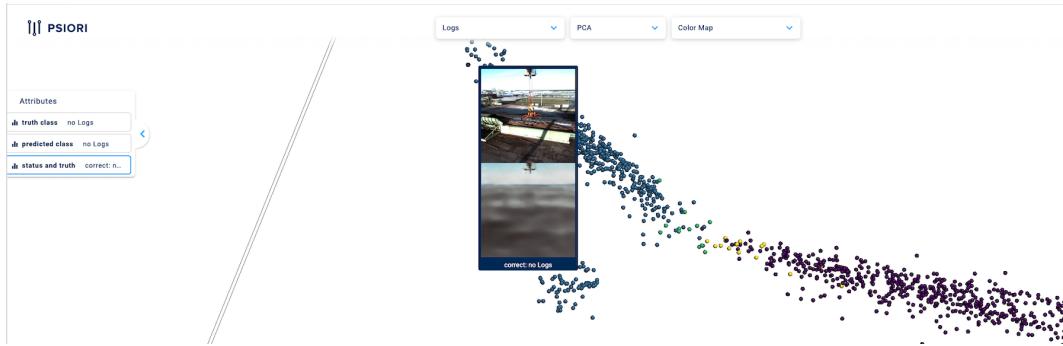
Für den praktischen Teil der Abschlussarbeit wurde insbesondere Cnvrge [cn] genutzt. Cnvrge.io ist eine "full-stack Data Science Platform" welche Werkzeuge für die Erstellung, Verwaltung, Bereitstellung und Automatisierung von maschinellem Lernen bereitstellt. Cnvrge erlaubt es Arbeitsbereiche mittels Containern zu erstellen. Die Container können dabei auf Maschinen in Azure [Mi20] zugreifen. Für die Experimente wurde ein vorgefertigter Container mit einer tesla-k80 [Nv20], fünf CPUs und 49 GB Arbeitsspeicher genutzt.

## 2 Grundlagen

---

Für die Entwicklung wurden Python [Py20], Jupyter Notebooks [**ProjectJupyter**] und das Framework Tensorflow [Ma15] genutzt. Die wichtigsten Bibliotheken für die Arbeit sind Keras [Ch15] , Numpy [Ol06] , Matplotlib [Hu07] , scikit-learn [Pe11] , ConfigSpace [**Lindauer.8162019**] , Bayesian Optimization and Hyperband [SAF18].

Für die Visualisierung von Bildeinbettung wurde das Werkzeug "PSIORI Visualizererweitert und eingesetzt. Der Visualizer erlaubt es Daten in 3-D darzustellen. Dabei können Filter eingesetzt werden, der Blickwinkel geändert werden, die Daten mit zusätzlicher Information versehen werden und Zoomen ist möglich.



**Abbildung 2.2:** Beispiel PSIORI Visualizer

Die Abbildung 2.2 zeigt einen Screenshot einer Visualisierung einer Einbettung. Als zusätzliche Information sind die Datenpunkte entsprechend einer Klassifikation in True-Positiv, True-Negativ, False-Negativ und False-Positiv eingefärbt. Wird über einen Datenpunkt mit der Maus geschwebt, werden zusätzliche Informationen zu dem Datenpunkt angezeigt. Diese Funktion wurde insbesondere zum Anzeigen eines Originalbildes, ihrer Rekonstruktion mittels Autoencoder und einer Beschriftung genutzt.

Kern der erstellten Werkzeuge ist das Framework Psipy [PS19]. Psipy ist ein Python-Framework für maschinelles Lernen welches von PSIORI selbst entwickelte Modelle zusammenfasst und eine einheitliche API zu Verfügung stellt. Diese API ist an die API des verbreiteten Frameworks scikit-learn angelehnt. Es können Modelle basierend auf scikit-learn und Tensorflow eingebunden werden. In den nachfolgenden Abschnitten werden die, für die Arbeit, wichtigsten bestehenden Module des Frameworks vorgestellt.

**saveable.py** Das Modul Saveable ist eine flexible Basisklasse die Kernfunktionalität zum Speichern und Laden von Python-Objekten bietet. Es können Modelle,

welche diverse Bibliotheken nutzen auf eine einheitliche Art und Weise gespeichert werden. Um die Klasse Saveable nutzen zu können, müssen erbende Klassen ihre Konstruktorargumente an die Basisklasse übergeben. Zusätzlich ist es notwendig eine Erweiterung beim Speichern und Laden zu implementieren. Beim Speichern ist es notwendig eine Erweiterung, um alle (meist ein) Module und weitere Argumente zu implementieren. Beim Laden müssen die gespeicherten Module und Argumente geladen werden. In Listing 2.1 ist die Erweiterung zum Speichern eines Tensorflow-Models abgebildet.

```

1     ...
2     zip_file.add("model.h5", self.model)
3     ...

```

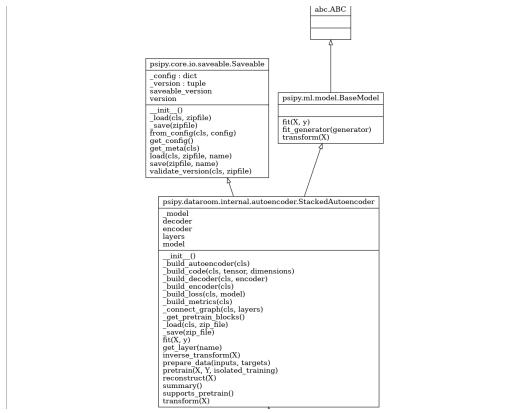
**Listing 2.1:** Erweiterung zum Speichern eines Tensorflow Models

