

Automatisierte Machine Learning-basierte Erfassung und Auswertung geschriebener Spielergebnisse

STUDIENARBEIT

für die Prüfung zum

Bachelor of Science (B.Sc.)

des Studiengangs Cybersecurity / Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

von

Tom Sperling

Alexander Scherer

Dirk Kremer

April 2022

Bearbeitungszeitraum

18. Oktober 2021 - 19. April 2022 (02. Mai. 2022)

Matrikelnummer, Kurs

6397318, TINF19CS1

Matrikelnummer, Kurs

3003040, TINF19CS1

Matrikelnummer, Kurs

8879153, TINF19IT2

Betreuer der DHBW

Florian Rosenzweig

Unterschrift Betreuer

Erklärung

**Gemäß §5 (3) der „Studien- und Prüfungsordnung DHBW Technik – StuPrO
DHBW Technik“ vom 29. September 2017.**

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: *Automatisierte Machine Learning-basierte Erfassung und Auswertung geschriebener Spielergebnisse selbstständig verfasst* und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

** falls beide Fassungen gefordert sind*

DHBW Mannheim, April 2022

Tom Sperling

Alexander Scherer

Dirk Kremer

Kurzfassung

Maschinelles Lernen ist seit einigen Jahren ein wichtiger Bereich in der Informationstechnik und dort nicht mehr wegzudenken. Die Einsatzmöglichkeiten sind dabei nahezu unbegrenzt. So wird es von Netflix, oder anderen Streaminganbietern, zum Vorschlagen neuer Serien und Filme genutzt, als Spamfilter bei E-Mails oder digitale Assistenten wie *Alexa* von Amazon oder *Google Now* von Google. Aber auch zur Objekterkennung in Bilder wie in der Medizin zur Erkennung von Tumoren oder zur Gesichtserkennung bei Kameras. Die Entwicklung von maschinellem Lernen ist dabei keineswegs abgeschlossen, sondern schreitet immer weiter voran. Es werden somit immer neue Einsatzgebiete gefunden, dazu gehört z.B. das autonome Fahren.

Im Rahmen dieser Arbeit sollen die Technologien des maschinellen Lernens verwendet werden, um ein Kniffel Ergebnisblatt auszuwerten. Die ausgewerteten Ergebnisse sollen ansprechend dargestellt werden, um dem Nutzer einen Überblick zu ermöglichen. Außerdem soll eine langfristige Auswertung der Ergebnisse ermöglicht werden.

Für ein möglichst gutes und genaues Ergebnis soll zunächst ein Vergleich zwischen verschiedenen Machine Learning Algorithmen durchgeführt werden. Im Zuge dieser Arbeit wurde sich dabei für die Algorithmen *K-Nearest-Neighbor (KNN)*, *Support Vector Machine (SVM)* und *Convolutional Neural Network (CNN)* entschieden.

Abstract

Machine learning has been an important area in information technology for several years and it is impossible to imagine life without it. The possible applications are almost unlimited. For example, it is used by Netflix or other streaming providers to suggest new series and movies, as a spam filter for e-mails or digital assistants such as *Alexa* from Amazon or *Google Now* from Google. But also for object recognition in images like in medicine for tumor detection or for face recognition in cameras. The development of machine learning is by no means finished, but continues to progress. Thus, new fields of application are found all the time, including autonomous driving, for example.

In the context of this work the technologies of the machine learning are to be used, in order to evaluate a Kniffel result sheet. The evaluated results shall be presented in an appealing way to provide the user with an overview. Furthermore, a long-term evaluation of the results shall be made possible.

For the best and most accurate results, a comparison between different machine learning algorithms should be performed first. In the course of this work, the algorithms *K-Nearest-Neighbor (KNN)*, *Support Vector Machine (SVM)* and *Convolutional Neural Network (CNN)* were chosen.

Vorwort

In dieser Arbeit werden einige besondere Begriffe hervorgehoben. Zum Einen sind kursiv geschriebene Ausdrücke Eigennamen oder unternehmensspezifische Bezeichnungen und werden nicht näher erläutert. Zum Anderen sind fachspezifische Begriffe und Abkürzungen in blauer Farbe markiert und werden im Abkürzungsverzeichnis auf Seite [X](#), beziehungsweise im Glossar auf Seite [XI](#) erläutert. In digitaler Form sind diese mit einem Hyperlink zur jeweiligen Seite versehen.

Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit die Sprachform des generischen Maskulinums angewendet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

Inhaltsverzeichnis

Abbildungsverzeichnis	VII
Tabellenverzeichnis	VIII
Quellcodeverzeichnis	IX
Abkürzungsverzeichnis	X
1 Einleitung	1
1.1 Motivation	1
1.2 Aufgabenstellung	1
1.3 Vorgehensweise	2
1.4 Aufbau der Studienarbeit	3
2 Theorie	4
2.1 Machine Learning	4
2.1.1 Grundlegende Theorie zu <i>Machine Learning</i>	4
2.1.2 MNIST	5
2.2 Vergleich verschiedener <i>Machine Learning</i> -Algorithmen	6
2.2.1 K-Nearest-Neighbor	6
2.2.2 Support Vector Machine	12
2.2.3 Convolutional Neural Networks	21
2.2.4 Auswertung des Vergleiches	32
3 Konzept	34
3.1 Genutzte Technologien	34
3.1.1 OpenCV	34
3.1.2 SciKit-learn	35
3.1.3 Tensorflow	35
3.1.4 Flask	35
3.1.5 Dash	36
3.2 Auswertung und Visualisierung der Daten	36
3.2.1 Auswertung	36
3.2.2 Visualisierung	38
4 Umsetzung	40
4.1 Image Preprocessing / Bildvorverarbeitung	40
4.2 Zahlenerkennung	45
4.2.1 Probleme bei der Zahlenerkennung	46
4.3 Auswertung & Darstellung	49

5 Ergebnisse und Ausblick **51**

Literatur **53**

Abbildungsverzeichnis

2.1	Funktionsweise des <i>KNN</i>	9
2.2	Beispiel für eine <i>Hyperplane</i>	13
2.3	Beispiele für <i>separierende Hyperplanes</i>	15
2.4	Beispiel für eine <i>Maximal Margin Hyperplane</i>	16
2.5	<i>Hyperplanes</i> bei nicht-linearen Daten [6, S. 353]	17
2.6	Ein <i>fully-connected ANN</i> mit zwei versteckten Schichten [11]	22
2.7	Bildliche Darstellung der <i>Aktivierungsfunktion</i> [11]	23
2.8	Graph einer <i>Kreuzentropie</i> [15]	24
3.1	MockUp für die Visualisierung der Daten	39
4.1	<i>Image Alignment</i>	41
4.2	Korrespondenzen zwischen <i>Key Points</i>	42
4.3	Zahlen in einem 8×8 Gitter	47
4.4	Zahlen in einem 28×28 Gitter	48
4.5	Screenshot vom Dashboard mit Dummy-Daten, selbst erstellt und zugeschnitten	50

Tabellenverzeichnis

1.1 User Stories	2
----------------------------	---

Quellcodeverzeichnis

2.1	Pythoncode zum Testen des KNN	10
2.2	Testergebnisse des KNN	11
2.3	Pythoncode zum Testen der SVM	19
2.4	Testergebnisse der SVM	20
2.5	Pythoncode zur Testvorbereitung vom CNN	28
2.6	Pythoncode zur Konstruktion vom CNN	29
2.7	Pythoncode zum Darstellen der Daten	30
2.8	Hilfsfunktion für Präzision [1]	31
2.9	Testergebnisse vom CNN	31
4.1	Code für <i>Image Alignment</i>	44
4.2	Testergebnisse der SVM mit 28x28 großen Bildern	49

Abkürzungsverzeichnis

ANN Artificial Neural Network

API Application Programming Interface

CNN Convolutional Neural Network

KNN K-Nearest-Neighbor

ReLU Rectified Linear Unit

SVM Support Vector Machine

1 Einleitung

1.1 Motivation

Brett- und Gemeinschaftsspiele haben während der Corona-Pandemie einen „Boom“ erlebt. Es war keine Seltenheit, dass auf herkömmliche Gesellschaftsspiele zurückgegriffen wurde, um die Zeit im Lockdown zu überbrücken. [2]

Persönlich wurde häufig das Spiel *Kniffel* mit der Familie gespielt, bei dem jeder Spieler seine Spielergebnisse auf ein Blatt Papier niederschreibt. Als Studenten der Informatik und Interessierte der Statistik sind die Spielergebnisse in analoger Form so nur mühselig auszuwerten. Um zum Beispiel Trends über mehrere Spiele hinweg zu analysieren, müsste eine Digitalisierung der Daten vorgenommen werden. Diese Digitalisierung ist jedoch als manuelle Datenverarbeitung sehr zeitaufwendig.

In den letzten Jahren ist *Machine Learning* und *Artificial Intelligence* im Bereich der Informatik ein stetig wachsendes Forschungsgebiet. Es entstehen zunehmend Lösungen für Probleme unter dem Einsatz von Algorithmen aus dem Bereich der *Künstlichen Intelligenz*. Die allgemeine Erkennung von Zahlen auf einem Bild ist bereits ein ausgiebig untersuchtes Forschungsgebiet. Der Einsatz für die Digitalisierung von Spielergebnissen vom Spiel Kniffel ist jedoch weitestgehend unerforscht.

1.2 Aufgabenstellung

Im Rahmen der vorliegenden Studienarbeit soll ein entsprechendes System entwickelt werden, mit dessen Hilfe analoge papierbasierte Spielergebnisse des Gesellschaftsspieles „Kniffel“ beziehungsweise „Yathzee“ digitalisiert werden können. Für die Erkennung der einzelnen Spielergebnisse und die Überführung in ein digitales Format soll eine automatische *Machine Learning*-gestützte Bilderkennung eingesetzt werden. Nach der Digitalisierung der Daten sollen die Spielergebnisse ausgewertet und visualisiert werden.

Für die Implementierung der Webapplikation können drei große Arbeitspakete identifiziert werden. Zuerst müssen die Daten hochgeladen werden können und das hochgeladene Bild mit den analogen Spielergebnissen für die Bildverarbeitung vorbereitet werden. Nach dem Prozess des Image Processing müssen mit Hilfe eines *Machine Learning*-Algorithmus die einzelnen

ID	User Story
01	Als ein Nutzer SOLL ein Bild hochgeladen werden können
02	Als ein Nutzer SOLLEN die Spielergebnisse auf dem Bild mit einem Machine Learning Algorithmus ausgewertet werden
03	Als ein Nutzer SOLLEN die digitalisierten Spielergebnisse überprüfbar und wenn nötig korrigierbar sein
04	Als ein Nutzer SOLLEN die digitalisierten Spielergebnisse in einer Datenbank abgespeichert werden
05	Als ein Nutzer SOLL eine statistische Auswertung aller hochgeladenen Spielergebnisse zur Verfügung stehen
06	Als ein Nutzer KANN man sich registrieren
07	Als ein Nutzer KANN man sich ein- und ausloggen
08	Als ein Nutzer KANN man seine eigenen hochgeladenen Spielergebnisse anzeigen lassen

Tabelle 1.1: User Stories

Ziffern auf dem Bild erkannt werden. Nach der Bilderkennung sollen die Spielergebnisse dann ausgewertet und für den Nutzer visualisiert werden.

Für das Software-Projekt wurden die User Stories in der Tabelle 1.1 definiert.

1.3 Vorgehensweise

Bevor die Umsetzung der User Stories für das zu entwickelnde System erfolgt, soll zunächst ein wissenschaftlicher Vergleich verschiedener Algorithmen zur Ziffernerkennung durchgeführt werden. Der Vergleich soll dazu dienen, einen geeigneten Algorithmus für das Projekt auszuwählen. Es wurde im Vorhinein eine Auswahl der zu untersuchenden Algorithmen getroffen. Die drei *Machine Learning*-Algorithmen, die es zu vergleichen gilt, sind *K-Nearest-Neighbor*, die *Support Vector Machine* und *Convolutional Neural Networks*.

Nach der Auswertung des Vergleichs soll die Umsetzung des Systems erfolgen. Die Umsetzung erfolgt durch den Einsatz der definierten User Stories in 1.1, welche für den Entwicklungsprozess in weitere kleinere Arbeitspakete unterteilt wurden. Der Entwicklungsprozess wurde agil gestaltet durch die Verwendung von *Kanban*. Die definierten Arbeitspakete wurden auf einem *Kanban*-Board visualisiert und zugeordnet.

1.4 Aufbau der Studienarbeit

Der Aufbau der Studienarbeit ist so gestaltet, dass am Anfang eine Einführung in das Thema *Machine Learning* stattfindet, um die Grundlagen des Themas zu erläutern. Die Einführung beginnt in Kapitel 2.1 und geht dann über in den Vergleich der verschiedenen Algorithmen der Bilderkennung in Kapitel 2.2. Der Vergleich beginnt, indem jeder Algorithmus zunächst vorgestellt und der Ablauf des jeweiligen Tests behandelt wird. Die Tests werden in der folgenden Reihenfolge durchgeführt und diskutiert:

- *K-Nearest-Neighbor* 2.2.1
- *Support Vector Machine* 2.2.2
- *Convolutional Neural Networks* 2.2.3

Anschließend wird im Kapitel 2.2.4 die Evaluierung des Vergleichs vorgenommen und diskutiert, weshalb welcher Algorithmus für die Umsetzung ausgewählt wurde.

In Kapitel 3 wird auf die Technologien eingegangen, welche für die Umsetzung des Systems verwendet werden. Das Kapitel beschäftigt sich ebenfalls mit der Konzeptionierung der Datenvisualisierung.

Die Umsetzung des Systems wird in Kapitel 4 dargelegt. Dabei wird besonders darauf eingegangen, welche Herausforderungen während der Entwicklung des Systems aufkamen und wie diese gelöst wurden. Für die Umsetzung kristallisierten sich drei größere, umfassende Arbeitsschritte heraus. In jedem Unterkapitel 4.1, 4.2, 4.3 wird auf das jeweilige Thema eingegangen und die dazugehörigen Arbeitsschritte diskutiert.

Abschließend werden im Kapitel 5 die Ergebnisse der Studienarbeit diskutiert und ein Ausblick gegeben über mögliche Forschungen und Erweiterungen im Themenfeld.

2 Theorie

2.1 Machine Learning

In den nachfolgenden Kapiteln werden die nötigen Grundlagen zum Thema *Machine Learning* erläutert, die für das Verständnis dieser Arbeit relevant sind. Zu Beginn wird es eine kurze Einleitung geben, in der *Machine Learning* definiert und erklärt wird. Außerdem wird die MNIST-Datenbank vorgestellt, deren Daten als Basis für unsere Algorithmen dienen wird. Daraufhin werden die einzelnen *Machine Learning*-Algorithmen näher beschrieben, die in die engere Auswahl für das Projekt gekommen sind. Dazu gehören die Algorithmen *K-Nearest Neighbor*, die *Support Vector Machine* und *Convolutional Neural Networks*. Diese werden dann in einem späteren Kapitel miteinander verglichen, um den geeigneten *Machine Learning*-Algorithmus für dieses Projekt zu ermitteln.

2.1.1 Grundlegende Theorie zu *Machine Learning*

Machine Learning oder auf Deutsch maschinelles Lernen ist ein Teilgebiet innerhalb der Informatik, das sich aus dem Bereich der Künstlichen Intelligenz entwickelt hat. Es handelt sich hierbei um die Entwicklung von Algorithmen, die auf der Basis eines Datensatzes trainiert werden und daraufhin Vorhersagen über weitere unbekannte Daten treffen können. Beim maschinellen Lernen wird, im Gegensatz zur traditionellen Programmierung keine Liste an statischen Programmanweisungen ausgeführt, sondern es wird anhand von Beispieldaten ein Modell konstruiert, mit dem dann datengesteuerte Vorhersagen oder Entscheidungen getroffen werden können.^[3]

Wenn wir zum Beispiel mit Hilfe eines Programmes den Spritverbrauch eines PKWs vorhersagen wollen, müssen zuerst Daten über das Auto gesammelt werden, die womöglich im Zusammenhang mit dem Spritverbrauch stehen könnten. Diese Daten des PKWs werden dann mit dem dazugehörigen Spritverbrauch in einen der verschiedenen verfügbaren *Machine Learning*-Algorithmen gegeben, der auf Basis dieser Daten trainiert wird. Nachdem der Prozess des maschinellen Lernens abgeschlossen ist, können auch Parameter bis dahin unbekannter Autos hineingegeben werden und der Algorithmus kann nun anhand der Trainingsdaten eine Prognose für den Spritverbrauch des unbekannten Autos abgeben. *Machine Learning* kann in vielen unterschiedlichen Situationen eingesetzt werden. Es braucht jedoch immer eine große Anzahl an Trainingsdaten, um einen solchen Algorithmus sinnvoll trainieren zu können.

Eine einfache Form des *Machine Learning* ist die lineare-Regression. Dabei handelt es sich auch um einen Algorithmus, der auch Vorhersagen treffen kann. Die lineare-Regression basiert auf einer mathematischen Gleichung der Form: $h(x) = \theta_0 + \theta_1x$ mit θ_0 und θ_1 als Konstanten. Die Trainingsdaten enthalten jeweils einen Input x und einen dazugehörigen Output y . Bei dem obigen Beispiel wäre y der Spritverbrauch und x zum Beispiel der Hubraum. Beim Trainieren versucht der Algorithmus die beiden Konstanten θ_0 und θ_1 in der Gleichung $h(x)$ so zu wählen, dass der Unterschied zum erwarteten Output y minimiert wird. Nach dem Trainieren ist die Gleichung optimiert und es wurden Werte für die beiden Konstanten ausgewählt. Nun kann auch ein unbekanntes x in die Funktion gegeben werden, die dann eine Vorhersage für y berechnet.^[3]

Dies wäre ein einfaches Beispiel für den Einsatz vom maschinellen Lernen. Es können jedoch auch mathematisch kompliziertere Sachverhalte durch *Machine Learning* gelöst werden. Ein Einsatzbereich ist zum Beispiel die Erkennung von handgeschriebenen Ziffern anhand einer Bilddatei. Auch hier kann durch maschinelles Lernen ein Algorithmus mit einer Vielzahl an Bilddateien trainiert werden, damit dieser enthaltene Muster erkennt und eine Vermutung für die abgebildete Ziffer abgibt.