**autoencoder.py** Das Modul Autoencoder enthält die drei Klassen StackedAutoencoder, FullyConnectedAutoencoder und ConvolutionalAutoencoder. Der StackedAutoencoder wird als Basisklasse für die anderen beiden Klassen genutzt. Im Konstruktor werden Methoden aufgerufen, welche in den abgeleiteten Klassen ausprogrammiert sind. Dabei wird ein Keras-Modell für einen Encoder und Decoder entsprechend von Parametern erstellt. Als weitere wichtige Methoden gibt es die Methode *pretrain(..)* und *fit(..)*. Mittels *pretrain(..)* werden die Schichten eines symmetrischen Autoencoders von außen nach innen wie in Greedy Layer-Wise Training of Deep Networks vortrainiert. Die Auswahl der Schichten erfolgt wieder in den abgeleiteten Klassen. In der *fit(..)*-Methode wird nach einigen Prüfungen die Methode *ffit(..)* [fit Keras] des Kerasmodels aufgerufen. In Abbildung 2.3 ist das Klassendiagramm mit den öffentlichen Methoden des ConvolutionalAutoencoder dargestellt.

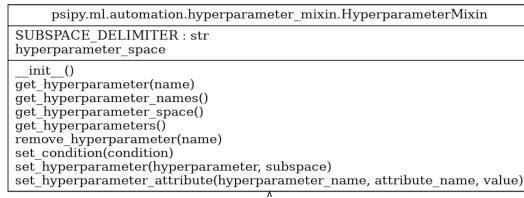
Quelle Autoencoder  
pretrain

Quelle fit Keras

**hyperparameter\_mixin.py** Hyperparameter\_mixin wird zum standardisierten Verwalten von Hyperparametern für AutoML-Klassen genutzt. Auf die Hyperparameter kann anschließend einheitlich zugegriffen werden. Abbildung 2.4 Zeigt das zugehörige UML-Klassendiagramm mit den Methoden zum Hinzufügen, Löschen und Laden der Hyperparameter. Da die Methoden öffentlich sind, können über jede erbende Klasse die Hyperparameter eigenständig verwaltet werden.



**Abbildung 2.3:** Klassendiagramm ConvolutionalAutoencoder



**Abbildung 2.4:** Klassendiagramm Hyperparamettermixin

## 2.5 Einordnung und bestehende Systeme

Die Bilddaten und Aufgabenstellungen der neuronalen Netzwerke sind in die Problemstellungen des Autocrane-Projekts von PSIORI einzuordnen. Es sind also echte Datensätze und echte Problemstellungen, wobei die gezeigten Aufgabenstellungen und Modelle nicht zwingend in dem Autocrane-Projekt zum Einsatz kommen. Das Autocrane-Projekt ist ein laufendes Projekt, welches das Ziel hat, einen feststehenden Rundlaufkran vollautomatischen zu steuern. In Abbildung 2.5 ist ein Rundlaufkran abgebildet. Diese Art von Kran werden in holzverarbeitenden Anlagen zum Befüllen von Fülltrichtern oder Förderbändern eingesetzt. Der Kran kann sich um 360 Grad drehen. Der Greifer kann nach oben, unten und entlang mittels eines Schlittens entlang Auslegers bewegt werden. Um die Bilder aufnehmen zu können, wurde an der Kabine am Hauptstandfuß eine Kamera angebracht. Die Kamera ist auf das Ende des Auslegers und den Bereich darunter ausgerichtet. Die Kamera bewegt sich also mit dem Rundlaufkran und somit ist der Greifer immer im Bild. Für das Autocarne-Projekt sind insbesondere drei Anwendungsfälle interessant. Die Baumstämme werden mittels LKW angeliefert und müssen nach vorgegebenen Regeln (z. B. Ausrichtung, freier Lagerplatz) als Holzstapel gelagert werden. Der Fülltrichter muss mit Holz aus den Holzstapeln gefüllt werden. Der Fülltrichter muss mit Holz aus einem LKW gefüllt werden. Es ergeben sich also Aufgabenstellungen

wie Greifer-Erkennung, Baumstamm-Erkennung, LKW-Erkennung, Strategien für das entladen und aufbewahren der Baumstämme und vieles mehr. Im Normalbetrieb werden täglich 140-200 LKW entladen. Die Ladung ist 9 - 18 Meter lang und 34 - 40 Tonnen schwer. [PS20]



**Abbildung 2.5:** Rundlaufkran (Foto: ANDRITZ)

**Greifererkennung** Bei der Aufgabenstellung Greifer-Erkennung muss in einem Bild die Position eines Rahmens um den Greifer gefunden werden. Abbildung 2.6a zeigt ein Bild mit Rahmens um den Greifer. Es handelt sich um eine klassische 'Object-Detection' Aufgabe.

PSIORI hat die Aufgabe mittels neuronalem Netzwerk gelöst. Dabei wurde auf die Technik des Single Shot MultiBox Detector (SSD) [todo <https://arxiv.org/abs/1512.02325>] zurückgegriffen. Das Netz liegt als frozen\_inference\_graph.pb vor. Protocoll Buffer [<https://developers.google.com/protocol-buffers/>] ist ein sprachneutraler, plattformneutraler, erweiterbarer Mechanismus zur Serialisierung strukturierter Daten. In diesem Fall enthält die Datei den eingefrorenen Graph und die Model Gewichte. Um Vorhersagen treffen zu können, wurde eine Klasse erstellt, welche mittels einer Tensorflowsession und dem Model vorhersagen trifft. abei liefert das Model pro Bild einen Rahmen, in welchem sich der Greifer befindet und einen Vertrauenswert. Der Vertrauenswert sagt aus, wie sicher sich das Netzt mit seiner Aussage ist. Die Vorhersagegenauigkeit dieses Modells wird als Basislinie und Vergleichswert für die durchgeföhrten Versuche genutzt.

**Baumstammklassifikation** Die Aufgabe der Baumstammklassifikation hat zum Ziel, zu erkennen ob sich Baumstämme im Greifer befindet oder nicht. Es handelt



(a) Greifer mit Rahmen

(b) Greifer mit Baumstämmen

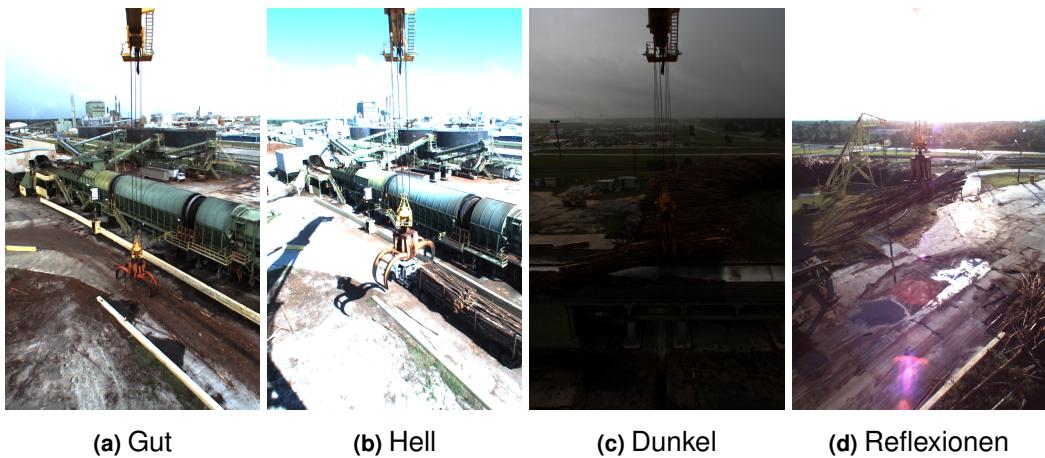
**Abbildung 2.6:** Greifer

sich um eine Klassifikationsaufgabe. Ein Klassifikator, für die Frage ob sich Baumstämme in dem Greifer befinden liegt als Tensorflow-MetaGraph[[https://www.tensorflow.org/api\\_docs/python](https://www.tensorflow.org/api_docs/python)] vor. Um Vorhersagen treffen zu können, wurde eine Klasse, welche den MetaGraph nutzt erstellt. Die Vorhersage des Modells liefert die vorhergesagte Klasse mit einer Wahrscheinlichkeit zurück. Dieses Model wird wie das Greifererkennung Model als Basislinie und Vergleichswert für die durchgeführten Versuche genutzt.

## 2.6 Datenverständnis

Mittels Kamera an der Kabine können neue unbeschriftete Bilder aufgenommen und bei PSIORI abgelegt werden. Durch den Aufbau des Rundkrans und der Kameraposition befindet sich der Greifer immer im Bild. Der Hintergrund der Bilder ändert sich stark. Die geografische Lage des Rundkrans ist im Hinblick auf das Wetter positiv. Es fällt kein Schnee und es gibt wenige Regentage. In Abbildung

2.7 sind Ausprägungen der Bildqualität dargestellt. Abgesehen von Bildern in guter Qualität gibt es helle Bilder, dunkle Bilder und Bilder mit Reflexionen. Die Bilder sind 1024 auf 648 Pixel groß und in Farbe. Die einzelnen Pixel können dabei Werte zwischen 0 und 255 annehmen. Entsprechend der beiden Aufgabenstellungen Greifererkennung und Baumstammklassifikation werden Daten mit einer passenden Beschriftung benötigt.



**Abbildung 2.7:** Bildqualität

**Greiferdatensatz** Der Greifer Datensatz enthält Bilder, in welchen der Greifer mittels Rahmen markiert ist. Abbildung 2.6a zeigt ein beispielhaftes Bild mit markiertem Greifer. Die Annotationen der Position des Greifers wurde pro Bild in einer XML-Datei abgelegt. Konkret wurde die Position als x, y Koordinaten in der Form ymin, xmin, ymax und xmax abgespeichert. Über die vier Werte lässt sich problemlos ein Rahmen um den Greifer spannen. Der Datensatz besteht aus 4.684 durch Personen annotierten Bildern.

**Baumstammdatensatz** Der Baumstamm Datensatz enthält Bilder, welche die Annotation, ob sich Baumstämme im Greifer befinden oder nicht enthält. Die Bilder sind durch zwei Ordner in Bilder mit und Bilder ohne Baumstämme aufgeteilt. Abbildung 2.6b zeigt ein Bild, in welchem der Greifer Baumstämme greift. In Abbildung 2.6a befinden sich keine Baumstämme im Greifer. Im Gegensatz zu dem Greiferdatensatz wurde dieser Datensatz (zum Teil) im Rahmen der Abschlussarbeit angefertigt. Hierzu wurde mit Quality Match GmbH [<https://www.quality-match.com>] zusammengearbeitet. Die Quality Match GmbH arbeitet mit Crowdworkern zusammen, um Datensätze mit kontrollierter Annotationsqualität zu erstellen.

**Weiterer Datensatz** Für Mitte 2020 ist ein weiterer Datensatz geplant. Dieser Datensatz enthält circa 80.000 beschriftete Bilder. Er wird sowohl die Annotation 'Logs ja / nein' als auch die Rahmen um den Greifer enthalten. Zusätzlich sind für diesen Datensatz weitere Beschriftungen wie Helligkeit, Winkel des Greifers und weitere Annotationen geplant.

### 2.7 Datenvorbereitung

In Vorbereitung auf die Modellierungsphase wurde ein finaler Datensatz erstellt und Werkzeuge zum Laden und vorbereiten der Daten implementiert.