2.1.2 MNIST

Für die Erkennung von handgeschriebenen Ziffern trainieren wir den *Machine Learning*-Algorithmus mit den Daten aus der MNIST-Datenbank. MNIST steht für *Modified National Institute of Standards and Technology Datenbank*. Es ist eine öffentlich zugängliche Datenbank, die handgeschriebene Ziffern enthält. Sie enthält einen Trainingssatz aus 60.000 Beispielen, sowie einen dazugehörigen Testdatensatz, der 10.000 Testobjekte enthält. Dabei ist jede Ziffer als ein Graustufen-Bild der Größe 28x28 abgespeichert. Mit diesem Datensatz können einfach *Machine Learning*-Algorithmen, wie *Convolutional Neural Networks* oder eine *Support Vector Machine* trainiert werden, um dann unbekannte handschriftliche Ziffern zu erkennen.

2.2 Vergleich verschiedener *Machine Learning*-Algorithmen

In diesem Kapitel sollen unterschiedliche *Machine Learning*-Algorithmen im Hinblick auf das Projekt miteinander verglichen werden. Es wurde bereits eine Vorauswahl getroffen und für den Vergleich wurden die Algorithmen *K-Nearest-Neighbor*, die *Support Vector Machine* und *Convolutional Neural Networks* ausgewählt. Für einen fairen Vergleich wird beim Trainieren aller zuvor genannten *Machine Learning*-Algorithmen die bereits vorgestellte MNIST-Datenbank verwendet, die Datensätze zu handschriftlichen Ziffern enthält.

Die Kriterien, nach denen der Vergleich durchgeführt werden soll, wurden bereits im Vorfeld definiert. Es wurde sich auf die Kriterien Komplexität, Präzision, Genauigkeit sowie die Einfachheit der Umsetzung geeinigt. Die **Komplexität** beschreibt hier die (*Zeit-*)**Komplexität** des verwendeten Algorithmus und wird mit der O-Notation dargestellt. Die **Genauigkeit** bei *Machine Learning*-Algorithmen ist definiert durch die Anzahl der richtig zugewiesenen Trainingsobjekte durch die Anzahl aller Trainingsobjekte, während die **Präzision** durch folgende Gleichung bestimmt werden kann:

$$\text{Präzision} = \frac{\text{Richtig Positiv}}{\text{Richtig Positiv} + \text{Falsch Positiv}}$$

Danach wird die **Einfachheit der Umsetzung** der zu bewertenden *Machine Learning*-Algorithmen von der testenden Person auf Basis einer Skala von eins bis fünf beurteilt.

In den folgenden Unterkapiteln werden die drei *Machine Learning*-Algorithmen aus der Vorauswahl näher beschrieben und eine Evaluation nach den genannten Kriterien vorgenommen.

2.2.1 K-Nearest-Neighbor

Der *K-Nearest-Neighbor (KNN)*, zu Deutsch 'K-nächste Nachbarn', ist ein Machine Learning Algorithmus, der 1951 von Joseph Hodeges und Evelyn Fix entwickelt und später von Thomas Cover weiterentwickelt wurde.

Der K-Nearest-Neighbors (*KNNs*) eignet sich zur Klassifizierung und Regression von Objekten, wurde im Laufe dieser Arbeit aber ausschließlich zur Klassifizierung verwendet. Der Algorithmus gehört zu den überwachten Lernverfahren, d.h. er nutzt als Eingabeobjekte während des Trainings ein Bild, das später auch zur Klassifizierung übergeben wird. Zusätzlich wird auch noch ein Label mit jedem Bild übergeben, das angibt, um welches Objekt es sich handelt.

Da der *KNN-Algorithmus* zur Kategorie der *Lazy Learner* gehört, erstellt dieser kein Modell während der Trainingsphase, sondern erst, sobald eine Klassifizierungsanfrage stattfindet. Während der Trainingsphase passen *Lazy Learner* keine Gewichte an, um damit genauere

Vorhersagen treffen zu können, sondern sie speichern lediglich die Trainingsdaten anhand ihrer Merkmale in einem \mathcal{R}^N dimensionalem Diagramm ab, N entspricht hierbei der Anzahl der Merkmale. Diese Merkmale bilden später die Grundlage für die Klassifizierung. Dieser Entscheidungsprozess ist dabei einer der denkbar einfachsten, denn es wird lediglich mit den k nächsten Nachbarn eine Mehrheitsentscheidung getroffen. Das Ergebnis davon bestimmt die Kategorie des neuen Objektes.

Der Entscheidungsprozess des *KNN* läuft dabei in 5 Schritten ab:

1. Bestimmen von K
2. Berechnen der Distanz anderer Objekte
3. K nächste Nachbarn bestimmen
4. Bestimmung der Kategorie der Nachbarn
5. Kategorie zuweisen

1. Bestimmen von K

K ist eine feste Konstante, die vom Benutzer festgelegt wird. Dabei gibt es keinen festen Wert, um die perfekte Anzahl an Nachbarn zu bestimmen. Je nach Einsatzzweck kann es sinnvoll sein, ein höheres oder niedrigeres k zu wählen. Bei einer binären Klassifizierung, also nur zwei zu bestimmende Klassen, ist es sinnvoll ein ungerades k zu wählen. Dadurch kann es später bei der Mehrheitsentscheidung nicht zu einem Unentschieden kommen. Wenn es nur darum geht den nächsten Nachbarn zu bestimmen, also $k = 1$ ist, nennt man dies den *Nearest Neighbor* oder auch *1-Nearest Neighbor*.

Der Schritt zur Bestimmung der Konstante k wird dabei nur beim Erstellen des Algorithmus durchgeführt und danach diesem permanent übergeben.

2. Berechnen der Distanz anderer Objekte

Wird dem Algorithmus nach der Trainingsphase ein neues Objekt übergeben, wird es anhand seiner Merkmale in das Diagramm, das zuvor erstellte wurde, eingetragen. Von dem neu eingetragenen Objekt wird nun die Distanz zu allen anderen gespeicherten Objekten bestimmt. Zur Bestimmung dieses Abstandes können dabei verschiedene Verfahren angewendet werden. Die Populärsten sind dabei *Euklidischer Distanz / Abstand* und die *Manhattan-Metrik* oder auch *Mannheimer Metrik* genannt.

Die *Euklidische Distanz* ist die Luftlinie zwischen zwei Punkten und gibt damit die direkte Distanz wieder. Sie wird wie folgt berechnet:

$$\sqrt{\sum_{i=0}^n (x_i - y_i)^2}$$

n ist dabei die Anzahl der Merkmale, die jedes Objekt besitzt.

x & y sind die Position der beiden Objekte.

Die *Manhattan-Metrik* bestimmt die Distanz anhand der Summe der absoluten Differenz der einzelnen Koordinaten.

$$\sum_{i=0}^n |x_i - y_i|$$

n entspricht auch hier wieder der Anzahl der Merkmale.

x & y entsprechen ebenso wieder den Koordinaten zweier Objekte.

3. k nächste Nachbarn bestimmen

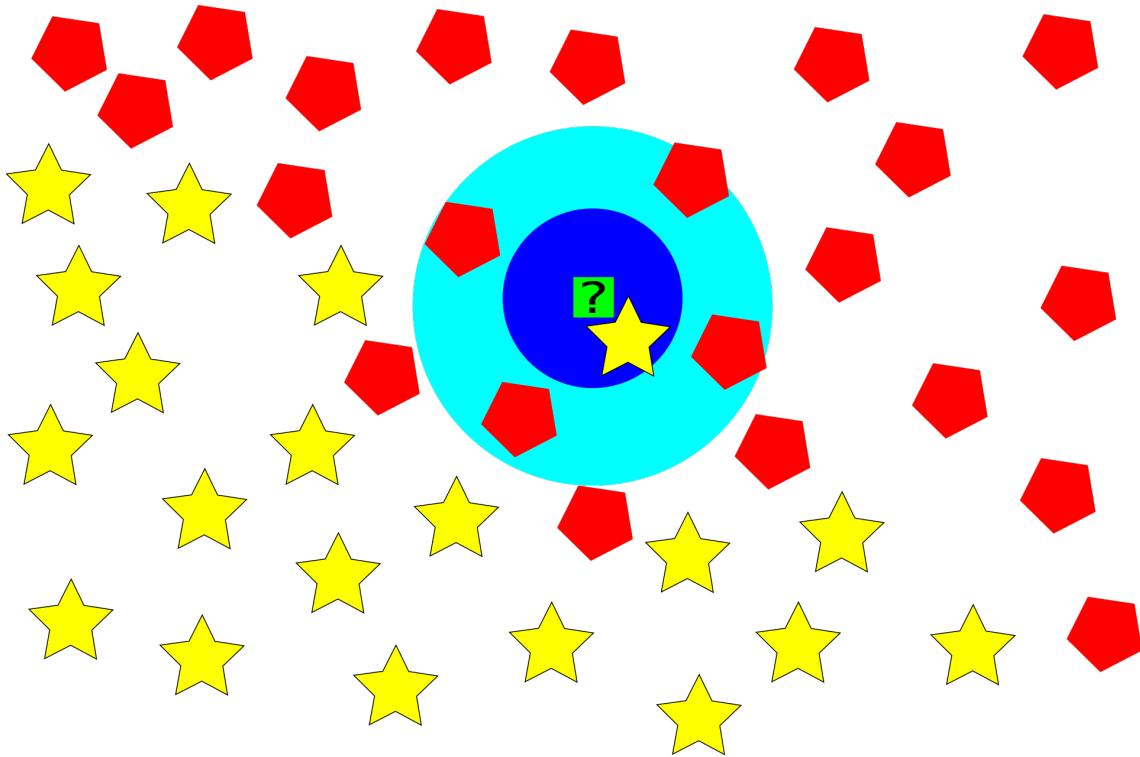
Mithilfe der Distanz können wir nun feststellen welche die nächsten Nachbarn zu unserem neuen Objekt sind. Dabei sind nur die k -nächsten von Interesse.

4. Bestimmung der Kategorie der Nachbarn

Im vierten Schritt stellen wir fest, zu welcher Kategorie jeder der k Nachbarn gehört und zählen diese. Bei anderen Varianten des *KNN* werden auch je nach Entfernung zum neuen Objekt Gewichtungen verteilt, dadurch erhält der nächste Punkt die höchste Gewichtung und je weiter entfernt, desto geringer das Gewicht. Hierdurch wird mehr auf räumliche Nähe zu einzelnen Punkten geachtet und weniger zu großen Ansammlungen.

5. Kategorie zuweisen

Als Letztes müssen wir dem neuen Punkt eine Kategorie zuweisen. Dabei wird die am häufigsten unter den Nachbarn vertretende Kategorie gewählt. Bei der gewichteten Variante wird die Kategorie mit dem größten Gewicht gewählt.[\[4\]](#)

Abbildung 2.1: Funktionsweise des *KNN*

Die Grafik 2.1 demonstriert den Entscheidungsprozess des *KNN*. Es werden immer nur die k nächsten Nachbarn beachtet. So wird das neue Objekt bei $k = 1$ der Kategorie *gelber Stern* zugeordnet, dargestellt durch den dunkelblauen Kreis. Beachtet man hingegen die fünf nächsten Nachbarn, also hat $k = 5$, hat man 1x *gelber Stern* und 4x *rotes Fünfeck*, dargestellt durch den hellblauen Kreis. Durch die Mehrheitsentscheidung wird das neue Objekt nun der Kategorie *rotes Fünfeck* zugewiesen.

Bewertung des *K-Nearest-Neighbor*

Der *KNN* wird wie die beiden anderen Algorithmen nach den Kriterien Komplexität, Genauigkeit, Präzision und Einfachheit der Umsetzung bewertet.

Da der *KNN* während des Trainings lediglich Daten speichert, besitzt er hierbei keine Komplexität. Beim Klassifizieren von Testdaten beträgt seine Komplexität $O(np)$, n entspricht dabei der Anzahl von Testdaten und p die Anzahl der Features.^[5]

Die drei Punkte Genauigkeit, Präzision und Einfachheit der Umsetzung können mit einem kurzen Testprogramm bewertet werden, das in 2.1 zu sehen ist. Hierfür gibt es in der Programmiersprache **Python** die Bibliothek **Scikit-learn**. Diese neben *KNN* noch einige weitere bereits fertig implementierte Machine Learning Algorithmen. Außerdem bietet sie Zugang zu verschiedenen Datensätzen, dazu gehört auch der **MNIST**-Datensatz.

Im Programm werden nun zunächst einmal die benötigten Funktionen und Klassen aus den verschiedenen Abhängigkeiten importiert. Anschließend werden die Daten vorbereitet. Dafür wird der Datensatz geladen und für die bessere Verarbeitung in ein **NumPy**-Array gewandelt und danach in die Trainings- und Testdaten aufgeteilt. Der *KNN* wird nun mit den Werten $k = 5$, $p = 2$ und der **minkowski-Metrik** initialisiert. Der Parameter p gibt dabei die Power der Metrik an. So verhält sich die *Minkowski-Metrik* für $p = 1$ wie die *Manhattan-Metrik* und bei $p = 2$ wie die *Euklidische Distanz*. Diesem *KNN* werden jetzt die Trainingsdaten übergeben und anschließend wird er mit den Testdaten getestet. Zum Schluss werden noch einige Ausgaben erzeugt, um die benötigten Informationen zu erhalten.

```
1 # Importieren der Abhängigkeiten
2 from sklearn import metrics
3 from sklearn.datasets import load_digits
4 from sklearn.neighbors import KNeighborsClassifier
5 from sklearn.model_selection import train_test_split
6
7 # Vorbereiten des Datensatzes
8 digits = load_digits()
9 n_samples = len(digits.images)
10 data = digits.images.reshape((n_samples, -1))
11
12 # Aufteilen in Trainings- und Testdaten
13 X_train, X_test, y_train, y_test = train_test_split(
14     data, digits.target, test_size=0.5, shuffle=False)
15
16 # Initialisieren des KNN
17 KNN_classifier = KNeighborsClassifier(
18     n_neighbors=5, p=2, metric='minkowski')
19
20 # Trainingsdaten speichern
21 KNN_classifier.fit(X_train, y_train)
22
23 # Klassifizieren der Testdaten
24 predicted = KNN_classifier.predict(X_test)
25
26 # Ergebnisse ausgeben
27 report = metrics.classification_report(y_test, predicted)
28 print("\nKlassifizierungsbericht %s:\n%s\n" %
29       (KNN_classifier, report))
30 score = KNN_classifier.score(X_test, y_test)
31 print("Genauigkeit des Algorithmus: ", score)
```

Quellcode 2.1: Pythoncode zum Testen des KNN

Die Ausgabe des Programmes gibt damit sofort die Präzision und Genauigkeit *KNN* an.

```

1      Klassifizierungsbericht KNeighborsClassifier():
2          precision    recall   f1-score   support
3
4          0         0.99    1.00     0.99     88
5          1         0.95    0.98     0.96     91
6          2         0.98    0.93     0.95     86
7          3         0.89    0.90     0.90     91
8          4         1.00    0.95     0.97     92
9          5         0.96    0.98     0.97     91
10         6         0.99    1.00     0.99     91
11         7         0.95    1.00     0.97     89
12         8         0.95    0.90     0.92     88
13         9         0.91    0.92     0.92     92
14
15 accuracy                      0.96     899
16 macro avg                     0.96     0.96     899
17 weighted avg                  0.96     0.96     0.96     899
18
19
20 Genauigkeit des Algorithmus: 0.9555061179087876

```

Quellcode 2.2: Testergebnisse des KNN

Der *KNN-Algorithmus* erzielt mit einer durchschnittlichen Präzision von 95,7% und einer Genauigkeit von 95,5% bereits sehr gute Ergebnisse.

Als Letztes muss nun noch die Komplexität der Umsetzung *KNN* bewertet werden. Durch die Verwendung der Bibliothek **Scikit-learn** ist es sehr einfach und auch schnell möglich ein Programm wie das obige zu schreiben. Darum bewerte ich dieses Kriterium für den *K-Nearest Neighbor* mit **5/5** Punkten.

Damit ist die Bewertung des *KNN* abgeschlossen und es folgt jetzt eine Erklärung und Bewertung der *Support Vector Machine*.

2.2.2 Support Vector Machine

Die *Support Vector Machine* ist ein Algorithmus zur Klassifizierung von Daten in verschiedene Klassen. Sie wurde in den 1990er Jahren innerhalb der Informatiker-Community entwickelt und gewinnt seitdem immer weiter an Popularität. Support Vector Machines ([SVMs](#)) können in vielen verschiedenen Szenarien eingesetzt werden und werden zu den besten Klassifizierern gezählt. Bevor die SVM entwickelt wurde, gab es verschiedene Vorgängertechnologien aus denen am Ende die heutige *Support Vector Machine* hervorgegangen ist. Diese Algorithmen sind zum Einen der *Maximal Margin Classifier* und der *Support Vector Classifier*, der eine Weiterentwicklung des zuvor Genannten darstellt. Die *Support Vector Machine* ist wiederum eine Erweiterung des *Support Vector Classifier*. Jeder dieser Klassifizierer wird meist lose unter dem Begriff *Support Vector Machine* zusammengefasst, müssen jedoch klar unterschieden werden.[\[6, S. 337\]](#)

Die Terminologie im Bereich der *Support Vector Machine* ist nicht ganz klar und kann sich in verschiedenen Veröffentlichungen unterscheiden. Eine andere Differenzierung der Vorgängertechnologien ist die Einteilung in Lineare Klassifizierer und nicht-lineare Klassifizierer. Auch zu erwähnen ist der Lagrangische *Support Vector Machine* Algorithmus, mit dem iterativ SVMs trainiert werden können.[\[7, S. 207\]](#) In dieser Arbeit verwenden wir die zuvor genannte Einteilung in *Maximal Margin Classifier*, *Support Vector Classifier* und *Support Vector Machine*. An ihr lässt sich die kontinuierliche Weiterentwicklung der Algorithmen am besten verfolgen und am übersichtlichsten darstellen. Außerdem wird die Chronologie der hinzugekommenen Konzepte deutlicher.