**Erste Iteration** Als Erstes wurden die Daten mittels Skripten in Training, Test und Validation Daten aufgeteilt. Anschließend wurden die Daten auf der Cnvrge-Plattform in einen versionierbarer Datenspeicher geladen. In Tabelle 2.1 ist die finale Datenaufteilung zu sehen. Die Greifer Daten sind in 70% Trainingsdaten und jeweils 15% Test und Validierungsdaten aufgeteilt. Die Baumstammdaten sind in 80% Trainingsdaten 10% Testdaten und 10% Validierungsdaten getrennt worden. Die Trainingsdaten werden zum Trainieren der Modelle genutzt, die Testdaten zum Überprüfen der Modelle und die Validierungsdaten werden am Ende der Experimente als finale Überprüfung der Ergebnisse eingesetzt.

	Train	Test	Validation	Summe
<b>Greifer</b>	3.279	703	704	4.686
<b>Baumstämme j/n</b>	9.749	1.221	1.225	12.195

**Tabelle 2.1:** Datenaufteilung - Train Test Validation

Für das Laden der Daten wurde ein Modul 'data\_loader.py' erstellt. Dieses Modul enthält die drei Klassen DataLoader, GrappleDataLoader, LogsDataLoader. DataLoader ist eine abstrakte Klasse, welche Methoden zum Laden der Train-, Test- und Validationdaten definiert. Sie liefern entsprechend eines Parameters, bis zur maximalen Anzahl an Bildern, Bilder als Numarray zurück. Die Klassen GrappleDataLoader und LogsDataLoader implementieren für den jeweiligen Datensatz die konkreten Methoden zum Laden der Daten.

Mittels dem Modul 'data\_preparation.py' und der Klasse Preprocessing werden die Bilder auf die passende Größe verkleinert oder vergrößert und auf den Wertebereich zwischen 0 und 1 normalisiert.

**Zweite Iteration** In der zweiten Iteration wurde ein neues Modul namens data\_generator\_provider.py erstellt. Da Bilder als Numpyarray direkt in den Speicher geladen werden können nicht beliebig viele Bilder genutzt werden. Dieses Problem wird von den Keras ImageDataGeneratoren [<https://keras.io/preprocessing/image/>] adressiert. Sie erlauben es Bilder stapelweise bereitzustellen. Zusätzlich werden bei den Imagedatageneratoren direkt die Zielgrößen bereitgestellt. Das Modul implementiert jeweils für den Baumstamm Datensatz und den Greiferdatensatz eine Klasse zum Bereitstellen von ImageDatageneratoren.

Da die Modelle Mehrfach-Ausgang-Modelle sind erfolgt zusätzlich noch eine Aufbereitung der Bereitstellung der Daten. Standardmäßig stellen die Generatoren Stapel mit Einträgen der Form  $X, Y$  bereit. Wobei X die Eingangsdaten sind und Y die Zielgröße. Zum Beispiel kann X ein Bild sein und Y die zugehörige Klasse. Die Werkzeuge mit zweitem Kriterium benötigen Generatoren die Einträge der Stapel der Form  $X, [X, Y]$  erzeugen. Die Zielgröße hat sich zu einer Liste mit zwei Größen verändert. Die erste Zielgröße entspricht den Eingangsdaten, sie werden für den Decoder-Ausgang genutzt. Die zweite Zielgröße wird für den zweiten Ausgang genutzt. Sie entspricht der Zielgröße eines normalen ImageDataGeneratoren. Das Ganze wird mit Hilfe der Klasse tensorflow.keras.utils.Sequence erreicht. Ihr wird im Konstruktor ein ImageDataGenerator übergeben. Bei der Bereitstellung eines Elementes wird der Rückgabewert des Generators angepasst. Die entscheidenden Codezeilen sind in Listing 2.2 dargestellt.

```
1 res = self.generator.next()  
2 return res[0], [res[0], res[1]]
```

**Listing 2.2:** Aufbereitung Generatorergebnis in Python

Für das Vortrainieren werden Generatoren bereitgestellt, welche Stapel mit Einträgen der Form  $X, X$  erzeugen. Hierbei wird auf Standardfunktionalität der Klasse ImageDataGenerator zurückgegriffen.

Die ImageDataGeneratoren bieten Standardfunktionalität zum Anpassen der Daten. Alle Generatoren normalisieren die Werte der Bilder zwischen 0 und 1. Die Generatoren für die Trainingsdaten führen noch zufällige Veränderungen der Helligkeit

## 2 Grundlagen

---

und Kanalverschiebungen durch. Bei den Baumstamm Daten werden die Bilder zusätzlich horizontal umgedreht.

## 3. Werkzeuge

Dieses Kapitel erläutert die erstellten Werkzeuge im Detail. Die Werkzeuge bauen auf einander auf und werden der Reihe nach erläutert.

### 3.1 ConvolutionalSecondCriterionAutoencoder

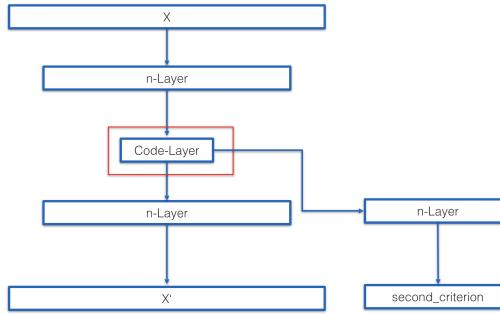
Der SCAE erweitert einen ConvolutionalAutoencoder um ein weiteres Kriterium. Es gibt also zusätzlich zu der Rekonstruktion des Autoencoders einen weiteren Ausgang. Der zweite Ausgang kann wie jeder Ausgang für eine Binärklassifikation, für eine Multiklassifikation, für eine Regression oder jede andere beliebige Aufgabe genutzt werden. In Abbildung 3.1 ist der schematische Aufbau des SCAE abgebildet. Die Schichten des zweiten Kriteriums werden an die Code-Schicht des NN angehängt. Es können beliebig viele Schichten genutzt werden. Die Verlustfunktion des NN besteht aus der Summe der einzelnen Verlustfunktionen und einer Gewichtung. Sie lautet im Detail:

$$loss = weight1 * loss\_autoencoder + weight2 * loss\_secondcriterion \quad (3.1)$$

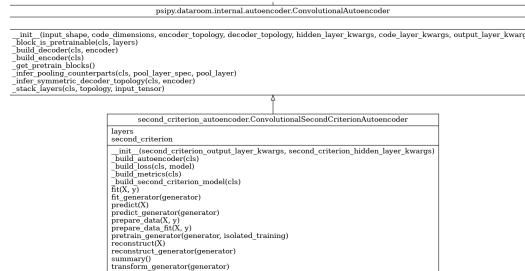
Die Gewichtung der Verlustfunktionen kann dem SCAE per Konstruktor-Argument übergeben werden. Das Werkzeug ist als Python-Module implementiert. Dabei implementiert die Klasse ConvolutionalSecondCriterionAutoencoder den ConvolutionalAutoencoder aus Psipy. In Abbildung 3.2 ist das Klassendiagramm des SCAE dargestellt. Über den Konstruktor können alle Argumente welche zum Erstellen des Models notwendig sind per doppeltes Sternchen Wörterbuch Argument (\*\*kwargs) an die Klasse übergeben werden. Diese Technik erlaubt es eine mit Schlüsselwörtern versehene Argumentliste variabler Länge zu übergeben. Die Argumentlisten werden beinahe in allen Methode zum Einsatz gebracht. Sie werden insbesondere genutzt, um Argumente an die zugehörigen Keras-Methoden zu übergeben. Die Na-

### 3 Werkzeuge

---



**Abbildung 3.1:** Schema SecondCriterionAutoencoder



**Abbildung 3.2:** Klassendiagramm ConvolutionalSecondCriterionAutoencoder

mensgebung der Methode orientiert sich dabei an Keras. So wird z. B. in dem Methodenaufruf *fit(..)* unter anderem auch die Keras-Methode *fit(..)* aufgerufen. Um einen SCAE zu trainieren, ist es notwendig eine Instanz zu erzeugen, die Methode *pretrain(..)* aufzurufen und ihn anschließend mit der Methode *fit(..)* zu trainieren. In dem Methodenaufruf *pretrain(..)* wird das Modell erstellt und schichtenweise vortrainiert. Das eigentliche Training erfolgt in der Methode *fit(..)*. Alternativ können auch die zugehörigen Generatorenklassen aufgerufen werden.

In Listing 3.1 ist beispielhaft dargestellt, wie ein SCAE erstellt wird. In den ersten 10 Zeilen wird die Architektur erstellt. Ab Zeile 12 wird eine Instanz eines CSCA mittels Argumentenliste erstellt. Zu beachten ist, dass hier keine Decoder-Architektur übergeben wird. Wenn keine Decoder-Architektur bereitgestellt wird sie beim Erstellen des eigentlichen Modells aus der Encoder-Architektur abgeleitet.

```

1  encoder_topology = [("Conv2D", {"filters": 8, "kernel_size": (3, 3)}),
2  ("Conv2D", {"filters": 8, "kernel_size": (3, 3)}),
3  ('MaxPooling2D', {"pool_size": (2, 2)}),
4  ("Conv2D", {"filters": 16, "kernel_size": (3, 3)}),
5  ('MaxPooling2D', {"pool_size": (2, 2)}),
6  ("Conv2D", {"filters": 16, "kernel_size": (3, 3)}),
7  ("Flatten", {}),
8  ("Dense", {"units": 16})]
9
10 second_criterion_topology = [{"Dense", {"units": num_classes}}] 
```

```

11 cscsa = ConvolutionalSecondCriterionAutoencoder(
12     input_shape=(28, 28, 1),
13     code_dimensions=3,
14     encoder_topology=encoder_topology,
15     second_criterion_topology=second_criterion_topology,
16     hidden_layer_kwargs = {'activation': 'relu'},
17     output_layer_kwargs = {'activation': 'sigmoid'},
18     second_criterion_hidden_layer_kwargs = {'activation': 'relu'},
19     second_criterion_output_layer_kwargs = {'activation': 'softmax'},
20     second_criterion_loss = 'categorical_crossentropy',
21     loss_weights=[8., 1.],
22     second_criterion_metrics = {'second_criterion': 'accuracy'},
23     code_layer_kwargs=dict())
24

```

**Listing 3.1:** Beispiel Erstellung ConvolutionalSecondCriterionAutoencoder in Python

Listing 3.2 zeigt den Aufruf der Methode Pretrain. Der Aufruf führt zu einem Schichtenweise-Trainieren des Netzwerkes mit den Daten x\_train bei 20 Epochen und einer Stapelgröße von 64.

```
1 cscsa.pretrain(x_train, epochs = 20, batch_size = 64)
```

**Listing 3.2:** Beispielauftrag Pretrain in Python

Der Methodenaufruf *fit(..)* funktioniert wie der *fit(..)-Aufruf* in Keras. In Zeile drei des Listing 3.2 ist zu erkennen, dass die Zielgrößen der verschiedenen Ausgänge einfach als Python-Wörterbuch übergeben werden können.

```

1 history = cscsa.fit(
2     x_train,
3     {"decoder": x_train, "second_criterion": y_train},
4     epochs=200,
5     batch_size = 64,
6     validation_data=(x_test, {"decoder": x_test, "second_criterion": y_test}))