Beginnen wir mit der einfachsten Form dieser Klassifizierer, dem *Maximal Margin Classifier*. Dieser wird durch eine sogenannte *Hyperplane* definiert. In einem p -dimensionalen Raum ist eine *Hyperplane* ein flacher affiner Subraum der Dimension $p - 1$. Bei zwei Dimensionen wäre eine *Hyperplane* ein flacher 1-dimensionaler Subraum oder in anderen Worten eine Gerade. In einem 3-dimensionalen Raum ist die *Hyperplane* demnach eine Ebene. *Hyperplanes* in höherdimensionalen Räumen sind schwer für den Menschen vorstellbar, folgen jedoch ebenfalls der Definition. Die mathematische Definition einer *Hyperplane* im 2-dimensionalen Raum lautet:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$$

mit den Parametern β_0 , β_1 und β_2 . Es lässt sich erkennen, dass die Definition mit der einer Geraden übereinstimmt und damit unserer oben beschriebenen Definition einer *Hyperplane* folgt. In einem p -dimensionalen Raum wird die Gleichung einer *Hyperplane* wie folgt erweitert:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p = 0$$

Eine *Hyperplane* teilt den p -dimensionalen Raum in zwei Hälften. Erfüllt ein Punkt $X = (X_1, X_2, \dots, X_p)$ des p -dimensionalen Raumes nicht die obige Gleichung, liegt er entweder auf der einen Seite

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p > 0$$

oder auf der anderen Seite der *Hyperplane*

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p < 0$$

Abbildung 2.2 zeigt beispielhaft eine solche *Hyperplane*, die durch die Gleichung $1+6X_1+4X_2 = 0$ definiert ist. Sie trennt den Raum in einen blauen Bereich, wo alle Punkte $1+6X_1+4X_2 > 0$ erfüllen und eine rote Seite, auf der für alle Punkte $1+6X_1+4X_2 < 0$ gilt. [6, S. 338]

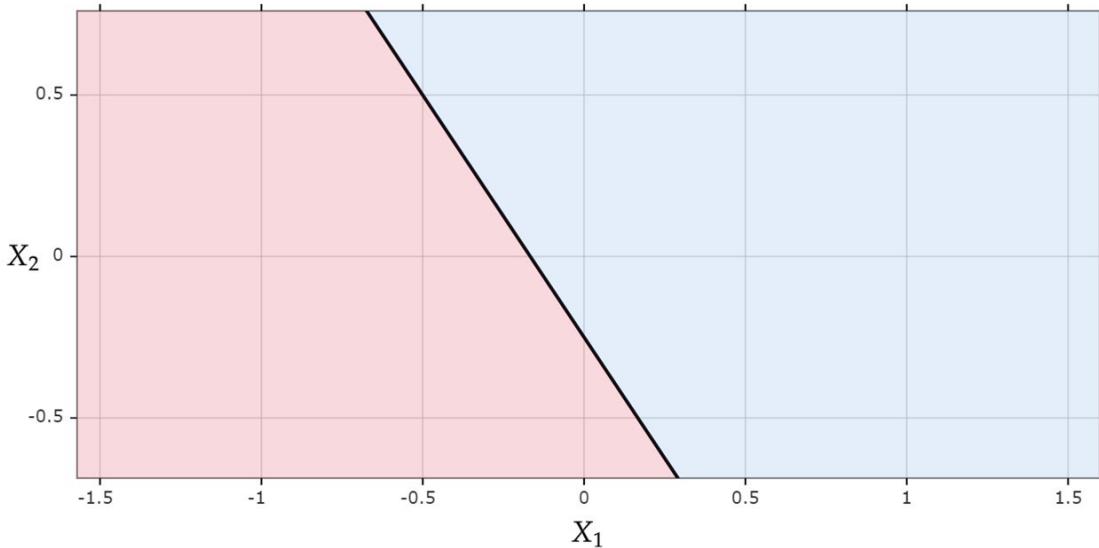


Abbildung 2.2: Beispiel für eine *Hyperplane*

Wir haben nun eine Matrix der Trainingsdaten \mathbf{X} der Form $n \times p$, die aus n Trainingsobjekten innerhalb eines p -dimensionalen Raumes besteht.

$$x_1 = \begin{pmatrix} x_{11} \\ \vdots \\ x_{1p} \end{pmatrix}, \dots, x_n = \begin{pmatrix} x_{n1} \\ \vdots \\ x_{np} \end{pmatrix}$$

Diese Trainingsobjekte sind definiert als Vektoren der Länge p , die die einzelnen beobachteten *Features* $x^* = (x_1^* \dots x_p^*)^T$ enthalten und werden jeweils einer von zwei Klassen zugeordnet $y_1, \dots, y_n \in \{-1, 1\}$. Wenn die *Hyperplane* die Trainingsobjekte perfekt anhand ihrer Labels trennt, dann spricht man von einer *separierenden Hyperplane*. Alle Objekte aus der einen Klassen fallen auf die linke Seite der *Hyperplane*. Während die Trainingsobjekte mit dem

anderen Label auf der rechten Seite der *Hyperplane* liegen. Eine *separierende Hyperplane* lässt sich mathematisch definieren durch folgende zwei Gleichungen:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p > 0 \text{ if } y_i = 1$$

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p < 0 \text{ if } y_i = -1$$

für alle $i = 1, \dots, n$. Diese *separierende Hyperplane* kann nun als Klassifizierer eingesetzt werden. Um ein unbekanntes Testobjekt nun zu klassifizieren, muss nur geschaut werden auf welcher Seite der *Hyperplane* das Objekt liegt. Liegt das Testobjekt auf der positiven Seite der *Hyperplane* wird ihm das Label 1 zugewiesen. Wenn es auf der negativen Seite liegt, wird das Testobjekt der Klasse -1 zugeordnet. Je weiter entfernt das Testobjekt von der *separierenden Hyperplane* liegt, desto sicherer können wir uns sein, dass die Klassenzuweisung korrekt ist. Je kleiner der Abstand zur *Hyperplane* wird, desto unsicherer ist, ob das Testobjekt der richtigen Klasse zugeordnet worden ist. Der Abstand der einzelnen Testobjekte zur *separierenden Hyperplane* zeigt uns mit welcher Gewissheit das SVM die richtige Klassifizierung getroffen hat.[6, S. 339 - 141]

Beim *Maximal Margin Classifier* wird die oben genannte Methode der *separierenden Hyperplane* eingesetzt, um Daten klassifizieren zu können. Wenn die Daten durch eine *separierende Hyperplane* getrennt werden können, gibt es unendlich viele Möglichkeiten, wie diese liegen kann. Für bestmögliche Ergebnisse beim Klassifizieren muss deshalb die optimale *separierende Hyperplane* gefunden werden. Die natürliche Wahl ist es die 'Mittigste' der *separierenden Hyperplanes* zu verwenden. Diese wird auch *Maximal Margin Hyperplane* genannt. Mit Hilfe der Trainingsdaten muss eine *separierende Hyperplane* gefunden werden, die so weit wie möglich von den nächsten Trainingsobjekten entfernt liegt.[8, S. 1565f.]

Die Abbildung 2.3 zeigt zehn verschiedene Trainingsobjekte, die alle durch zwei Variablen definiert sind und einem der beiden Label $\{\text{blau}, \text{rot}\}$ zugeordnet sind. Außerdem sind drei unterschiedliche *Hyperplanes* abgebildet, die alle *separierende Hyperplanes* im Bezug auf unsere Trainingsdaten sind. Die Auswahl der „perfekten“ *Hyperplane* für die SVM ist damit ein mathematisches Optimierungsproblem. Weitere Informationen zu dem Trainingsmodell einer SVM sind in dem Werk von Suthaharan zu finden.[7, S. 210ff.]

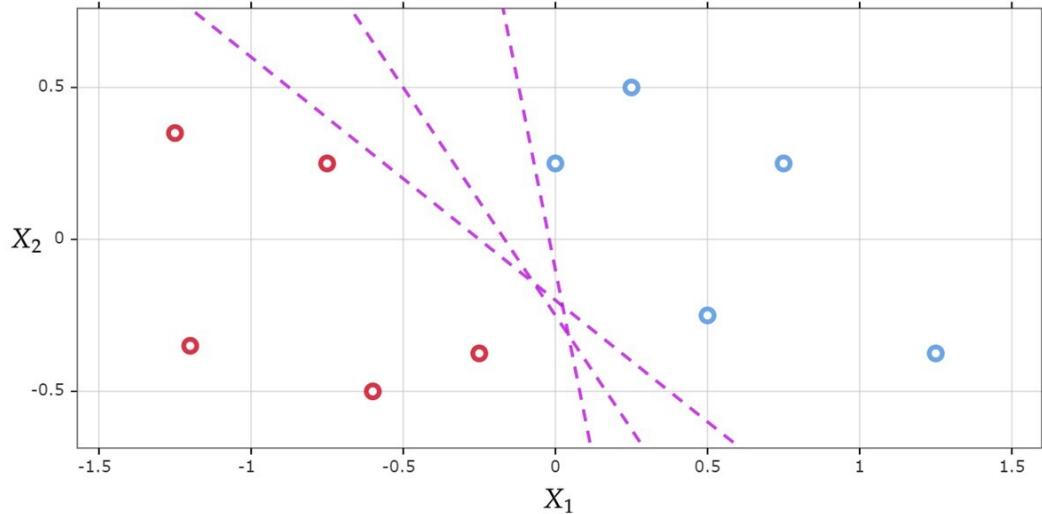


Abbildung 2.3: Beispiele für *separierende Hyperplanes*

Der Abstand zwischen der *separierenden Hyperplane* und den nächstgelegenen Trainingsobjekten beider Labels wird als *Margin* bezeichnet. Die *Hyperplane*, deren *Margin* am größten ist, wird daraus folgend auch *Maximal Margin Hyperplane* genannt. Die Trainingsobjekte, die am nächsten zu der *Hyperplane* liegen und damit auch denselben Abstand zu dieser haben, werden als die *Stützvektoren* bezeichnet. Sie „stützen“ die *Maximal Margin Hyperplane*. Die *Stützvektoren* sind demnach die einzigen Trainingsobjekte, die die Position der *Hyperplane* überhaupt beeinflussen.[6, S. 341] Mit Hilfe der *Maximal Margin Hyperplane* kann der *Maximal Margin Classifier* nun neue unbekannte Testdaten klassifizieren, je nach dem auf welcher Seite sie liegen. Je größer dabei die *Margin* der zugrunde liegenden *Hyperplane*, desto erfolgreicher der Klassifizierer.[8, S. 1566]

Eine solche *Maximal Margin Hyperplane* ist in 2.4 abgebildet. Die abgebildeten Trainingsdaten sind dieselben wie auch in Abbildung 2.3. Es wurde die „mittigste“ *separierende Hyperplane* gefunden, die die Trainingsdaten anhand ihrer Labels trennen kann. Außerdem zu sehen, sind die am nächsten gelegen Trainingsobjekte auf beiden Seiten der *Hyperplane*, die als *Stützvektoren* dienen. Die *Margin* wird visualisiert, durch die beiden Geraden, die parallel zur *separierenden Hyperplane* liegen und durch die beiden *Stützvektoren* verlaufen. Der Abstand dieser beiden Geraden zu der *Hyperplane* ist die *Margin*.

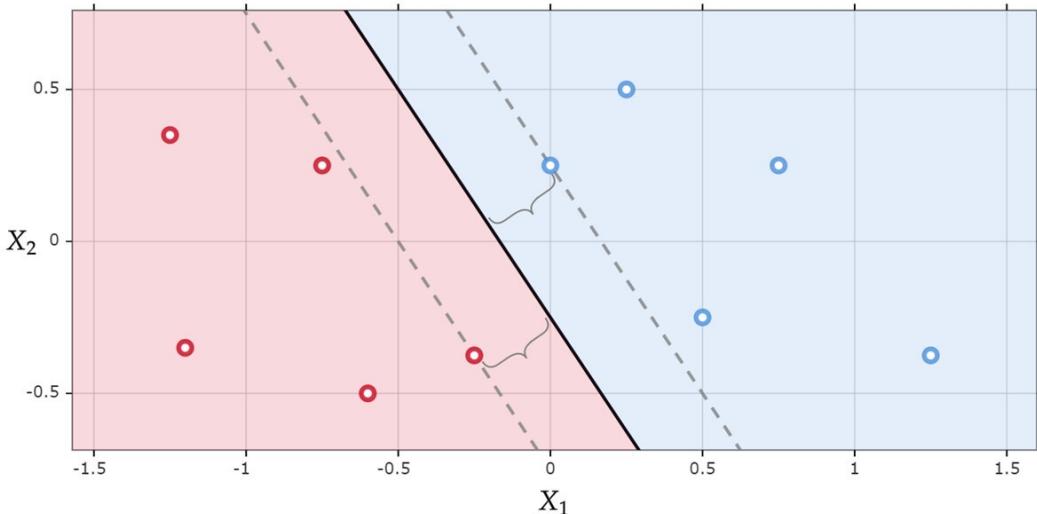


Abbildung 2.4: Beispiel für eine *Maximal Margin Hyperplane*

Je nach Trainingsdaten gibt es Szenarien in denen keine *separierende Hyperplane* existiert, die die Daten perfekt anhand ihrer Labels trennen kann oder die *Hyperplane* liegt nicht optimal, um eine effiziente Klassifizierung durchführen zu können, da die Stützvektoren ungünstig liegen. Der *Maximal Margin Classifier* hat eine hohe Sensibilität für einzelne Trainingsobjekte und kann demnach leicht übertrainiert werden. Es können zum Beispiel Trainingsobjekte existieren deren *Features* nicht den Normen einer Klasse entsprechen und damit eine Anomalie darstellen. Das besagte Trainingsobjekt würde dann auf der falschen Seite der zuvor gefundenen *Hyperplane* liegen. Deswegen müssen, um eine bessere Klassifizierung und eine höhere Robustheit für einzelne Trainingsobjekte zu gewährleisten, auch nicht perfekt *separierende Hyperplanes* in Betracht gezogen werden.[6, S. 343 - 345]

Dies wird vom *Support Vector Classifier* umgesetzt. Er wird auch als *Soft Margin Classifier* bezeichnet. Wie der Name schon sagt, besitzt dieser eine 'weiche' *Margin*. Der *Support Vector Classifier* erweitert den bereits besprochenen *Maximal Margin Classifier* um eine 'weiche' *Margin*, die von einem Nutzer-definierten Parameter abhängig ist. Der Nutzer muss selbst entscheiden, wie durchlässig die *Margin* sein soll und wie vielen Trainingsobjekten erlaubt werden soll die *Margin* zu durchdringen, ohne deren finale Ausrichtung zu beeinflussen. Wie dieser Parameter gesetzt werden muss, hängt von den Trainingsdaten und dem gewünschten Ergebnis ab und muss durch Ausprobieren ermittelt werden. Beim Setzen dieses Parameters muss zwischen Durchlässigkeit der *Hyperplane* und der Breite der *Margin* abgewägt werden. Durch diese 'weiche' *Margin* werden vereinzelt Testobjekte mit abnormalen Charakteristiken der falschen Klasse zugeordnet. Für die meisten Testobjekte steigt jedoch die Genauigkeit der Klassifizierung und aufgrund der 'weichen' *Margin* ist es leichter eine *separierende Hyperplane* zu finden.[8, S. 1566]

Trotzdem gibt es immer noch Szenarien bei denen die Trainingsdaten nicht durch einen *Support Vector Classifier* getrennt werden können. In manchen Fällen sind die Trainingsdaten nicht linear zu trennen. In diesen Fällen muss der *Feature-Raum* erweitert werden. Hierfür wurde aus dem *Support Vector Classifier* die *Support Vector Machine* entwickelt. Dabei wurden sogenannte *Kernel-Funktionen* hinzugefügt. Mit Hilfe von quadratischen, kubischen oder weiteren Polynomfunktionen höherer Grade wird der *Feature-Raum* der Trainingsobjekte erweitert. Auf Basis der bisherigen *Features* werden so mathematisch neue *Features* berechnet, die dann womöglich durch eine *Hyperplane* zu trennen sind, jedoch von den ursprünglichen *Features* abhängig sind. Der *Feature-Raum* wird so gesehen von den *Kernel-Funktionen* um eine oder mehrere Dimensionen erweitert. Auch bei diesen *Kernel-Funktionen* gibt es keine genaue Regel und die für die Trainingsdaten passende *Kernel-Funktion* muss durch Ausprobieren gefunden werden. So können nun auch nicht-lineare Daten klassifiziert werden. [8, S. 1566f.][7, S. 224 - 227]

Die nachfolgende Abbildung 2.5 zeigt zwei Beispiele hypothetischer Trainingsdaten, die nicht linear zu trennen sind. Der *Feature-Raum* wurde mit Hilfe einer *Kernel-Funktion* erweitert und es konnte in einem höherdimensionalen Raum eine *separierende Hyperplane* gefunden werden, die die Trainingsobjekte anhand ihrer Labels trennen kann. Die zwei Grafiken zeigen nun wie diese höherdimensionale *Hyperplane* in einem 2-dimensionalen Raum aussehen würde und wie sie die Daten anhand ihrer Klasse aufteilt.