```

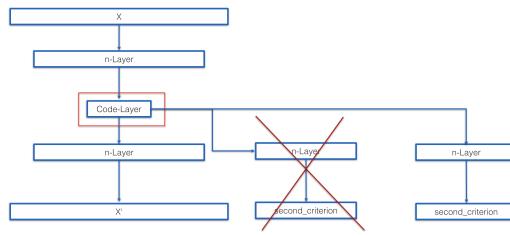
**Listing 3.3:** Beispielauftrag Fit in Python

## 3.2 TransferSecondCriterionAutoencoder

Ein TransferSecondCriterionAutoencoder basiert auf einem SCAE. Das zweite Kriterium wird durch ein neues Kriterium ersetzt. Zum Beispiel kann ein SCAE eine Objektkennung durchführen. Bei einem TransferSecondCriterionAutoencoder wird die Objektkennung durch eine Klassifizierungsaufgabe ersetzt. Die Architektur und die Gewichte des Autoencoders werden weiterverwendet. Abbildung 3.3 zeigt den Aufbau des TSCAE. Im Klassendiagramm 3.4 für den TSCAE ist zu sehen. Das

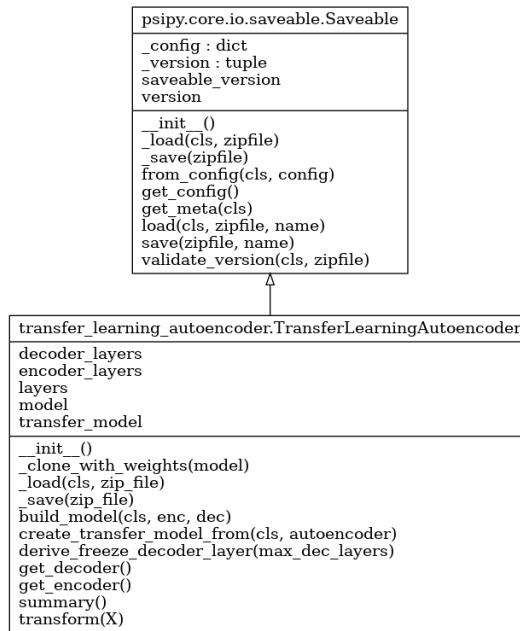
### 3 Werkzeuge

---



**Abbildung 3.3:** Schema TransferSecondCriterionAutoencoder

Besondere ist, dass ein SCAE als Konstruktorargument übergeben wird. Die Einstellungen für den ConvolutionalAutoencoder werden aus diesem Model kopiert. Zusätzlich müssen nur noch die Einstellungen für das zweite Kriterium übergeben werden. Aus diesen wird das Model für das zweite Kriterium erstellt. Neu hinzugekommen ist, ein Parameter(freeze\_encoder\_layers) über den eingestellt werden kann, ob und welche Schichten nicht neu trainiert werden können. Es können direkt Schichten ausgewählt werden oder es kann über einen Ganzahlenwert beginnend über die erste Schicht Schichten eingefroren. Werden nur Werte für den Encoder übergeben, werden Werte für den Decoder (freeze\_decoder\_layers) daraus abgeleitet. Listing 3.4 zeigt beispielhaft die Anwendung dieses Werkzeuges. Im Vergleich



**Abbildung 3.4:** Klassendiagramm TransferSecondCriterionAutoencoder

zu dem SCAE ist die Anwendung schon deutlich einfacher. Es gibt weniger Hyperparameter zum Beachten. Da das Modell auf einem trainierten SCAE basiert ist

kein *pretrain(..)* mehr notwenig. Die Methode *fit(..)* wird auf dieselbe Weiße wie bei SCAE angewendet.

```

1  tscm = TransferLearningConvolutionalSecondCriterionAutoencoder(csc_autoencoder,
2    second_criterion_topology=second_criterion_topology,
3    second_criterion_loss = 'binary_crossentropy',
4    second_criterion_hidden_layer_kwargs = {'activation': 'relu'},
5    second_criterion_output_layer_kwargs = {'activation': 'sigmoid'},
6    loss_weights=[1, 0.01],
7    freeze_encoder_layers = 2
8    ,freeze_decoder_layers =[0,1])
9
10 history = tscm.fit(
11   x_train,
12   {"decoder": x_train, "second_criterion": y_train},
13   epochs=1,
14   batch_size = 128,
15   validation_data=(x_test,{ "decoder": x_test, "second_criterion": y_test}))
16

```

**Listing 3.4:** Beispiel TransferSecondCriterionAutoencoder in Python

### 3.3 AutoTransferSecondCriterionAutoencoder

Der AutoTransferSecondCriterionAutoencoder ist ein TransferSecondCriterionAutoencoder welcher mithilfe von HpBandSter AutoML zur Hyperparameteroptimierung einsetzt. Konkret erbt die Klasse von `hpbandster.core.worker`. Zur Speicherung und Verwaltung der Hyperparameter wird auf die Klasse `HyperparameterMixin` zurückgegriffen. Bei der Instanziierung der Klasse werden sinnvolle Standardwerte für Hyperparameter gesetzt. Die Werte können aber später noch angepasst werden. In Abbildung 3.5 ist das zugehörige Klassendiagramm abgebildet. In Listing 3.5 ist eine einfache Implementierung eines AutoTransferSecondCriterionAutoencoder dargestellt. Der Konstruktor unterscheidet sich nur an einer Stelle zum Konstruktor des TransferSecondCriterionAutoencoder. Es wird keine Instanz des SCAE übergeben, sondern ein Pfad zu einem abgespeicherten Modell eines SCAE. Im Gegensatz zur bisherigen Vorgehensweise werden die Trainings und Testdaten mit der Methode *set\_generators(..)* oder *set\_numpydata(..)* übergeben. Ziel ist es dabei die Komplexität der Methode *optimize(..)* zu reduzieren. Durch den Aufruf von *optimize(..)* wird der Optimierungsvorgang gestartet. Die Parameter sind dabei die Anzahl an Iterationen, die Optimierungsstrategie und ein Wörterbuch, welches zusätzliche Einstellungen für die einzelnen Optimierer enthalten kann. In jeder Itera-