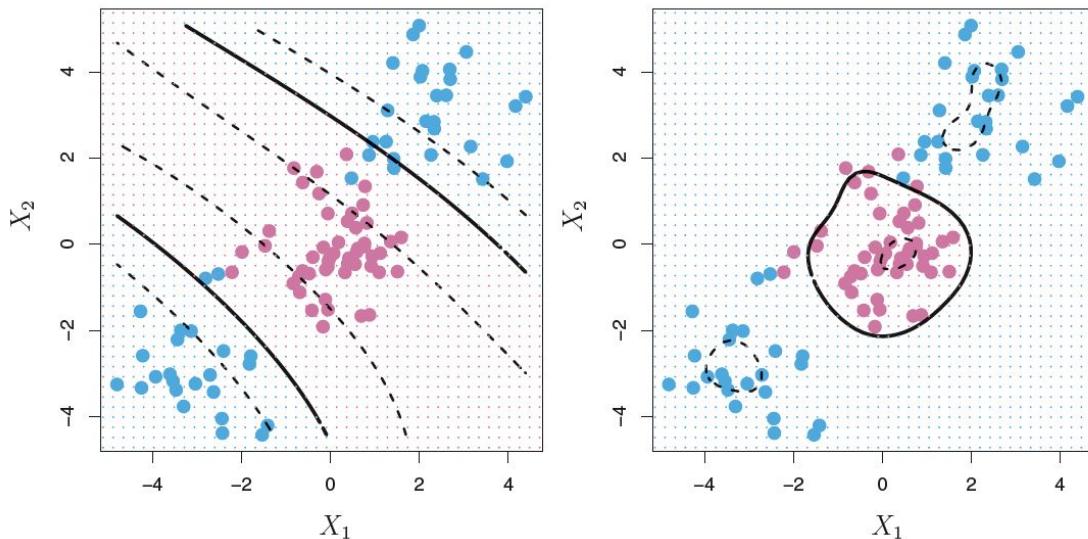


Abbildung 2.5: *Hyperplanes* bei nicht-linearen Daten [6, S. 353]

Mit Hilfe der *Support Vector Machine* können wir nun einen Großteil an Datensätzen klassifizieren. Eine *separierende Hyperplane* kann die Trainingsdaten jedoch immer nur in zwei Klassen einteilen. Wie können dann handgeschriebene Ziffern und Zahlen durch eine *Support Vector Machine* erkannt werden? Es gibt zwei Möglichkeiten dies zu bewerkstelligen, wenn $K > 2$

Klassen existieren. Die erste Methode ist die 1-gegen-1-Klassifizierung. Hier wird für jede Paarung der Klassen $\binom{K}{2}$ eine SVM erzeugt. Bei einer Klassifizierung eines neuen Testobjektes wird die Klasse ausgewählt, zu der das Objekt bei den $\binom{K}{2}$ paarweisen Klassifizierern am häufigsten zugewiesen wurde. Die Alternative zu dieser Methode ist die 1-gegen-alle-Klassifizierung. Hier werden K SVMs erstellt. Eine SVM für jede Klasse und die $K - 1$ restlichen Klassen. Ein unbekanntes Testobjekt wird dann der Klasse zugeordnet, bei der die SVM sich am sichersten bei der Zuordnung ist. So kann die *Support Vector Machine* auch Testobjekte verschiedener Klassen unterscheiden.[6, S. 355f.][8, S.1567]

Bewertung der *Support Vector Machine*

Für einen fairen Vergleich zwischen den einzelnen *Machine Learning*-Algorithmen muss die *Support Vector Machine* nach den bereits festgelegten Bewertungskriterien: Komplexität, Genauigkeit, Präzision und Einfachheit der Umsetzung beurteilt werden. Die mathematische Komplexität des *Support Vector Machine*-Algorithmus lässt sich nachschlagen. Das Trainieren einer SVM hat eine Komplexität von $O(n^2p + n^3)$ mit n als Anzahl der Traingsobjekte und p als Anzahl der *Features*. Bei dieser Komplexität wird angenommen, dass die *Support Vector Machine* eine *Kernel*-Funktion $K(x_i, x_j)$ besitzt. Davon ausgehend, dass die häufigsten *Kernel*-Funktionen eine Komplexität von $O(p)$ besitzen, können wir auf die obige Komplexität für die SVM schließen. Das ist bei Gaußschen, Polynom- und Sigmoidfunktionen der Fall, muss jedoch nicht für jede *Kernel*-Funktion gelten. Die Komplexität bei dem Klassifizieren der Testdaten entspricht dann $O(n_{sv}p)$. Hier ist n_{sv} die Anzahl der Stützvektoren.[5]

Um die Genauigkeit und Präzision einer *Support Vector Machine* bewerten zu können, soll ein kurzes Testprogramm geschrieben werden. Hierfür wurde die Programmiersprache **Python** verwendet, die mit **SciKit-learn** bereits eine Bibliothek für die Anwendung verschiedener *Machine Learning*-Algorithmen besitzt. Das fertige Programm ist in 2.3 zu sehen. Zuerst werden die benötigten Funktionen aus der **SciKit-learn**-Bibliothek importiert. Dann wird der 1797-große Datensatz geladen, der die geschriebenen Ziffern 0, ..., 9 enthält und die Daten mit Hilfe eines **NumPy**-Arrays vorbereitet. Danach werden die Daten halbiert und in einen Trainingsdatensatz, sowie in einen Testdatensatz aufgeteilt. Mit **SciKit-learn** lässt sich dann in zwei Zeilen ein SVM-Objekt erstellen und mit dem Trainingsdatensatz trainieren. Danach werden die Testdaten in die SVM gegeben und am Ende in einem Klassifizierungsbericht ausgewertet.

```

1 # Importieren relevanter Bibliotheken
2 from sklearn import svm
3 from sklearn import metrics
4 from sklearn.datasets import load_digits
5 from sklearn.model_selection import train_test_split
6
7 # Laden des SciKit-learn-Datensatzes
8 digits = load_digits()
9
10 # Vorbereiten der Daten
11 n_samples = len(digits.images)
12 data = digits.images.reshape((n_samples, -1))
13
14 # Aufteilen der Daten in Trainings- & Testdaten
15 X_train, X_test, y_train, y_test = train_test_split(
16     data, digits.target, test_size=0.5, shuffle=False)
17
18 # Trainieren der SVM mit den Trainingsdaten
19 svm_classifier = svm.SVC(gamma=0.001)
20 svm_classifier.fit(X_train, y_train)
21
22 # Klassifizieren der Testdaten
23 predicted = svm_classifier.predict(X_test)
24
25 # Ausgeben der Ergebnisse
26 print("\nKlassifizierungsbericht %s:\n%s\n" % (svm_classifier, metrics.
27     classification_report(y_test, predicted)))
28 print("\nGenauigkeit: ", svm_classifier.score(X_test, y_test))

```

Quellcode 2.3: Pythoncode zum Testen der SVM

Aus dem Klassifizierungsbericht in 2.4 kann direkt die Präzision und Genauigkeit dieser *Support Vector Machine* herausgelesen werden. Die *Support Vector Machine* aus unserem Test hat eine durchschnittliche Präzision von 0.97, sowie eine gerundete Genauigkeit von ebenfalls 0.97.

```

1 Klassifizierungsbericht SVC(gamma=0.001):
2     precision    recall   f1-score   support
3
4      0         1.00    0.99    0.99    88
5      1         0.99    0.97    0.98    91
6      2         0.99    0.99    0.99    86
7      3         0.98    0.87    0.92    91
8      4         0.99    0.96    0.97    92
9      5         0.95    0.97    0.96    91
10     6         0.99    0.99    0.99    91
11     7         0.96    0.99    0.97    89
12     8         0.94    1.00    0.97    88
13     9         0.93    0.98    0.95    92
14
15 accuracy          0.97    899
16 macro avg       0.97    0.97    899
17 weighted avg   0.97    0.97    899
18
19 Genauigkeit: 0.9688542825361512

```

Quellcode 2.4: Testergebnisse der SVM

Damit ist bereits eine Mehrheit der Kriterien zur Bewertung der SVM abgedeckt. Als Letztes muss nun die Einfachheit der Umsetzung bewertet werden. Es spricht für die *Support Vector Machine*, dass uns die gesamte Umsetzung des Algorithmus bereits abgenommen wurde und alle benötigten Funktionen durch die **Python**-Bibliothek **SciKit-learn** bereitgestellt wurden. Die *Support Vector Machine* ist damit sehr einfach für unser Projekt zu verwenden. Wie bereits im obigen Beispielcode 2.3 zu sehen, braucht es nur drei Zeilen Code um eine SVM zu initialisieren, zu trainieren und neue Testdaten zu klassifizieren. Deshalb gebe ich der *Support Vector Machine* eine Punktzahl von 5/5. Damit ist die Bewertung dieses *Machine Learning*-Algorithmus abgeschlossen. Als Nächstes folgt eine Erläuterung der *Convolutional Neural Networks* und deren Beurteilung.

2.2.3 Convolutional Neural Networks

Die letzte künstliche Intelligenz, die es in diesem Vergleich zu untersuchen gilt, ist die des Convolutional Neural Networks ([CNNs](#)).

Die CNN wird, ähnlich wie die SVM als ein Konstrukt zum Klassifizieren von Bildern genutzt, jedoch unterscheiden sie sich in der Herangehensweise das Problem zu lösen. Anders als die SVM handelt es sich bei der CNN um keine algorithmische Umsetzung, sondern um eine vordefinierte Menge an Anweisungen, die zur Lösung eines spezifischen Problems führen. Aufgrund der vordefinierten Menge an Anweisungen ist ein CNN jedoch auf Probleme beschränkt, welche wir verstehen und lösen können. Dies hat zur Folge, dass kein Ansatz dem anderen überzuordnen ist, sondern die Art des Problems definiert, ob konventionelle Algorithmen oder neuronale Netzwerke besser geeignet sind. [9]

Um nun den Aufbau eines CNN zu verstehen, betrachten wir zunächst ein normales Artificial Neural Network ([ANN](#)) oder *Neural Network*. Beide Terminologien werden in der Literatur kommutativ genutzt. Um den Unterschied zu den biologischen neuronalen Netzwerken zu verdeutlichen, wird fortlaufend in dieser Arbeit ANN als Terminologie genutzt.

Artificial Neural Network

Im Jahr 1943 entwickelte der Neurophysiologe Warren McCulloch und ein junger Mathematiker Walter Pitts das erste Model eines ANN, das in der Studie „The Logical Calculus of the Ideas Immanent in Neverous Activiy“ veröffentlicht wurde. Dieses war in der Lage, einfache logische Operationen durchzuführen. Im Laufe der Historie wurde das Konstrukt ANN immer weiterentwickelt und hat in der Informatiker-Community an Ansehen erlangt, bis die Technologie an ihre damaligen Grenzen gestoßen ist und die Technologie zunächst vernachlässigt wurde. Heutzutage sind ANN einer der wichtigsten Bestandteile in der Erforschung der Künstlichen Intelligenz und des maschinellen Lernens, welches auf die enorme Innovation in der Computerindustrie und der Leistungsverbesserung der Rechner zurückzuführen ist. [9]

Ein ANN ist im Aufbau ähnlich zu einem primitiven biologischen Nervensystem. Es besteht aus einer gewissen Anzahl an Knoten, die Neuronen repräsentieren und miteinander verbunden sind. Die miteinander verbunden Knoten geben Informationen an den jeweils nächsten Knoten weiter, nachdem ihre eingegangene Information verarbeitet wurde. In einem ANN sind die Knoten in verschiedene Schichten aufgeteilt, wobei die Anzahl der Knoten nicht festgelegt ist. Es können Schichten mit einem Knoten erstellt werden oder mit mehreren Tausenden. Jede dieser Schichten dient einem genau definierten Zweck im Netzwerk. Um den Zweck genauer zu definieren, werden die Schichten in drei Gattungen unterteilt, in die Eingabeschicht, die versteckten Schichten und die Ausgabeschicht. Die Eingabeschicht nimmt in der Regel einen

mehrdimensionalen Eingabevektor entgegen. Dieser wird in der Eingabeschicht verarbeitet und das Ergebnis wird an die Knoten der nächsten Schicht verteilt. Die danach kommende Schicht verarbeitet die Informationen, die sie bekommen hat und verteilt diese dann wieder an die nächsten Knoten in der folgenden versteckten Schicht oder an die Ausgabeschicht, wenn keine weitere verdeckte Schicht folgt. Die Knoten innerhalb einer Schicht sind nicht miteinander verknüpft, sie sind nur mit Knoten aus der vorherigen und danach kommenden Schicht verbunden. Wenn alle Knoten mit allen Konten aus der vorangegangen und danach kommenden Schicht verbunden sind, bezeichnet man das ANN an diesem Punkt als *fully-connected*. Ein solches *fully-connected ANN* ist in Abbildung 2.6 zu erkennen. Das Netzwerk verfügt beispielsweise über vier Schichten, eine Eingabeschicht, eine Ausgabeschicht und zwei versteckten Schichten. In der Abbildung werden die Knoten als Kreise dargestellt und die Verbindungen als gerichtete Linien. Die versteckten Schichten haben zum Beispiel vier Knoten. [10]

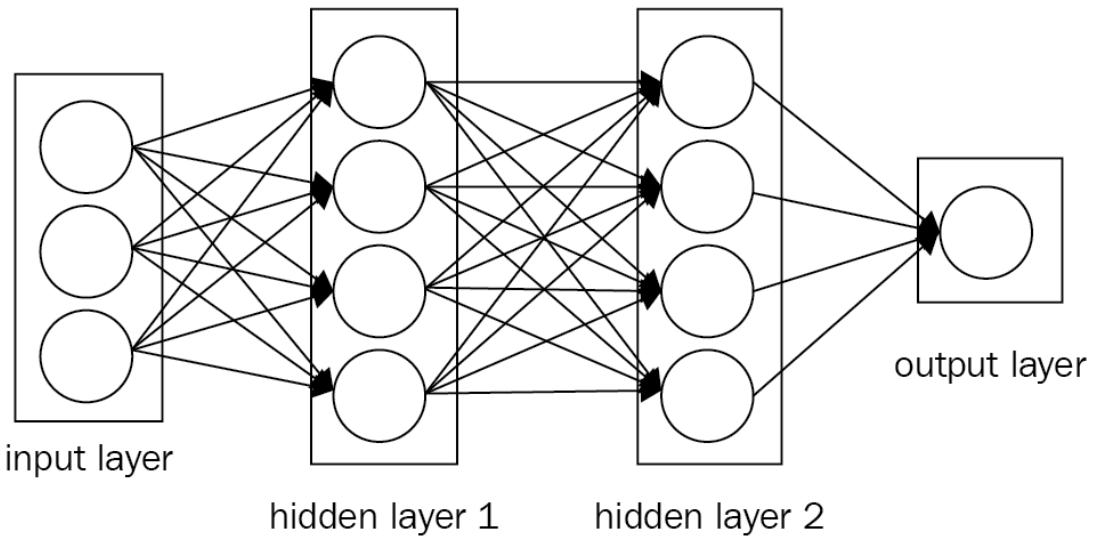


Abbildung 2.6: Ein *fully-connected ANN* mit zwei versteckten Schichten [11]

Um nun genauer nachzuvollziehen, wie ein Knoten in einem ANN operiert, gehen wir genauer auf die mathematische Definition eines Knotens ein.

Die Verarbeitung der Eingabevektoren erfolgt durch die Anwendung der sogenannten Aktivierungsfunktion a . Die Aktivierungsfunktion ist die Summe der gesamten Eingabevektoren $[x_1, \dots, x_n]$ multipliziert mit einer Gewichtsmatrix $[w_1, \dots, w_n]^T$, wobei n die Anzahl der vorherigen Knoten ist. Mathematisch ist sie wie folgt definiert:

$$a = \sum_{i=1}^n x_i w_i$$

In der Abbildung 2.7 wird die Aktivierungsfunktion a dargestellt. Es ist erkennbar, wie jeder Eingabevektor x_i mit einem Gewicht w_i multipliziert wird und anschließend die Summe aller Ergebnisse gebildet wird. Das Ergebnis der Aktivierungsfunktion a ist die Ausgabe eines Knotens und wird anschließend von den folgenden Knoten in der nächsten Schicht als Eingabevektor verarbeitet.

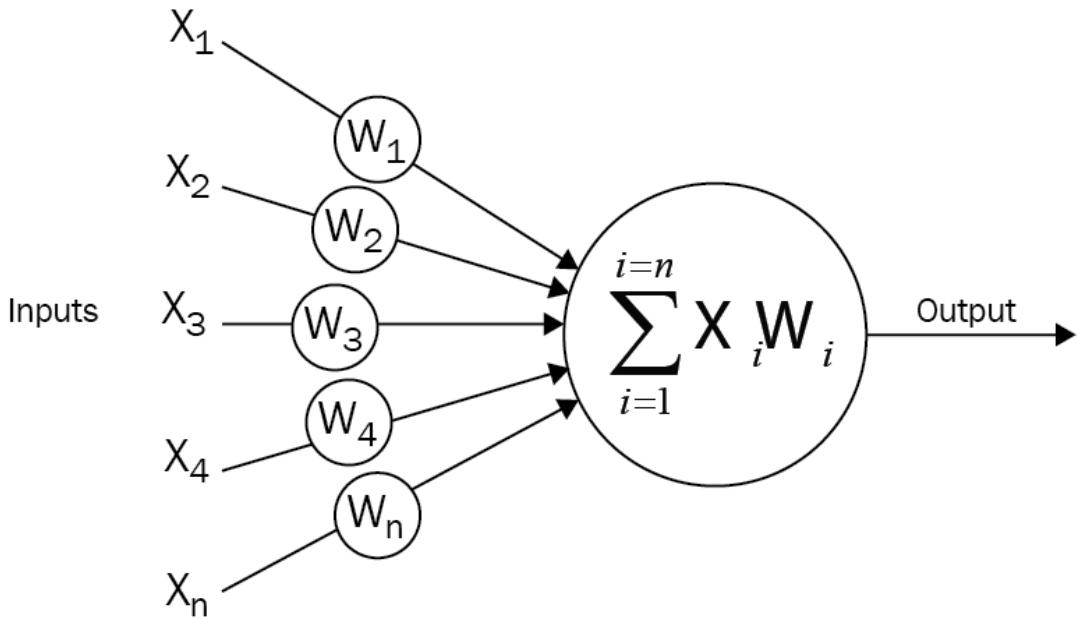


Abbildung 2.7: Bildliche Darstellung der *Aktivierungsfunktion* [11]

Mit dieser Funktion können wir nachvollziehen, wie jeder einzelne Knoten seine Entscheidung trifft.[12]

Zusätzlich können wir eine *Verlustfunktion* für das Netzwerk definieren, um die Genauigkeit der Entscheidungen zu messen. Bei den ANN existieren zwei verschiedene Arten von Verlustfunktionen, die Klassifizierungs-Verlustfunktion und die Regression-Verlustfunktion. Die Klassifizierung-Verlustfunktion wird genutzt, wenn ein Netzwerk als Klassifizierer eingesetzt wird, wie es in diesem Test der Fall ist. Die Regression-Verlustfunktion wird bei Anwendungen der Vorhersage von bestimmten Werten verwendet. Unter den zwei Arten der Verlustfunktion für ANN existieren wiederum verschiedene mathematische Funktionen, mit denen sich der Wert je nach Anwendung berechnen lässt.[13]

Im Folgenden wird die sogenannte *Kreuzentropie* oder auch *log-loss* genannt als Verlustfunktion verwendet. Diese Verlustfunktion wird bei binärer Klassifikation eingesetzt und berechnet den Verlustwert mit Hilfe eines booleschen Wertes und dem errechneten Entscheidungswert des ANN. Der boolesche Wert wird durch 0 oder 1 ausgedrückt, dieser sagt aus, ob das zu klassifizierende Objekt richtig erkannt wurde oder nicht. In der Abbildung 2.8 wird die Kreuzentropie dargestellt. Anhand dieser Darstellung ist zu erkennen, wie der Verlustwert abhängig vom Entscheidungswert

des ANN ist. Wenn der Entscheidungswert sich der 1 annähert, verringert sich der Wert der Kreuzentropie. Für eine Klassifikation von zwei Klassen wird die Kreuzentropie wie folgt berechnet:

$$-(y \times \log(p) + (1 - y) \times \log(1 - p))$$

- y ist der boolesche Wert, der angibt, ob das zu klassifizierende Objekt richtig erkannt wurde.
- p ist der errechnete Entscheidungswert des ANN.