### 3 Werkzeuge



Abbildung 3.5: Klassendiagramm AutoTransferSecondCriterionAutoencoder

tion der Optimierung wird ein neuer TransferSecondCriterionAutoencoder erstellt und mittels der ausgewählten Hyperparameter und übergebenen Daten trainiert.

```
1 tscm = AutoTransferConvolutionalSecondCriterionAutoencoder(max_deep_freeze=2,
2     path_to_model = path_to_base_model,
3     second_criterion_topology=second_criterion_topology,
4     second_criterion_loss = 'categorical_crossentropy',
5     second_criterion_hidden_layer_kwarg = {'activation': 'relu'},
6     second_criterion_output_layer_kwarg = {'activation': 'softmax'},
7     second_criterion_metrics = {'second_criterion': 'accuracy'}
8 )
9
10 tscm.set_generators(train_datagenerator, test_datagenerator)
11
12
13 best_config, history = tscm.optimize(3
14     , 'RandomSearch'
15     , optimization_kwarg = optimization_kwarg)
```

Listing 3.5: Beispiel AutoTransferSecondCriterionAutoencoder in Python

Das Werkzeug unterstützt derzeit die Optimierer, Randomsearch, Hyperband und BOHB. In Tabelle 3.1 sind die Hyperaparameter und ihre mögliche Werte dargestellt. Der Wertebereich für die Stapelgröße wurde aus der Autocrane-Problemstellung abgeleitet. Sie kann maximal, so groß werde, dass kein Speicherplatz-Fehler auftreten kann. Die maximale Anzahl der Epochen wurde bewusst groß gewählt. Es sollen weitere Problemstellungen ohne viel Konfigurationsaufwand gelöst werden können. Absurd lange Laufzeiten können durch ein EarlyStopping Kriterium verhindert werden. Die Maximalanzahl der möglicherweise einzufrierenden Schichten wird über den Anwender des Werkzeuges definiert, sie sind anwendungsspezifisch.

durch Verhältnis  
anpassen.

Budget berück-  
gen / notieren

<b>Hyperparameter</b>	<b>Werte</b>	<b>Datentyp</b>
<b>Optimierer</b>	Adam, sgd, rmsprop	Kategorial
<b>Batch_size</b>	32 - 1024	Ganzzahl
<b>Epochen</b>	10 - 10.000	Ganzzahl
<b>autoencoder_loss_weight</b>	0.01 - 1	Fließkommazahl
<b>second_criterion_loss_weight</b>	0.01 - 1	Fließkommazahl
<b>freeze_encoder_layers</b>	0 - Parameter	Ganzzahl
<b>freeze_decoder_layers</b>	0 - Parameter	Ganzzahl

**Tabelle 3.1:** Standard Hyperparameter für AutoML-Suche



## 4. Experimente

- Callbacks

### 4.1 Basislinie

wie wird prinzipiell evaluiert? PSIORI Grapple-detection PSIORI Logs-Klassifizierung

### 4.2 ConvolutionalSecondCriterionAutoencoder

**CodeLayer** 3,7,10,15,...

**Gewichtung der Verlustfunktion** 1/1; 1/10; 10/1; 1/0; 0/1;

**Bestes Ergebnis** Datenänderungen (führen zu es funktioniert nicht) Kohavi, R., John, G.: Automatic Parameter Selection by Minimizing Estimated Error. In: Priditidis, A., Russell, S. (eds.) Proceedings of the Twelfth International Conference on Machine Learning, pp. 304–312. Morgan Kaufmann Publishers (1995))

### 4.3 TransferSecondCriterionAutoencoder

**Datenmenge**

**Gewichtung der Verlustfunktion**

#### **4.4 AutoTransferSecondCriterionAutoencoder**

## **5. Fazit**

### **5.1 Zusammenfassung**

### **5.2 Kritische Reflexion**

### **5.3 Ausblick**

insbesondere die möglichen Addons aufführen





# Abkürzungsverzeichnis

<b>AutoML</b>	automatisiertes maschinelles Lernen . . . . .	6
<b>NAS</b>	Neural Architecture Search . . . . .	6
<b>HPO</b>	Hyperparameter-Optimierung . . . . .	6



# **Tabellenverzeichnis**

2.1	Datenaufteilung - Train Test Validation . . . . .	14
3.1	Standard Hyperparameter für AutoML-Suche . . . . .	23



# **Abbildungsverzeichnis**

2.1	Schema Autoencoder . . . . .	5
2.2	Beispiel PSIORI Visualizer . . . . .	8
2.3	Klassendiagramm ConvolutionalAutoencoder . . . . .	10
2.4	Klassendiagramm Hyperparametermixin . . . . .	10
2.5	Rundlaufkran . . . . .	11
2.6	Greifer . . . . .	12
2.7	Bildqualität . . . . .	13
3.1	Schema SecondCriterionAutoenocder . . . . .	18
3.2	Klassendiagramm ConvolutionalSecondCriterionAutoencoder . . . . .	18
3.3	Schema TransferSecondCriterionAutoenocder . . . . .	20
3.4	Klassendiagramm TransferSecondCriterionAutoenocder . . . . .	20
3.5	Klassendiagramm AutoTransferSecondCriterionAutoenocder . . . . .	22