Im Fall der Zahlenerkennung existiert für jede Zahl eine Klasse. Das Prinzip der Kreuzentropie lässt sich auf eine beliebige Anzahl von Klassen erweitern, in dem für jedes Label pro Beobachtung der Verlustwert berechnet wird und letztendlich summiert wird.

$$-\sum_{c=1}^M y_{o,c} \times \log(p_{o,c})$$

- c ist das Klassenlabel.

[14]

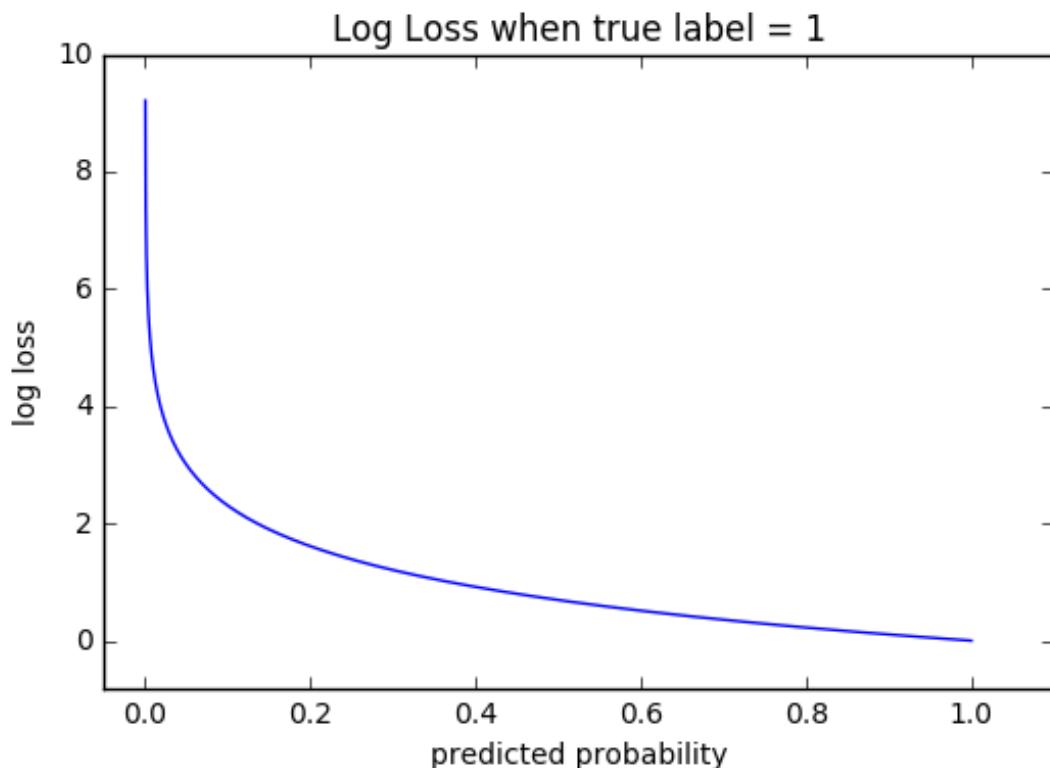


Abbildung 2.8: Graph einer *Kreuzentropie* [15]

Von Artificial Neural Network zu Convolutional Neural Network

Wie im Vorfeld angeführt, handelt es sich bei den CNN um eine Spezialisierung eines ANN. Diese Spezialisierung erweitert und spezifiziert Aspekte eines ANNs, welche im Folgenden genauer erläutert werden.

Da wir das CNN für die Klassifikation von Bildinhalten verwenden, muss die Eingabeschicht für diesen Datentypen vorbereitet werden. Die Besonderheit hierbei liegt darin, dass die Knoten zusätzlich zu den Schichten in drei Dimensionen aufgeteilt werden. Diese Aufteilung bezieht sich jedoch nur auf die Verknüpfungen zu den Knoten in der nächsten Schicht. Die Dimensionen sind für ein Bild die Höhe und Breite. Eine zusätzliche dritte Dimension ist die sogenannte Tiefe, welche Auskunft über die Anzahl der folgenden verknüpften Knoten bietet.

Die Architektur der CNN erweitert das Prinzip der Ein/Ausgabeschicht und den versteckten Schichten insoweit, als dass die versteckten Schichten in weitere Kategorien eingeteilt werden können. Bei einem CNN gibt es zusätzlich zu der Ein/Ausgabeschicht, die Filter-Schichten (engl. Convolutional Layer), Aggregations-Schichten (Pooling Layer) und die vollständig verbundenen Schichten (engl. Fully Connected Layer, Dense Layer). Bei der Ausgabeschicht handelt es sich ebenfalls um eine vollständig verbundene Schicht. [10]

Die einzelnen Aufgaben und Eigenarten der verschiedenen Schichten werden im Folgenden behandelt.

Input Layer

In der Eingabeschicht werden die Bilddaten gespeichert und in einer dreidimensionalen Matrix dargestellt.[11]

Filter-Schicht

Die Filter-Schicht, auch häufig als Feature-Extraction-Layer bezeichnet, extrahiert Eigenschaften aus dem Eingabebild und übt die Hauptanzahl an Berechnungen aus. Es handelt sich hierbei um Faltungsoperationen, die Ähnlichkeiten zur Fourier-Transformation und zur Laplace-Transformation aufweisen, und bilden das Merkmal eines CNNs. Ein Neuron einer Filter-Schicht betrachtet einen bestimmten Bereich einer vorherigen Schicht in Form einer Matrix und bildet daraus ein Skalarprodukt, um die den Bereich auf nur eine Zahl zu reduzieren. Die Architektur eines CNNs, die aus den verschiedenen Schichten und der Filter-Schichten besteht, ermöglichen, dass weniger Neuronen benötigt werden im Gegensatz zu anderen vielschichtigen neuronalen Netzwerken.[11] Die Knoten in der Filter-Schicht nutzen die sogenannte Rectified Linear Unit (**ReLU**)-Funktion als Aktivierungsfunktion. Diese ist, wie der Name bereits beschreibt, eine

lineare Funktion, welche leicht modifiziert ist, sodass der Rückgabewert der Funktion gleich null ist, wenn der Eingabewert kleiner gleich null ist. Das bringt diverse Vorteile bei der Optimierung von CNNs und ANNs. Lineare Funktionen sind leichter zu optimieren und zu trainieren. Die ReLU Funktion lässt sich mathematisch als das Maximum einer Menge definieren, welche zwei Elemente besitzt, den Wert 0 und den Wert x .

$$\text{ReLU}(x) = \max\{0, x\}$$

[16]

Aggregations-Schicht

Die Aggregations-Schicht ist für das Heruntersetzen der Dimensionen zuständig, um die Parameter der Aktivierungsfunktion zu reduzieren. Durch die Anwendung der zuvor liegenden Filter-Schichten wird die räumlichen Dimension erhöht und die damit einhergehend Anzahl der Parameter der darauf folgenden Schicht. Dies hat zur Folge, dass die Komplexität des Models erhöht wird. Diese Überanpassung wird mit den Aggregations-Schichten entgegengewirkt.[11] Für das Skalieren wird am häufigsten die *max pooling* Funktion verwendet. Diese Funktion reduziert die Dimensionen, ausgenommen der Tiefe, auf eine Größe von etwa 25% der Originalgröße.[10]

Vollständig-Verbundene-Schicht

Die vollständig verbundene Schicht ist die einer versteckten Schicht in einem ANN gleich. Diese Schicht ist hauptsächlich für die Klassifizierung und Berechnungen der Wahrscheinlichkeit zuständig. Die Aktivierungsfunktion der Knoten in dieser Schicht ist wie bei der Filter-Schichten die ReLU Funktion. Einleitet wurde erwähnt, dass es sich bei der Ausgabeschicht ebenfalls um eine vollständig verbundene Schicht handelt. Diese wird auch als *soft-max* Schicht bezeichnet und gibt die letztendliche Klassifizierung des Bildes zurück.[11]

Bewertung der Convolutional Neural Networks

Damit wir ein CNN mit der SVM und dem KNN vergleichen können, werden dieselben Bewertungskriterien untersucht. Die für die Bewertung relevanten Kriterien waren:

- Komplexität
- Genauigkeit
- Präzision
- Einfachheit der Umsetzung

Bevor wir genauer auf die Bewertungskriterien eingehen, werden wir uns mit dem für den Test konstruierten CNN beschäftigen. Dieser wurde in der Programmiersprache **Python** Version 3.9 geschrieben. Für die Konstruktion und die Ausführung des Tests benötigen wir verschiedene Bibliotheken, die dabei helfen können, die Bewertungskriterien zu ermitteln, sowie den CNN zu modellieren. Bei diesem Test wurde die Bibliothek **Tensorflow** verwendet, welche wiederum die **Keras** Bibliothek verwendet. Zusätzlich wurde die Bibliothek **NumPy** und **matplotlib** verwendet, um mit Hilfe von *NumPy* die Daten zu strukturieren und mit *matplotlib* die Ergebnisse darzustellen. Den Programmcode zum Importieren der notwendigen Bibliotheken, sowie die Vorbereitung für den CNN ist in Codeabschnitt 2.5 zu finden. In diesem Abschnitt werden zunächst die Bibliotheken importiert und die MNIST-Datenbank wird geladen. Beim Laden der Daten werden die Daten bereits in Trainingsobjekte und Testobjekte unterteilt.

```

1 import tensorflow as tf
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from keras import backend as K
5
6 mnist = tf.keras.datasets.mnist
7
8 (x_train, y_train), (x_test, y_test) = mnist.load_data()
9
10 y_train = y_train.astype('float32')
11 y_test = y_test.astype('float32')
12
13 # Erstellung von Binaerrendaten
14 mask = x_train > 127.5
15 maskb = x_train <= 127.5
16 x_train[mask] = 255
17 x_train[maskb] = 0
18
19 # Shift to -1 to 1
20 x_train, x_test = (x_train - 127.5) / 127.5, (x_test - 127.5) / 127.5
21
22 # Datenaufteilung fuer Validierung
23 x_val = x_train[-10000:]
24 y_val = y_train[-10000:]
25 x_train = x_train[:-10000]
26 y_train = y_train[:-10000]

```

Quellcode 2.5: Pythoncode zur Testvorbereitung vom CNN

Im folgenden Codeabschnitt wird das CNN-Model definiert. Das Model wurde mit folgenden Schichten definiert:

- Filter-Schicht (Eingabeschicht)
- Filter-Schicht
- Filter-Schicht
- Aggregations-Schicht
- Vollständig-verbundene-Schicht
- Vollständig-verbundene-Schicht (Ausgabeschicht)

Die zusätzlich zu sehenden Schichten *Dropout* und *Flatten* wurden definiert, um die Komplexität des Models zu reduzieren. Dieser Effekt ist nur während des Trainierens sichtbar. Bei der *Dropout*-Schicht handelt es sich um eine Schicht, die mit einer gewählten Wahrscheinlichkeit

von 0,25 einen Knoten aus der Schicht ausschaltet, um *overfitting* zu verhindern. Die *Flatten*-Schicht formt die mehrdimensionalen Daten in eine einzelne Dimension um, damit diese von der vollständig-verbunden-Schicht verarbeitet werden kann.

```

1 #CNN
2 model = tf.keras.models.Sequential([
3     tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
4                         activation='relu',
5                         input_shape=(28, 28, 1)),
6     tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
7                         activation='relu'),
8     tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
9                         activation='relu'),
10    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
11    tf.keras.layers.Dropout(0.5),
12    tf.keras.layers.Flatten(),
13    tf.keras.layers.Dense(128, activation='relu'),
14    tf.keras.layers.Dropout(0.25),
15    tf.keras.layers.Dense(10, activation='softmax')
16])
17
18 model.summary()
19 model.compile(optimizer='adam',
20                 loss='sparse_categorical_crossentropy',
21                 metrics=['accuracy', precision_m])
22
23 x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1],
24                               x_train.shape[2], 1))
25 x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], x_test.
26                               shape[2], 1))
27 x_val = np.reshape(x_val, (x_val.shape[0], x_val.shape[1], x_val.shape
28                               [2], 1))
29
30 model_data = model.fit(x_train, y_train, epochs=10, batch_size=128,
31                         validation_data=(x_val, y_val))
32 model.evaluate(x_test, y_test, verbose=2)

```

Quellcode 2.6: Pythoncode zur Konstruktion vom CNN

```

1 print(model_data.history)
2 plt.subplot(2, 1, 1)
3 plt.plot(model_data.history['accuracy'])
4 plt.plot(model_data.history['val_accuracy'])
5 plt.title('model accuracy')
6 plt.ylabel('accuracy')
7 plt.xlabel('epoch')
8 plt.legend(['train acc', 'val acc'], loc='lower right')
9 plt.subplot(2, 1, 2)
10 plt.plot(model_data.history['precision_m'])
11 plt.plot(model_data.history['val_precision_m'])
12 plt.title('model precision')
13 plt.ylabel('precision')
14 plt.xlabel('epoch')
15 plt.legend(['train precision', 'val precision'], loc='upper right')
16 plt.tight_layout()
17 plt.show()

```

Quellcode 2.7: Pythoncode zum Darstellen der Daten

Komplexität

Die *Zeit-Komplexität* von einem CNN-Modell ist die Summe der *Zeit-Komplexitäten* der einzelnen Schichten. Die genaue Berechnung als auch die genaue *Zeit-Komplexität* des oben erstellten Modells übertrifft den Rahmen dieser Arbeit. Für einen ausreichenden Vergleich zwischen den vorgestellten Klassifizierern wird lediglich die Komplexität der Filter-Schicht betrachtet. Die O-Notation der Filter-Schicht lautet wie folgt:

$$\sum_{l=1}^d n_{l-1} * s_l^2 * n_l * m_l^2$$

- l ist der Index von einer Filter-Schicht
- d ist die Anzahl der Filter-Schichten in einem Netzwerk
- n_l ist die Anzahl der Knoten in der l -ten Filter-Schicht
- n_{l-1} ist die Anzahl der Eingabeparameter
- s_l ist die dritte Dimension der Filter-Schicht
- m_l ist die Anzahl der Features die in der l -ten Filter-Schicht ausgegeben werden

[17]

Genauigkeit & Präzision

Um die Präzision vom CNN zu berechnen, muss zuerst eine Hilfsfunktion definiert werden, damit dieser Wert verfolgt wird. Die Metrik der Präzision wurde aus der Keras-Bibliothek 2.0 entfernt. Eine Lösung des Problems hat der Benutzer *Tasos* auf *StackExchange* veröffentlicht. [1]

```
1 def precision_m(y_true, y_pred):
2     true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
3     predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
4     precision = true_positives / (predicted_positives + K.epsilon())
5     return precision
```

Quellcode 2.8: Hilfsfunktion für Präzision [1]

Die Genauigkeit und die Präzision des CNN-Modells kann aus den Ausgaben der Epochenmetriken gelesen werden. Das Ergebnis der letzten Epoche wird in 2.9 dargestellt. Es ist zusehen, dass das CNN-Modell eine Genauigkeit von **99%** und eine Präzision von **92%** erreicht.

```
1 ...
2 Epoch 10/10
3 391/391 [=====] - 76s 195ms/step - loss: 0.0273
4     - accuracy: 0.9909 - precision_m: 0.9264 - val_loss: 0.0374 -
      val_accuracy: 0.9891 - val_precision_m: 0.9191
4 313/313 - 2s - loss: 0.0253 - accuracy: 0.9916 - precision_m: 0.9209 - 2
      s/epoch - 5ms/step
```

Quellcode 2.9: Testergebnisse vom CNN

Einfachheit der Umsetzung

In der Bewertung von diesem Kriterium wird von einem subjektiven Standpunkt ausgegangen. Es soll bewertet werden, wie einfach die Umsetzung des CNN-Modells in diesem Projekt ist. Das CNN-Modell wird mit **3/5** Punkten bewertet. Auch wenn das Forschungsgebiet der CNN in Bezug der Handschrifterkennung sehr gut beleuchtet ist, ist vor allem die Optimierung und Umsetzung eines CNNs komplex. Der CNN wird, ähnlich wie die anderen Algorithmen, bereits durch eine Bibliothek bereitgestellt, aber die Funktionen benötigen viele Parameter, wie zum Beispiel die Anordnung und die Definition der verschiedenen Schichten.

2.2.4 Auswertung des Vergleiches

Nachdem nun die drei Tests durchgeführt wurden muss ermittelt werden welcher Algorithmus für diese Arbeit verwendet wird. Hierfür werden die vier am Anfang genannten Kriterien miteinander verglichen.

- Komplexität
 - *KNN*: $O(np)$
 - *SVM*: $O(n_{sv}p)$
 - *CNN*: $\sum_{l=1}^d n_{l-1} * s_1^2 * n_1 * m_1^2$
- Präzision
 - *KNN*: 95,7%
 - *SVM*: 97,2%
 - *CNN*: 92,1%
- Genauigkeit
 - *KNN*: 95,5%
 - *SVM*: 96,9%
 - *CNN*: 99,2%
- Einfachheit der Umsetzung
 - *KNN*: 5/5
 - *SVM*: 5/5
 - *CNN*: 3/5

Mit einer hohen durchschnittliche Präzision und einer überschaubaren Komplexität kann der **SVM** diesen Vergleich für sich gewinnen. Allerdings ist es fragwürdig, ob es sinnvoll ist nicht einen Algorithmus zu wählen, der eine besseren Genauigkeit zu wählen, um möglichst immer die richtigen Zahlen zu erkennen.

Um dieses Problem zu lösen ist es möglich, die Test- und Trainingsdaten für *KNN* und *SVM* noch etwas besser vorzubereiten und damit ein besseres Ergebnis zu erzielen. Dafür müssen die Daten lediglich gemischt werden. Mit diesem Schritt können die beiden Algorithmen ihre Leistung nochmal signifikant steigern.

- Präzision
 - *KNN*: 99%
 - *SVM*: 99%
 - *CNN*: 92,1%
- Genauigkeit
 - *KNN*: 99,1%
 - *SVM*: 99,7%
 - *CNN*: 92,2%

Die anderen beiden Kriterien sowie der *CNN* ändern sich dabei nicht, da es keinen Einfluss auf sie hat.

Durch das Mischen gibt es nun einen eindeutigeren Gewinner, ohne Probleme bei der Genauigkeit: ***SVM***.

In dem später folgenden Kapitel 4.2.1 wird erneut eine neue Einschätzung des besten Algorithmus unter neuen Erkenntnissen vorgenommen.

3 Konzept

3.1 Genutzte Technologien

In diesem Kapitel werden die verwendeten Technologien und Frameworks beschrieben, die für das Projekt benötigt werden. Für die Umsetzung des Projektes fiel die Wahl auf die Programmiersprache **Python**. Sie ist vielseitig einsetzbar und es existieren bereits Bibliotheken mit Funktionalitäten, die in diesem Projekt benötigt werden. Diese relevanten Bibliotheken werden in den nächsten Unterkapiteln näher beleuchtet. Da wäre zum einen, die freie Programmbibliothek **OpenCV**, die den Bereich Bildbearbeitung abdeckt. Ebenfalls relevant ist das **SciKit-learn**, das bereits implementierte *Machine Learning*-Algorithmen zur Benutzung anbietet. Außerdem werden für das Projekt das Webframework **Flask** sowie die Bibliothek **Dash** genutzt. Beschrieben werden nur die relevanten Informationen, die für das Verständnis dieser Arbeit wichtig sind. Es besteht kein Anspruch auf Vollständigkeit in diesem Kapitel.

3.1.1 OpenCV

OpenCV ist eine plattformunabhängige Bibliothek im Bereich Computer Vision. Sie stellt Funktionen für verschiedenste Prozesse der Bildverarbeitung bereit. Dazu gehören unter anderem Bildanalyse und Objekterkennung, Kamerakalibrierung und Stereo-Vision sowie Ansätze für Machine Learning und Robotics. Die quelloffene Bibliothek wurde 1999 veröffentlicht und wird weltweit durch Entwickler in Forschungseinrichtungen und Unternehmen weiterentwickelt. Eine kommerzielle Nutzung ist aufgrund der BSD-Lizenz möglich. Die Implementierungen in **OpenCV** sind für echtzeitfähige Anwendungen konzipiert und dementsprechend optimiert. Teilweise existieren zusätzliche Implementierungen die mit CUDA oder OpenCL unter Nutzung der GPU beschleunigt werden. Durch sein breites Anwendungsspektrum ist **OpenCV** weitverbreitet und repräsentiert für viele Bereiche den Stand der Technik. **OpenCV** beinhaltet einen ORB Feature Detector und Descriptor sowie eine Sammlung von Algorithmen zur Zuordnung von Features. Diese Funktionalität kann bei der Bildvorverarbeitung der eingescannnten Spielergebnisse verwendet werden.

3.1.2 SciKit-learn

SciKit-learn ist eine freie Softwarebibliothek, die *Machine Learning*-Algorithmen für die Programmiersprache Python bereitstellt. Dazu gehören verschiedene Klassifikations-, Regressions- und Clustering-Algorithmen, wie zum Beispiel die *Support Vektor Machine*, Random Forest, Gradient Boosting, k-means oder DBSCAN. Sie basiert als SciKit (Kurzform für SciPy Toolkit) auf den numerischen und wissenschaftlichen Python-Bibliotheken **NumPy** und **SciPy**. Sie ist in verschiedenen Anwendungsbereichen einsetzbar und bietet effiziente Tools für maschinelles Lernen, Data Mining und Datenanalyse an. Weitere Funktionalitäten, die **SciKit-learn** für das maschinelle Lernen anbietet, sind Clustering, Cross Validation, Reduktion, Feature-Extraktion und Parameter-Tuning. Außerdem wird das Zusammenfassen und Darstellen der Daten vereinfacht.

Durch **SciKit-learn** konnten sehr einfach die einzelnen *Machine Learning*-Algorithmen *K-Nearest Neighbors* und *Support Vector Machine* ausprobiert werden. Es brauchte somit keine eigene Implementierung der Algorithmen und sie konnten ohne großen Aufwand miteinander verglichen werden. Auch bei der Implementierung der eigentlichen Ziffernerkennung kann das **SciKit-learn** eingesetzt werden.

3.1.3 Tensorflow

Ähnlich wie SciKit-learn ist auch **Tenosrflow** eine Softwarebibliothek die ebenfalls für *Machine Learning* verwendet wird und ist für Python und JavaScript verfügbar. **Tensorflow** ermöglicht es Modelle für neuronale Netzwerke zu erstellen und zu trainieren. Somit lassen sich *Convolutional Neural Networks*, Rekurrente neuronale Netze und viele weitere erstellen. Um die Nutzung von Tensorflow in Python zu vereinfachen wird noch die Application Programming Interfaces ([APIs](#)) von **Keras** verwendet. Diese minimiert den Aufwand zum Erstellen eines Modells und bietet klarer Fehlermeldungen sowie Zugang zu weiteren Datensätzen für das Training.

3.1.4 Flask

Flask ist Webframework für **Python**, das vom österreichischen Programmierer Armin Ronacher entwickelt worden ist. Bei diesem minimalistisch gehaltenem Framework liegt der Fokus auf eine hohe Erweiterbarkeit durch weitere Module und eine gute Dokumentation. Es bestehen nur zwei Abhängigkeiten zu weiteren Bibliotheken. Das wären zum einen die Template-Engine Jinja2 und Werkzeug, eine Bibliothek zur Erstellung von WSGI-Anwendungen. **Flask** nutzt für die Kommunikation zwischen Webserver und Webanwendung die Python-WSGI-Schnittstelle (Web Server Gateway Interface). Um die Programmierung zu erleichtern, stellt **Flask** für

Testzwecke einen mitgelieferten Webserver bereit. Ein weiteres Unterscheidungsmerkmal von **Flask** im Vergleich zu den anderen **Python**-Webframework, wie Django oder Web2py, ist, dass keine eigenen Komponenten angeboten werden, sondern die Funktionalitäten können sehr einfach durch weitere bestehende Bibliotheken erweitert werden. Das trägt dazu bei, dass **Flask** sehr minimalistisch gehalten werden konnte. Für die wichtigsten Funktionalitäten existieren bereits Erweiterungen, wie zum Beispiel für das Handhaben von Authentifizierung, Cookies und Sessions, das Konfigurieren des Cachings oder die Kompatibilität zu vielen Datenbanksystemen, wie MySQL, PostgreSQL oder MongoDB. Deshalb wird in diesem Projekt das Webframework **Flask** verwendet.

3.1.5 Dash

Plotly ist ein Unternehmen für technische Datenverarbeitung mit Hauptsitz in Montreal, Kanada, das Online-Tools für Datenanalyse sowie Datenvisualisierung entwickelt. Plotly bietet Online-Grafik-, Analyse- und Statistik-Tools sowie wissenschaftliche Grafikbibliotheken für Python, Perl, Arduino und weitere an. **Dash** ist ein Open-Source Python-Framework für die Entwicklung webbasierter Analyseanwendungen und Dashboards. Außerdem gibt es viele spezialisierte Open-Source-Bibliotheken für **Dash**, die auf die Entwicklung spezifischer **Dash**-Komponenten und Anwendungen zugeschnitten sind. Das **Dash**-Framework wird in diesem Projekt für die nachträgliche Analyse und Visualisierung der digitalisierten Spielergebnisse verwendet.

3.2 Auswertung und Visualisierung der Daten

Für die Auswertung der Daten und die daraus resultierenden Visualisierung muss zunächst definiert werden, welche Kriterien für das Dashboard relevant sind. Im Folgenden wird genauer darauf eingegangen, wie die *wichtigen* Werte aus den bereitstehenden Daten ermittelt werden und welche Methode genutzt werden soll, um diese zu visualisieren.

3.2.1 Auswertung

Ein Dashboard ist eine Darstellungsoption, mit der es einem Benutzer gelingen soll, einen Gesamtüberblick über die für ihn wichtigen Daten zu erhalten. Um diesen Gesamtüberblick erstellen zu können, muss eine Analyse des Benutzers vorgenommen werden, um zu ermitteln, welche Daten für ihn von Bedeutung sind. [18]

Da vorab festgelegt wurde, dass die Webapplikation ohne Benutzerregistrierung auskommen soll, können die Daten nicht direkt auf den Benutzer zugeschnitten werden, der seine Spielergebnisse hochlädt. Die Herausforderung besteht nun darin, Werte zu ermitteln, die für den Benutzer von Bedeutung sind, ohne einen persönlichen Bezug herstellen zu können. Dieser Aspekt resultiert darin, dass das Dashboard von mehreren Benutzergruppen genutzt wird. Es muss dementsprechend so konzeptioniert werden, dass Benutzer, die ihre Spielergebnisse hochladen, ihre wichtigen Daten erhalten, aber auch dass die Benutzer wichtige Informationen erhalten, die ihre Spielergebnisse nicht hochladen, wichtige Informationen erhalten. Die daraus resultierenden, für wichtig erachteten Werte, werden im Folgenden aufgelistet und diskutiert.

Total Sheets

Der erste Wert, der für relevant erachtet wird, ist die Summe der hochgeladenen handgeschriebenen Zettel. Da bei einem Kniffelspiel nicht immer alle sechs Spiele auf einem Zettel gespielt werden, kann dieser Wert die Anzahl der gespielten Sitzungen darlegen. Dieser Wert bietet darüber hinaus einen Indikator für die Nutzung der Webapplikation, da dieser Wert gleich ist mit der Anzahl der erfolgreichen Nutzung der Hochladen-Funktion. Mit dieser Angabe können die Nutzer einen Überblick über die gespielten Sitzungen erhalten und die Entwickler einen Überblick über die Nutzung der App.

First Upload

Der zweite Wert ist der Zeitpunkt, an dem der erste erfolgreiche Upload stattgefunden hat. Dieser Wert gibt Auskunft darüber, wie lange die App in Benutzung ist und wie sich die Daten über den Zeitraum entwickeln. Der Nutzer soll also mit der Angabe einen zeitlichen Überblick über die Berechnung der Daten erhalten, um diese besser einzuschätzen. Der Wert wird als ein Datum mit Tag, Monat und Jahr angegeben.

Total Yahtzees

Dieser Wert gibt Auskunft über die Summe der Yahtzees, die über alle Spiele hinweg gewürfelt wurden. Ein *Yahtzee*, oder auch *Kniffel* genannt wird gewürfelt, wenn alle Würfel die gleiche Augenzahl haben. Es ist das höchste Ergebnis, das ein Spieler mit einem Wurf erreichen kann. Dieser Wert ist daher von großer Wichtigkeit für einen Spieler. Im Zusammenhang mit dem ersten Wert kann der Nutzer einen Überblick über die Relation der Kniffel und der gespielten Sitzungen erhalten.

Total Score

Der vierte Wert gibt die Summe der erreichten Punkte für alle Spiele an. Dieser Wert hat keine besondere statistische Bedeutung, es soll lediglich als ein Wert dargestellt werden, der die Nutzer begeistert.

Highest Score

Der *Highest Score*-Wert soll die höchste erreichte Punktzahl eines Spielers, die mit einem Blatt erreicht wurde, darstellen. Da es verschiedenen Spielweisen des Spiels gibt, kann dieser Wert mehr aussagen, als wenn der höchste Wert eines Spiels eines Blattes genommen werden würde. Ein Nutzer kann somit einen Überblick über die Höchstpunktzahl erhalten, die ein Spieler erreicht hat.

Average Score

Der letzte Wert soll die durchschnittliche Punktzahl angeben, die in einer Sitzung erreicht wurde. Ähnlich wie beim *Highest Score*-Wert, wird dieser Wert nicht auf Basis der einzelnen Spiele auf einem Blatt ermittelt, sondern auf Basis der gesamten Sitzung. Es wird also das arithmetische Mittel über die Anzahl der *Total Sheets* und die Anzahl *Total Score* errechnet.

3.2.2 Visualisierung

Bei der Visualisierung kommt es darauf an, dass die Gesamtheit der Daten so präsentiert wird, dass diese einfach und zugänglich sind. Das schließt zum einen die optische Darstellung ein, aber auch die Navigation und Interaktion mit den Daten. Hierbei ist wiederum zu beachten, zu welcher Benutzergruppe eine Person gehört, da sich daraus die womögliche Medienkompetenz eines Nutzers ableiten lässt. [18] Das Ziel vom Dashboard soll es sein, die Daten möglichst zugänglich für jede Benutzergruppe zu machen. Aufgrund dessen sollen alle Daten, auf einem Blick erkennbar sein, sodass keine umfangreiche Navigation erforderlich ist, wie zum Beispiel eine Scroll-Bewegung. Die Abbildung 3.1 soll ein *MockUp* darstellen, wie die Darstellung der als wichtig definierten Werte aussehen könnte. Zusätzlich zu der einfachen Darstellung der Daten werden die Daten auch in einer grafischen Darstellung dargestellt. In dem Graphen sollen die Daten in Zusammenhang mit der Zeit dargestellt werden, um so Trends in den hochgeladenen Spielen genauer erkennen zu können.

3 Konzept



Abbildung 3.1: MockUp für die Visualisierung der Daten

4 Umsetzung

4.1 Image Preprocessing / Bildvorverarbeitung

Bevor die geschriebenen Spielergebnisse für die Auswertung in die *Support Vector Machine* gegeben werden können, müssen diese zuerst für den *Machine Learning*-Algorithmus vorbereitet werden. Dieser Prozess der Bildvorverarbeitung muss wiederum in weitere kleinere Schritte aufgeteilt werden. Der Prozess beginnt mit einer Bilddatei, die von einem Nutzer in die Webapplikation hochgeladen wird. Die *Support Vector Machine* kann jedoch nur einzelne Bildausschnitte verarbeiten, die eine einzelne geschriebene Ziffer enthält. Bei der Bildvorverarbeitung müssen demnach die handschriftlich ausgefüllten Ziffern des Nutzers aus dem hochgeladenen Bild erkannt und herausgeschnitten werden. Wie dies bewerkstelligt werden kann, wird im folgenden Kapitel herausgearbeitet.

Der Prozess bei Benutzung der Webapplikation kann folgendermaßen beschrieben werden: Der Nutzer füllt handschriftlich eine ausgedruckte Kniffel-Vorlage aus. Die Vorlage enthält eine Tabelle mit Zeilen für die möglichen Würfelergebnisse und Spalten für die einzelnen Spiele. Diese Kniffel-Vorlage wird dann vom Nutzer abfotografiert und auf die Webseite hochgeladen. Damit die *Support Vector Machine* die handschriftlichen Ziffern verarbeiten kann, müssen die einzelnen Zellen der Tabelle, die die Spielergebnisse enthalten, aus dem hochgeladenen Bild ausgeschnitten werden. Eine Idee ist ein Gitter über das Bild zu legen, das mit der Tabelle übereinstimmt, um so die einzelnen Zellen herauszuschneiden.

Ein Problem hierbei ist, dass das eingehende Bild von dem Nutzer abhängig ist. Die Qualität der Bilddatei und die Ausrichtung der darin abgebildeten Kniffel-Vorlage kann stark variieren. Es bestehen zwei Möglichkeiten dieses Problem zu lösen. Eine Lösung wäre es die Eckpunkte der Tabelle zu erkennen und dann das daraufzulegende Gitter anzupassen und daran auszurichten. Oder es wird das gesamte Bild genommen und so ausgerichtet, dass die abgebildete Tabelle immer an derselben Position liegt. So kann immer dasselbe Gitter über das Bild gelegt werden und es muss nicht nochmals angepasst werden. Bei der Umsetzung des Projektes fiel die Wahl auf den zweiten vorgestellten Lösungsansatz. Dieser Vorgang wird als *Image Alignment* bezeichnet.

Der Prozess des *Image Alignments* besteht aus drei verschiedenen Einzelschritten. Zuerst werden zwei Bilder in den Algorithmus hineingegeben, die beide dasselbe Objekt zeigen, jedoch aus verschiedenen Blickwinkel, beziehungsweise Perspektiven. Aus diesen beiden Eingangsbildern

wird eine dann eine sogenannte Homografiematrix berechnet. Um diese Homografiematrix berechnen zu können, gibt es verschiedene Möglichkeiten. Häufig werden hierfür die *Key Points* des Objektes betrachtet und wie die Position dieser Punkte auf den beiden Bildern miteinander korrespondieren. Zur Berechnung einer Homografiematrix können neben der feature-basierten *Key Point*-Korrespondez jedoch auch neuronale Netzwerke eingesetzt werden. Die Homografiematrix wird dann am Ende des *Image Alignments* auf eines der Eingangsbilder angewandt und die Perspektive des Bildes damit so verzerrt, dass beide Bilder nun das abgebildete Objekt aus derselben Perspektive zeigen.[19]

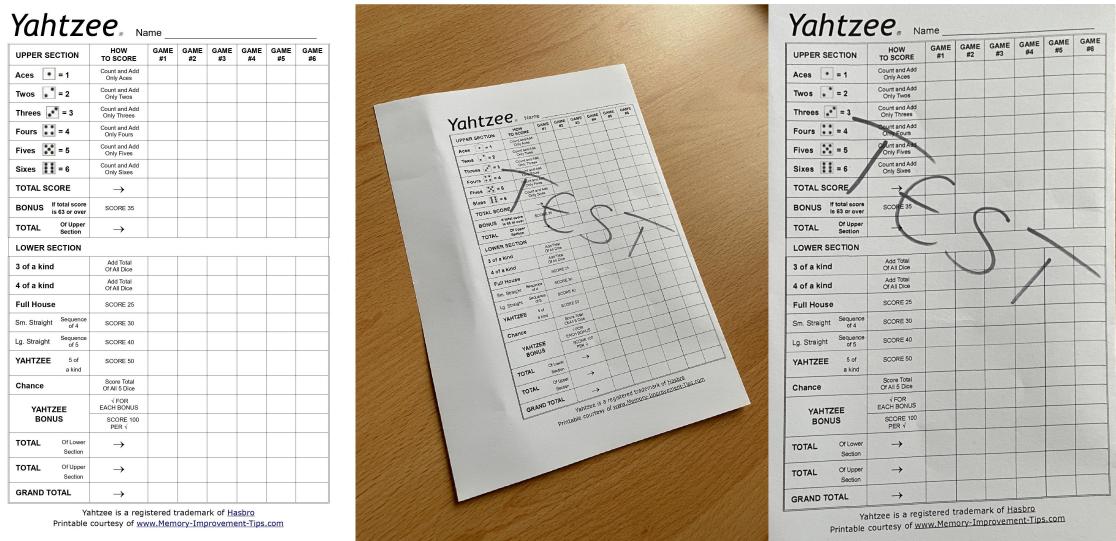


Abbildung 4.1: *Image Alignment*

Abbildung 4.1 visualisiert das oben beschriebene *Image Alignment*. Links ist die Yahtzee-Vorlage abgebildet, die auf der Webseite bereitgestellt wird und dann vom Nutzer heruntergeladen werden kann, der die Spielergebnisse einträgt. Die ausgefüllte Vorlage könnte dann so aussehen, wie das Bild in der Mitte, das das abfotografierte Formular zeigt. Zum Schluss Letzt sieht man auf der rechten Seite die abfotografierte Vorlage, die mit Hilfe von *Image Alignment* so ausgerichtet wurde wie das linke Bild.