# Listings

2.1	Erweiterung zum Speichern eines Tensorflow Models . . . . .	9
2.2	Aufbereitung Generatorergebnis in Python . . . . .	15
3.1	Beispiel Erstellung ConvolutionalSecondCriterionAutoencoder in Python . . . . .	18
3.2	Beispieldruck Pretrain in Python . . . . .	19
3.3	Beispieldruck Fit in Python . . . . .	19
3.4	Beispiel TransferSecondCriterionAutoencoder in Python . . . . .	21
3.5	Beispiel AutoTransferSecondCriterionAutoencoder in Python . . . . .	22



# Literatur

- [Be07] Bengio, Y.; Lamblin, P.; Popovici, D.; Larochelle, H.; Montreal, U.: Greedy layer-wise training of deep networks. In. Bd. 19, 2007.
- [Be12] Bergstra, James, and Yoshua Bengio.: Random search for hyper-parameter optimization. Journal of Machine Learning Research 13/, S. 281–305, 2012, URL: <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.
- [Ch15] Chollet, F. et al.: Keras, 2015.
- [cn] cnvrg.io: cnvrg.io, URL: <https://cnvrg.io/>.
- [EMH19] Elsken, T.; Metzen, J. H.; Hutter, F.: Neural Architecture Search: 3. In (Hutter, F.; Kotthoff, L.; Vanschoren, J., Hrsg.): Automatic Machine Learning: Methods, Systems, Challenges. Springer, S. 69–86, 2019.
- [FH19] Feurer, M.; Hutter, F.: Hyperparameter Optimization: 1. In (Hutter, F.; Kotthoff, L.; Vanschoren, J., Hrsg.): Automatic Machine Learning: Methods, Systems, Challenges. Springer, S. 3–38, 2019.
- [HKV19] Hutter, F.; Kotthoff, L.; Vanschoren, J., Hrsg.: Automatic Machine Learning: Methods, Systems, Challenges. Springer, 2019.
- [Hu07] Hunter: Matplotlib: A 2D graphics environment, 2007.
- [Jo18] Joaquin Vanschoren: Meta-Learning: A Survey. CoRR abs/1810.03548/, 2018.
- [KJ] Kohavi, R.; John, G.: Autmatic Parameter Selection by Minimizing Estimated Error. In. S. 304–312.
- [Le99] LeCun, Y.; Haffner, P.; Bottou, L.; Bengio, Y.: Object Recognition with Gradient-Based Learning. In: Shape, Contour and Grouping in Computer Vision. Springer Berlin Heidelberg, Berlin, Heidelberg, S. 319–345, 1999, ISBN: 978-3-540-46805-9.

- [Li17] Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; Talwalkar, A.: HYPERBAND: BANDIT-BASED CONFIGURATION EVALUATION FOR HYPERPARAMETER OPTIMIZATION. ICLR/, 2017, URL: <https://openreview.net/pdf?id=ry18Ww5ee>.
- [Ma11] Masci, J.; Meier, U.; Cireşan, D.; Schmidhuber, J.: Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction. In (Honkela, T.; Duch, W.; Girolami, M.; Kaski, S., Hrsg.): Artificial Neural Networks and Machine Learning – ICANN 2011. Springer Berlin Heidelberg, Berlin, Heidelberg, S. 52–59, 2011, ISBN: 978-3-642-21735-7.
- [Ma15] Martin Abadi; Ashish Agarwal; Paul Barham; Eugene Brevdo; Zhifeng Chen; Craig Citro; Corrado, G. S.; Andy Davis; Jeffrey Dean; Matthieu Devin; Sanjay Ghemawat; Ian Goodfellow; Andrew Harp; Geoffrey Irving; Michael Isard; Jia, Y.; Rafal Jozefowicz; Lukasz Kaiser; Manjunath Kudlur; Josh Levenberg; Dandelion Mané; Rajat Monga; Sherry Moore; Derek Murray; Chris Olah; Mike Schuster; Jonathon Shlens; Benoit Steiner; Ilya Sutskever; Kunal Talwar; Paul Tucker; Vincent Vanhoucke; Vijay Vasudevan; Fernanda Viégas; Oriol Vinyals; Pete Warden; Martin Wattenberg; Martin Wicke; Yuan Yu; Xiaoqiang Zheng: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015, URL: <https://www.tensorflow.org/>.
- [Mi18] Michelucci, U.: Hyperparameter Tuning. In: Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks. Apress, Berkeley, CA, S. 271–322, 2018, ISBN: 978-1-4842-3790-8.
- [Mi20] Micorsoft: Microsoft Azure, 2020, URL: <https://azure.microsoft.com/de-de/>.
- [Nv20] Nvidia: Tesla K80, 2020, URL: <https://www.nvidia.com/de-de/data-center/tesla-k80/>.
- [Ol06] Oliphant, T.: NumPy: A guide to NumPy, hrsg. von Trelgol Publishing USA, 2006, URL: <http://www.numpy.org/>.
- [Pe11] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12/, S. 2825–2830, 2011.
- [PS19] PSIORI GmbH: psipy documentation, hrsg. von PSIORI GmbH, 2019, URL: <https://psipy.azurewebsites.net/source/psipy.html>.

- [PS20] PSIORI GmbH: PSIORI, 2020, URL: <https://www.psiori.com/de>.
- [Py20] Python Software Foundation: The Python Language Reference, 2020, URL: <https://docs.python.org/3.7/reference/>.
- [SAF18] Stefan Falkner; Aaron Klein; Frank Hutter: BOHB: Robust and Efficient Hyperparameter Optimization at Scale. arXiv:1807.01774/, 2018, URL: <https://openreview.net/pdf?id=ry18Ww5ee>.
- [Va19] Vanschoren, J.: Meta-Learning: 2. In (Hutter, F.; Kotthoff, L.; Vanschoren, J., Hrsg.): Automatic Machine Learning: Methods, Systems, Challenges. Springer, S. 39–68, 2019.