Für unsere Zwecke nutzen wir **OpenCV**, das bereits *Key Point*-Erkennung von Bildern unterstützt. **OpenCV** sucht nach auffälligen Regionen, den sogenannten *Key Points*, innerhalb eines Eingangsbildes und extrahiert deren Deskriptoren. Mit diesen kann der umliegende Bereich um jeden *Key Point* quantifiziert werden. Die *Features* der *Key Points* werden als 128-dimensionaler Vektor angegeben. Wenn 528 *Key Points* in dem Eingabebild gefunden wurden, haben wir 528 Vektoren, die alle 128-dimensional sind. Diese Vektoren können dann in einen Algorithmus wie RANSAC gegeben werden, der die Korrespondenzen zwischen den *Key Points* der beiden Bilder bestimmt.

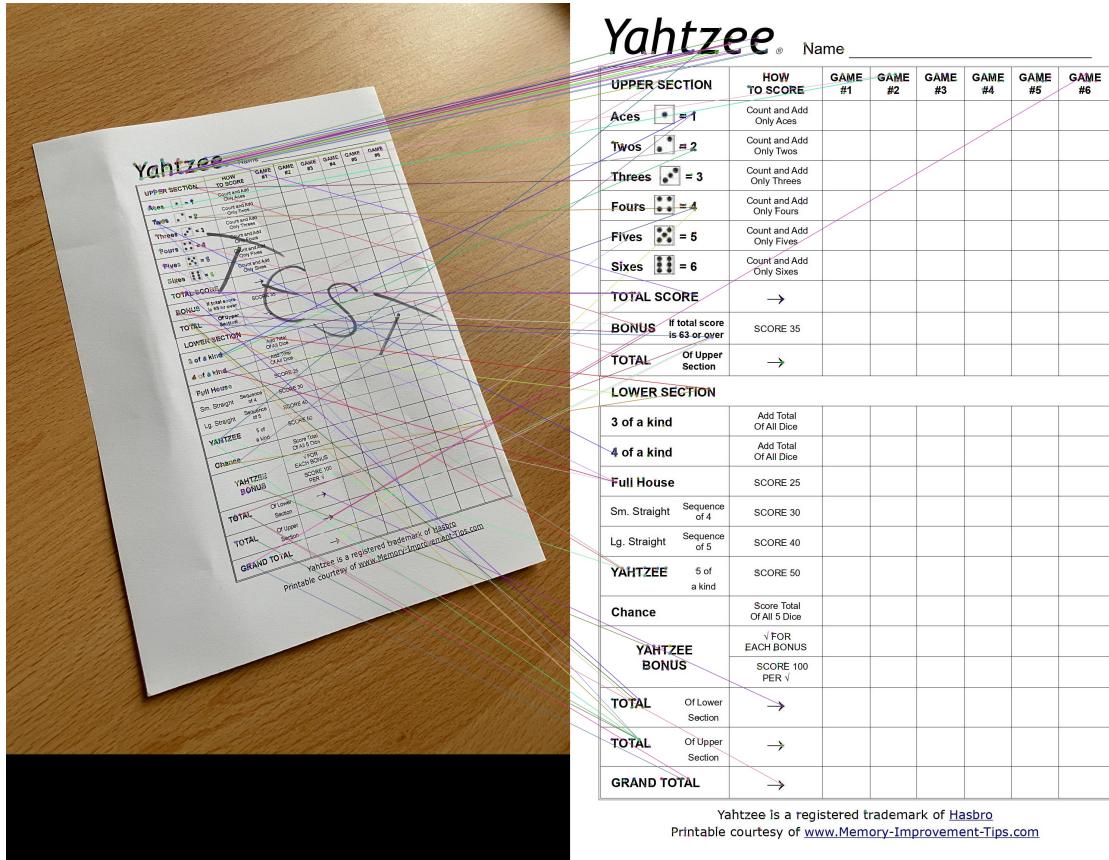


Abbildung 4.2: Korrespondenzen zwischen *Key Points*

Wenn genug Übereinstimmungen zwischen den *Key Points* der beiden Bilder existieren und ausreichend viele Korrespondenzen errechnet wurden, kann eine Homografiematrix mit den Ergebnissen erstellt werden. Diese Matrix beschreibt die perspektivische Verzerrung, die auf das Bild angewendet werden muss, um es auszurichten. Sie gibt die Rotation, Verschiebung und Skalierung an, die das abfotografierte Bild genauso ausrichten wie das Vorlagebild. Mehr zur Homografie ist [hier](#) nachzulesen.[19] Eine solche Homografiematrix sieht wie folgt aus:

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Viele der benötigten Funktionalitäten für das *Image Alignment* werden bereits durch **OpenCV** bereitgestellt. Die einzelnen Schritte müssen nun in **Python**-Code überführt werden. Der fertige Code, der in der Webapplikation verwendet wird, ist in 4.1 abgebildet. Als Vorlage für den Code diente die Implementierung von Rosebrock. Zuerst müssen das abfotografierte Bild und das Referenzbild von **OpenCV** eingelesen werden. Vor dem eigentlichen *Image Alignment* werden die beiden Bilder in ein Schwarzweiß-Format konvertiert.

Daraufhin werden mit Hilfe von **OpenCV** die ORB-*Features* auf den beiden Eingangsbildern erkannt. ORB steht hier für *Oriented FAST and Rotated BRIEF*. Die ORB-*Features* bestehen immer aus zwei Elementen: dem sogenannten *Locator*, sowie dem *Descriptor*. Der *Locator* identifiziert die *Features*, die unabhängig von Verschiebung, Rotation und Skalierung sind. Im *Locator* werden die Koordinaten dieser Punkte abgespeichert. Der *Descriptor* eines ORB-*Features* codiert die äußeren Merkmale des *Features* als ein Array von Zahlen. Idealerweise haben die gleichen Punkte auf zwei verschiedenen Bildern auch denselben *Descriptor*. Die maximale Anzahl der zu erkennenden *Features* werden mit dem Parameter `MAX_FEATURES` festgelegt.

Danach werden die erkannten *Features* der beiden Bilder einander anhand ihrer *Descriptors* zugeordnet. Die Matches werden dann sortiert und die schlechten Matches herausgefiltert. Die globale Variable `GOOD_MATCH_PERCENT` legt fest wie viele Matches nach dem Herausfiltern übrig bleiben sollen. In unserem Fall setzen wir die Variable auf 15%. Zu Darstellungszwecken wird im Code eine `matches.jpg` generiert, der die zuvor gefundenen Matches auf den beiden Bildern visualisiert. Mit dieser Codezeile wurde die Abbildung 4.2 erstellt.

Als Nächstes wird die Positionierung der einzelnen Matches extrahiert und in einen **NumPy**-Array geschrieben. Mit Hilfe des RANSAC-Algorithmus wird aus den Matches eine Homografiematrix berechnet. Diese Homografiematrix wird durch eine perspektivische Verzerrung auf das erste Bild angewendet. Das neue Bild sowie die Homografiematrix werden zum Schluss von der Funktion zurückgegeben. Bild 1 hat nun dieselbe Perspektive auf das abgebildete Objekt, wie die Vorlage in Bild 2.

```

1 import cv2
2 import numpy as np
3
4 MAX_FEATURES, GOOD_MATCH_PERCENT = 500, 0.15
5
6 def align_images(im1, im2):
7     # Bilder zu Schwarzweiss konvertieren
8     im1Gray = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
9     im2Gray = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)
10
11    # Erkennung der ORB-Features und Berechnung der Deskriptoren
12    orb = cv2.ORB_create(MAX_FEATURES)
13    keypoints1, descriptors1 = orb.detectAndCompute(im1Gray, None)
14    keypoints2, descriptors2 = orb.detectAndCompute(im2Gray, None)
15
16    # Matchen der Features
17    matcher = cv2.DescriptorMatcher_create(cv2.
18                                            DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING)
19    matches = matcher.match(descriptors1, descriptors2, None)
20
21    # Sortieren der Matches
22    matches = sorted(matches, key=lambda x: x.distance, reverse=False)
23
24    # Herausfiltern der schlechten Matches
25    numGoodMatches = int(len(matches) * GOOD_MATCH_PERCENT)
26    matches = matches[:numGoodMatches]
27
28    # Zeichnen der Matches (zu Darstellungszwecken)
29    imMatches = cv2.drawMatches(im1, keypoints1, im2, keypoints2, matches,
30                               None)
31    cv2.imwrite("matches.jpg", imMatches)
32
33    # Positionierung der Matches extrahieren
34    points1 = np.zeros((len(matches), 2), dtype=np.float32)
35    points2 = np.zeros((len(matches), 2), dtype=np.float32)
36    for i, match in enumerate(matches):
37        points1[i, :] = keypoints1[match.queryIdx].pt
38        points2[i, :] = keypoints2[match.trainIdx].pt
39
40    # Berechnung und Anwendung der Homografie
41    h, mask = cv2.findHomography(points1, points2, cv2.RANSAC)
42    height, width, channels = im2.shape
43    im1Reg = cv2.warpPerspective(im1, h, (width, height))
44
45    return im1Reg, h

```

Quellcode 4.1: Code für *Image Alignment*

Die Eingangsbilder werden nun durch das *Image Alignment* in dasselbe Format gebracht und zeigen danach die abgebildete Vorlage aus derselben Perspektive. Da die Zellen der Tabelle aufgrund des *Image Alignments* bis auf wenige Millimeter mit den Zellen der Vorlage übereinstimmen, können die Zellen anhand ihrer Koordinaten ausgeschnitten werden. Die minimalen Unterschiede, die das *Image Alignment* nicht ausgleichen kann, können hierfür unberücksichtigt bleiben.

Dafür wurde ein **Python**-Skript geschrieben, das die Koordinaten der einzelnen Zellen der Tabelle aus der Vorlage enthält. Es enthält die Parameter `column` und `row` und gibt dann die dazugehörigen Werte der definierten Zelle zurück. Diese Rückgabewerte sind die x und y-Koordinate des linken oberen Punktes der Tabellenzelle, sowie dessen Breite und Höhe. Diese vier Werte werden für den `crop()`-Befehl von **OpenCV** benötigt.

Mit Hilfe dieses Skriptes und **OpenCV** kann sehr einfach über die benötigten Spalten und Zeilen der Tabelle iteriert werden und die dazugehörige Zelle der Tabelle ausgeschnitten werden. Die ausgeschnittenen Tabellenzellen können jedoch noch nicht in den *Machine Learning*-Algorithmus gegeben werden, da es sich immer noch um mehrstellige Zahlen in der Zelle handeln könnte. Diese können noch nicht vom Algorithmus ausgewertet werden, da dieser nur einstellige Zahlen erkennen kann. Wie mehrstellige Zahlen erfasst werden können, wird im darauffolgenden Kapitel zur Zahlenerkennung beantwortet.

4.2 Zahlenerkennung

Durch die in der Bildvorverarbeitung 4.1 vorgenommenen Schritte wird das gesamte Blatt in die einzelnen Zellen aufgeteilt. Die Zellen werden dabei Spaltenweise bearbeitet, um so immer einen Spieldurchgang nach dem anderen abzuarbeiten. Durch die Aufteilung des kompletten Bildes in die einzelnen Zellen ist es möglich jeweils nur auf einen kleinen Teil des Bildes zu konzentrieren.

Um die einzelnen Zellen an den Algorithmus übergeben zu können und die erkannte und hoffentlich richtige Zahl zu erhalten, müssen zunächst ein paar Schritte unternommen werden. Diese Schritte werden dabei immer mit dem ursprünglichen Bildausschnitt, als der einen Zelle, durchgeführt. Als allererstes wird zunächst das Bild von einem farbigen in ein Graustufenbild gewandelt. Das geschieht, um die Farben zu entfernen, da diese den gesamten Prozess übermäßig kompliziert machen und für diese Klassifizierung keinen Mehrwert haben. Da es sich in fast jeder Zelle mehr als eine Zahl befinden kann, muss man zunächst eine Kantenerkennung auf das Bild anwenden. Hierfür müssen wir den Noise, also kleine Flecken oder andere störende Dinge, die zu klein für eine Zahl sind, entfernen. Jetzt können wir die Funktion `findContours` von *OpenCV* verwenden und erhalten damit die Anzahl der Konturen sowie deren Koordinaten. Diese Koordinaten teilen das Bild der ursprünglichen Zelle weiter auf, sodass nur noch eine

Zahl im Fokus ist. Um mit diesem neuen Bildausschnitt eine möglichst gute Erkennung zu gewährleisten, wird das Bild in ein sogenanntes Binärbild umgewandelt. Das heißt, dass das Bild danach nur noch aus schwarzen und weißen Pixeln besteht. Um dies zu erreichen, wird eine Grenzwertfunktion auf jedes Pixel angewendet. Ist der Grauwert des Pixels über dem Grenzwert wird er auf Weiß gesetzt, fällt er jedoch unter den Grenzwert wird der Pixel auf Schwarz gesetzt.

Durch diese Schritte haben wir uns für jede Zelle und jede darin enthaltende Kontur einen Bildausschnitt erzeugt und ihn in ein Binärbild gewandelt. Da wir unseren Algorithmus mit einer bestimmten Bildgröße trainiert haben, müssen wir den Bildausschnitt auch noch auf diese Größe anpassen. Dafür können wir, sollte der Ausschnitt zu groß sein, die *Resize* Funktion von *OpenCV* verwenden. Diese Funktion könnten wir auch verwenden, um ein kleines Bild zu vergrößern, allerdings ist es besser stattdessen das Bild mit schwarzen Pixeln aufzufüllen. Die zusätzlichen Pixel befinden sich ausschließlich am Rand des Bildausschnittes und erfüllen lediglich die Größenanforderungen des Algorithmus.

Nachdem wir nun einen Bildausschnitt mit einer Zahl haben, der sich nun auch in der richtigen Größe befindet, können wir ihn dem Algorithmus zur Klassifizierung übergeben. Die an dieser Stelle offensichtlich gewordenen Probleme werden im Kapitel 4.2.1 erläutert.

Der *CNN* gibt eine Liste von Wahrscheinlichkeiten für jede der Zahlen, also die Zahlen 0–9, zurück. Diese Wahrscheinlichkeit gibt an, wie sicher sich der Algorithmus ist, dass es sich um diese Zahl handelt. Da es automatisch abgearbeitet wird und nicht jede Zahl kontrolliert wird, wird immer die Zahl mit der höchsten Wahrscheinlichkeit genommen.

Sollte in der Zelle mehr als eine Kontur sein, also eine zwei- oder dreistellige Zahl darin stehen, wird dasselbe Verfahren auch für jede weitere Kontur angewendet. Die auf diese Art erhaltenen Zahlen werden nun anhand ihrer Position im Bild zu einer zweistelligen oder eben dreistelligen Zahl kombiniert.

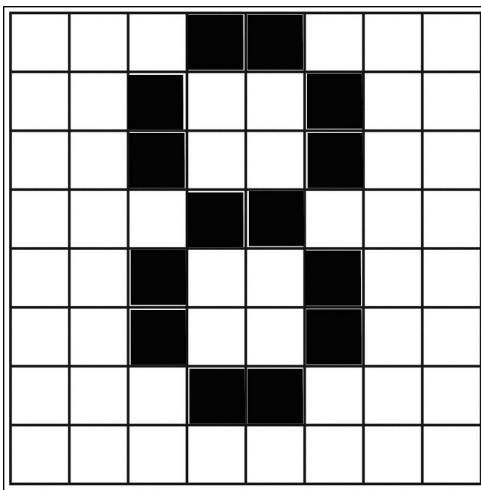
Sobald dieser Prozess für jede Zelle durchgeführt wurde, werden die Zahlen mit den vorhandenen Informationen zur Zelle in einer Datenbank gespeichert, sodass mit der Visualisierung 4.3 fortgefahren werden kann.

4.2.1 Probleme bei der Zahlerkennung

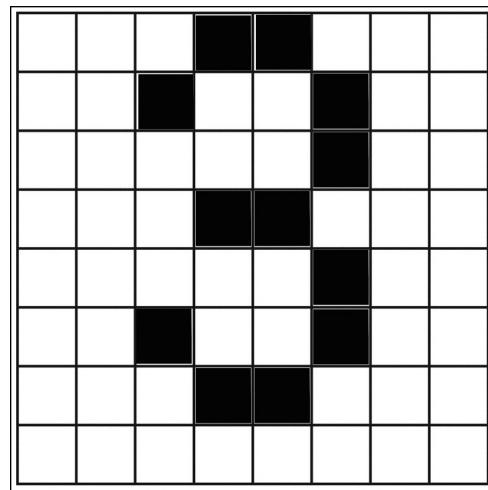
Bei der Verarbeitung der Bildausschnitte wird der jeweilige Ausschnitt verkleinert, um mit den Trainingsdaten übereinzustimmen. Dieser Schritt ist notwendig, da der SVM Algorithmus mit Bildern trainiert wurden, die alle eine Größe von 8×8 Pixeln haben. Die Größe der Bilder gibt dabei auch die Anzahl der Gewichte wieder, bei 8×8 großen Bildern sind es 64 Gewichte, bei 12×12 Pixeln wären es demnach 144 Gewichte und bei 28×28 Pixeln 784 Gewichte. Pro Pixel wird also jeweils ein Gewicht trainiert.

Bei dem Erkennen eines Bildes wird nun jeder Pixel überprüft, welche Farbe, in diesem Falle schwarz oder weiß, er besitzt. Je nach Anordnung erkennt jetzt der Algorithmus eine oder mehrere Zahlen. Da nicht jede Zahl immer gleich geschrieben ist, gibt es Pixel, die in jedem Fall eine Farbe haben müssen und andere wiederum können schwarz sein oder eben nicht. Diesen Wert geben die Gewichte des jeweiligen Pixels an. Bei einer 8 ist z.B. der Punkt in der Mitte immer schwarz. Der Bereich im oberen und unteren Kringel wiederum hat keine festgelegte Farbe, da es an dieser Stelle bei vielen verschiedenen Handschriften für diese Zahl kein einheitliches Muster gibt, sondern mal mehr und weniger weiße Fläche. Bei einer 1 hingegen wäre der Bereich unten links immer weiß.

Da es nun bei unterschiedlichen Zahlen auf einzelne Gewichte und wie diese trainiert sind ankommt, kann es schnell passieren, dass es bei ähnlichen Zahlen zu Verwechslungen kommt. Vergleicht man z.B. eine 8, siehe [4.3a](#), und eine 3, siehe [4.3b](#), fällt auf, dass sie sich nur um 2 Pixel unterscheiden. Hierbei kommt es natürlich immer darauf an, wie die Zahl geschrieben wurde.



(a) 8 in einem 8×8 Gitter

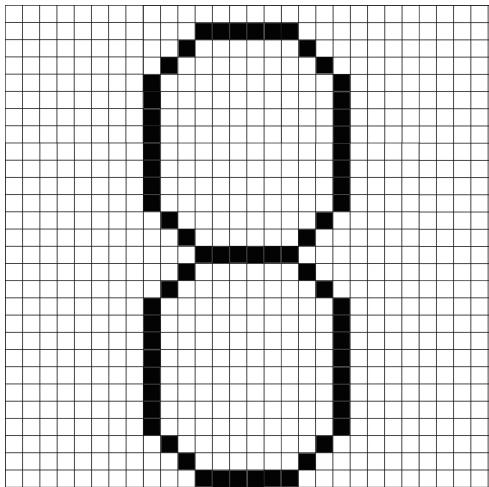


(b) 3 in einem 8×8 Gitter

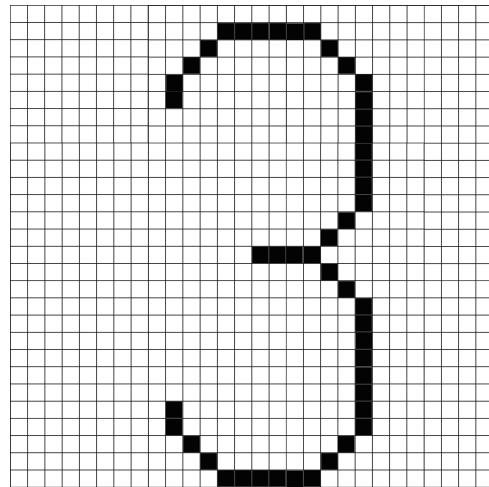
Abbildung 4.3: Zaheln in einem 8×8 Gitter

Je nachdem wie nun die Trainingsdaten des Algorithmus aussahen kann es sein das die Gewichte trainiert wurden, um beides als dieselbe Zahl zu erkennen. Da es in diesem Beispiel nur von einem einzigen Gewicht abhängig ist, ist das sogar wahrscheinlich.

Nutzt man jedoch stattdessen Bilder mit einer besseren Auflösung, also mit mehr Pixeln, kann das nicht so einfach passieren. Wie in [4.4](#) zu sehen unterscheiden sich die beiden Bilder um 18 Pixel, statt wie zuvor nur um 2 Pixel. Diese 18 Pixel bilden nun nicht nur einen größeren weißen Bereich, sondern es braucht auch mehr als nur ein einzelnes Gewicht, um die Vorhersage zu ändern. Außerdem ist auch, aufgrund der höheren Gesamtanzahl von Gewichten, der Einfluss eines einzelnen Gewichts auf das Gesamtergebnis geringer.



(a) 8 in einem 28×28 Gitter



(b) 3 in einem 28×28 Gitter

Abbildung 4.4: Zaheln in einem 28×28 Gitter

Dabei ist natürlich zu beachten, dass es für wirklich handgeschriebene Zahlen noch mehr Einflussfaktoren wie die bloße Farbe der Pixel gibt und auch jeder Mensch die Zahlen anders schreibt.

Allerdings haben diese zusätzlichen Gewichte auch einen großen Nachteil, da sie die Komplexität des Algorithmus erhöhen. Der SVM-Algorithmus ist bei den kleinen 8×8 Bildern, selbst bei weniger leistungsstarken Geräten, innerhalb von Millisekunden bis wenige Sekunden trainiert und einsatzbereit. Es wäre also möglich jederzeit neu zu trainieren ohne das es großen Einfluss auf die Nutzung der Anwendung hat. Wie in 2.2.4 zu sehen ist, besitzt er eine sehr gute Genauigkeit, weshalb sich auch zu Recht für diesen Algorithmus entschieden wurde.

Einen SVM-Algorithmus der hingegen mit Bildern der Größe 28×28 trainiert wird benötigt ungefähr 7 Minuten, es wäre also nicht möglich den Algorithmus mal schnell neu zu trainieren.

Aber, wie bereits erwähnt, muss jedes Bild, das dem Algorithmus zur Klassifizierung übergeben wird, zuvor auf die Größe von 8×8 Pixeln reduziert werden. Um dies zu erreichen, wird eine sogenannte Interpolation durchgeführt. Dabei wird aus den Farbwerten mehreren benachbarten Pixeln ein neuer Wert berechnet und somit vereint. Durch diesen Schritt kommt es allerdings zu Informationsverlust. Je größer der Unterschied zwischen dem Originalbild und dem Resultat ist, desto mehr Informationen gehen dabei verloren. Dies kann so weit gehen, dass am Ende nicht mal mehr eine Zahl zu erkennen ist und das Ergebnis der Klassifizierung nur geraten ist.

Trainiert man nun den SVM-Algorithmus mit einem Datensatz von 28×28 großen Bildern besitzt er, selbst mit mischen der Test- und Trainingsdaten, nur noch eine Genauigkeit von 94%.

[H]

```

1 Klassifizierungsbericht SVC(gamma=0.001):
2     precision    recall   f1-score   support
3
4      0.0        0.96     0.99     0.97      980
5      1.0        0.97     0.99     0.98     1135
6      2.0        0.94     0.92     0.93     1032
7      3.0        0.93     0.94     0.93     1010
8      4.0        0.92     0.95     0.94     982
9      5.0        0.92     0.91     0.92     892
10     6.0        0.95     0.96     0.96     958
11     7.0        0.95     0.93     0.94     1028
12     8.0        0.93     0.91     0.92     974
13     9.0        0.94     0.91     0.92     1009
14
15     accuracy          0.94     10000
16     macro avg       0.94     0.94     10000
17 weighted avg      0.94     0.94     10000
18
19 Genauigkeit: 0.9415

```

Quellcode 4.2: Testergebnisse der SVM mit 28x28 großen Bildern

Allerdings beinhaltet, wie sich an dieser Stelle herausgestellt hat, der in *Keras* vorhandene Datensatz bereits ausschließlich Bilder der Größe 28×28 . Besagter Datensatz wurde verwendet, um den *CNN* zu trainieren und dieser hat mit 99,16%, siehe 2.9, eine höhere Genauigkeit als der SVM-Algorithmus. Die um 5% höhere Genauigkeit kann bei der Benutzung der Anwendung einen entscheidenden Einfluss haben, wohingegen die Einfachheit der Umsetzung nur beim ersten Entwickeln des Algorithmus von Bedeutung ist. Aus diesem Grund ist es sinnvoll für die Klassifizierung der Zahlen den *CNN* anstatt des *SVM-Algorithmus* zu verwenden.

4.3 Auswertung & Darstellung

Nachdem die erkannten Spieldaten mit Hilfe des CNN erkannt wurden und in der Datenbank abgelegt wurden, werden die Ergebnisse im Dashboard dargestellt. Im Kapitel 3.2.2 wurde bereits darauf eingegangen, welche Daten im Dashboard dargestellt werden sollen und wie diese angeordnet werden. Darüber hinaus wurde bereits die Technologie in 3.1.5 vorgestellt, mit der diese Aufgabe bewältigt wird.

Um das Dashboard zu gestalten, wurden diverse Funktionen und *Best-Practices* von Dash umgesetzt. Das bedeutet, das Dashboard wurde mit Hilfe der HTML-API von Dash strukturell aufgebaut. Diese API ermöglicht es HTML-Tags durch die Anwendung von Funktionen zu defi-

nieren. Analog dazu verfügt Dash über eine API für die Nutzung von *Bootstrap*-Komponenten, die das Dashboard formatieren. [20]

Für die Darstellung der einzelnen Werte wurden sogenannte *Cards* von der *Bootstrap*-API verwendet. Diese stellen eine einfache Darstellung der Werte dar und wurden symmetrisch an der rechten Seite des Dashboards ausgerichtet.

Auf der linken Seite vom Dashboard befindet sich ein Graph, der ein gewisses Zeitintervall darstellt. Das Zeitintervall ist durch ein *Dropdown-Menü* auswählbar. Zur Auswahl stehen die letzten 7 Tage, die letzten 14 Tage, die letzten 30 Tage, die letzten 6 Monate und das letzte Jahr. Durch die Auswahl eines anderen Zeitraums aktualisiert sich der Graph automatisch, ohne die Seite neu zu laden. Der Graph verfügt außerdem über eine Tab-Navigation, die die Auswahl verschiedener Werte ermöglicht. Ein Nutzer kann so die Zeitintervalle für die **Uploads**, **Scores** und **Yahtzees** auswählen, um so eventuelle zeitliche Trends zu entdecken.

In der Abbildung 4.5 wird das erstellte Dashboard dargestellt. Bei den Daten in der Abbildung handelt es sich jedoch um Testdaten, die während der Entwicklung genutzt wurden.



Abbildung 4.5: Screenshot vom Dashboard mit Dummy-Daten, selbst erstellt und zugeschnitten

5 Ergebnisse und Ausblick

Im Rahmen der Arbeit ist eine Anwendung zur Auswertung eines Kniffel Ergebnisblatt unter Verwendung eines Machine Learning Algorithmus erstellt worden. Die Anwendung besteht aus einem Webserver, welcher die Schnittstelle zwischen dem Benutzer und der Verarbeitung darstellt. Der Webserver ermöglicht den Upload eines Ergebnisblattes und startet danach dessen Verarbeitung. Des Weiteren ermöglicht der Webserver einen Einblick in die Statistik der gespeicherten Daten. Die Ergebnisse eines neu ausgewerteten Blattes werden dabei mit abgespeichert und halten die Statistiken auf einem aktuellen Stand.

Die Webseiten wurden ansprechend und schlicht gestaltet, sodass eine einfache Bedienung ermöglicht wird. Ein weiterer positiver Punkt für eine einfache Bedienung ist die Möglichkeit, dass ein Bild auch dann verarbeitet werden kann, wenn es nicht perfekt gerade fotografiert wurde. Auf diese Weise kann schnell ein Bild gemacht werden ohne viel zu beachten.

Jedoch bedeutet dies auch eine große Einschränkung, denn das *Image Alignment* ist nur mit dem vorgegebenen Ergebnisblatt möglich.

Die Erkennung der Zahlen ist dafür gestaltet, um mit ein-, zwei- und dreistelligen Zahlen klarzukommen. Der *CNN-Algorithmus* bietet mit 99,16% zwar nicht die höchste Genauigkeit, im Vergleich mit den anderen Algorithmen, allerdings ist die Bildgröße von 28×28 Pixeln anstatt der 8×8 Pixel ein großer Vorteil. Dieser Vorteil bietet auch mehr Sicherheit bei der Erkennung. Die Statistiken bieten einen guten Einblick der vergangenen Spiele und stellt die erreichten Punkte übersichtlich dar. Die einzelnen Graphen bieten einen guten und einfachen Überblick des Verlaufs und ist damit eine um einiges bessere und angenehmere Möglichkeit als das Vergleichen der einzelnen Blätter miteinander.

Trotz aller Genauigkeit beim Erkennen der Zahlen in den einzelnen Zellen, kann dieser Punkt noch weiter verbessert werden. So sollte sichergestellt werden, dass Konturen nicht falsch erkannt werden, z.B. wenn zwei Striche einer Zahl etwas zu weit voneinander entfernt sind. Des Weiteren sollte auch eine Möglichkeit für den Nutzer eingeführt werden, um den Algorithmus zu kontrollieren. Dabei muss aber auch bedacht werden, dass eine Kontrolle nicht verpflichtend sein sollte.

Aber auch beim Upload der Bilder können Verbesserungen vorgenommen werden. Dazu gehört die Verarbeitung von allen möglichen Kniffel Ergebnisblättern zu realisieren und es nicht nur auf die Vorlage zu beschränken.

Des Weiteren kann man über ein Speichern der Bilder nachdenken, um die Möglichkeit zu bieten auch im Nachhinein nochmal darüber zu schauen und, sollte es implementiert sein, nochmal zu kontrollieren.

Als Letztes kann noch eine Art Benutzerverwaltung eingeführt werden und somit Statistiken für mehrere Nutzer anbieten. Die zusätzlichen Nutzer bieten dann auch die Möglichkeit für zusätzliche Statistiken, dazu gehört einem Vergleich der Leistungen der Benutzer untereinander.

Literatur

- [1] ZeleIBZeleIB, TasosTasos, matzematze, Ashok Kumar JayaramanAshok Kumar Jayaraman, Justin LangeJustin Lange und Viacheslav KomisarenkoViacheslav Komisarenko. *How to get accuracy, F1, precision and recall, for a keras model?* März 2020. URL: <https://datascience.stackexchange.com/a/45166>.
- [2] Deutsche Welle (www.dw.com). *Brettspiele Boomen in der Pandemie: DW:* 07.02.2021. Feb. 2021. URL: <https://www.dw.com/de/brettspiele-boomen-in-der-pandemie/a-56420599>.
- [3] Annina Simon, Mahima Singh Deo, S. Venkatesan und D.R. Ramesh Babu. „An Overview of Machine Learning and its Applications“. In: *International Journal of Electrical Sciences & Engineering (IJESE)* 1.1 (2015).
- [4] Prof. Dr. Kornmeyer Harald. *Script und Vorlesung des Machine Learning Moduls Wintersemesters 2021.* 2021.
- [5] *Computational complexity of machine learning algorithms.* <https://web.archive.org/web/20220312140637/https://www.thekerneltrip.com/machine/learning/computational-complexity-learning-algorithms/>. Letzter Zugriff: 12.03.2022.
- [6] Gareth James, Daniela Witten, Trevor Hastie und Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R.* Springer, 2013.
- [7] Shan Suthaharan. *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning (Integrated Series in Information Systems, 36).* Springer, 2015.
- [8] William S. Noble. „What is a support vector machine?“ In: *Nature Biotechnology* 24.12 (2006).
- [9] Bohdan Macukow. „Neural Networks – State of Art, Brief History, Basic Models and Architecture“. In: *Computer Information Systems and Industrial Management*. Hrsg. von Khalid Saeed und Władysław Homenda. Cham: Springer International Publishing, 2016, S. 3–14. ISBN: 978-3-319-45378-1.
- [10] Keiron O’Shea und Ryan Nash. *An Introduction to Convolutional Neural Networks.* 2015. DOI: [10.48550/ARXIV.1511.08458](https://arxiv.org/abs/1511.08458). URL: <https://arxiv.org/abs/1511.08458>.

- [11] Mohit Sewak, Md. Rezaul Karim und Pradeep Pujari. *Practical Convolutional Neural Networks - Implement advanced deep learning models using Python*. Birmingham: Packt Publishing Ltd, 2018. ISBN: 978-1-788-39414-7.
- [12] P.J. Braspenning, F. Thuijsman und A.J.M.M. Weijters. *Artificial Neural Networks - An Introduction to ANN Theory and Practice*. Berlin Heidelberg: Springer Science & Business Media, 1995. ISBN: 978-3-540-59488-8.
- [13] Rohit Dwivedi. *What are different loss functions used as optimizers in neural networks?* Juni 2020. URL: <https://www.analyticssteps.com/blogs/what-are-different-loss-functions-used-optimizers-neural-networks>.
- [14] Denis Conniffe. „Expected Maximum Log Likelihood Estimation“. In: *Journal of the Royal Statistical Society. Series D (The Statistician)* 36.4 (1987), S. 317–329. ISSN: 00390526, 14679884. URL: <http://www.jstor.org/stable/2348828> (besucht am 12.04.2022).
- [15] Brendan Fortuna und Marcelo Viana. *Loss functions*. Okt. 2019. URL: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#:%text=Cross%2Dentropy%20loss%2C%20or%20log%20predicting%20a%20probability%20of%20..
- [16] Ian GOODFELLOW, Yoshua BENGIO und Aaron COURVILLE. *Deep learning*. MIT Press, 2016.
- [17] Kaiming He und Jian Sun. *Convolutional Neural Networks at Constrained Time Cost*. 2014. DOI: [10.48550/ARXIV.1412.1710](https://doi.org/10.48550/ARXIV.1412.1710). URL: <https://arxiv.org/abs/1412.1710>.
- [18] Stephen Few. *Information dashboard design: Displaying data for at-a-glance monitoring*. Analytics Press, 2013.
- [19] Adrian Rosebrock. *Image alignment and registration with OpenCV*. <https://web.archive.org/web/20210703132843/https://www.pyimagesearch.com/2020/08/31/image-alignment-and-registration-with-opencv/>. Letzter Zugriff: 14.03.2022.
- [20] *Dash documentation; user guide*. URL: <https://dash.plotly.com/>.