|  |  |  |
| --- | --- | --- |
| **MA 3062** | | |
| Untersuchung und prototypische Umsetzung eines Lifelong Deep Neural Network Algorithmus | | |
| **Simon Kamm** | | |
|  | | |
| **Beschreibung des Prototyps** | | |
|  | Prüfer: | Prof. Dr.-Ing. Michael Weyrich |
|  | Betreuer: | Benjamin Maschler, M.Sc. |
| Start: 29.04.2019 | | Abgabe: 29.10.2019 |
|  | |  |

**Dokument Versionsverwaltung**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version | Autor | QS | Datum | Status | Änderungen |
| 0.1 | Kamm |  | 03.08.19 | in Bearb. | Erstellung |
| 1.0 | Kamm | Ms | 09.08.19 | Vorgelegt | Finalisierung SW-Prototypenbeschreibung |
| 1.1 | Kamm | Ms | 21.08.19 | Vorgelegt | Anpassung Prototypbeschreibung zu geänderter SW |

# Inhaltsverzeichnis

0 Inhaltsverzeichnis 2

1 Systemarchitektur 2

2 Beschreibung der Systemkomponenten 4

2.1 Main-Funktion 4

2.2 Input-Funktion 5

2.3 Modul A 6

2.4 Modul B 6

2.5 Datenteilung und -Vorbereitung 10

2.6 Kontinuierliches Lernen 11

2.7 Verteiltes Lernen 11

2.8 Berechnung und Visualisierung der Metriken 12

3 Installations- und Benutzungsanleitung 13

# Systemarchitektur

In diesem Abschnitt wird auf die Systemarchitektur des SW-Prototypen für die Realisierung des L DNN Algorithmus eingegangen. Die Gliederung des Systems in Komponenten folgt auf Basis von Aufgaben, welche in unterschiedlichen Funktionen/Klassen abgearbeitet werden. Die einzelnen Komponenten (Aufgabenblöcke) sowie deren Schnittstellen und die Datenflüsse werden in diesem Abschnitt benannt. Die Software ist nicht komplett identisch zu den Komponenten aufgebaut, jedoch lassen sich einzelne Funktionen/Klassen den Komponenten zuordnen. Für eine graphische Übersicht ist in Abbildung 1 die Systemarchitektur grafisch dargestellt. Die blauen Boxen stellen dabei die einzelnen Komponenten dar, die im Folgenden kurz genannt und in Kapitel 2 detaillierter beschrieben werden. Die Pfeile stellen dabei die Funktionsaufrufe und Datenflüsse dar.



Abbildung : Grafische Darstellung der Komponenten und deren Zusammenhang

Mithilfe der gewählten Komponentenstruktur ist es mit beschränktem Aufwand möglich, einzelne Bereiche in der Systemarchitektur auszutauschen, ohne weitere Komponenten verändern zu müssen. So kann beispielsweise ein anderes Modul A gewählt werden, ohne andere Komponenten davor oder danach anzupassen. Auch können in einem möglichen Experiment mit Live-Bildern die Input-Funktion und die Daten-Teilung und Vorbereitungskomponente ausgetauscht werden, wenn diese nicht benötigt werden oder anders realisiert wird.

Der generelle Programmablauf wird in der **Main-Funktion** gesteuert. Aus dieser Funktion heraus werden alle Komponenten sowie deren Funktionen aufgerufen. Hier werden zudem alle einstellbaren Parameter, die im weiteren Verlauf genutzt werden, zentral gesetzt, z.B. die Anzahl an Trainingsbildern pro Klasse.

In der **Input-Funktion** werden die ausgewählten Daten geladen, vorbereitet und separate Datensätze für die Trainings- und Testdaten an die Main-Funktion zurückgegeben. Zusätzlich kann ein zufälliges Trainingsbild vor und nach der Bild-Augmentation visualisiert werden um den Einfluss dieses Schrittes zu sehen.

In **Modul A** wird das vortrainierte CNN zur Feature-Extraktion geladen und eine Instanz dieser Klasse initialisiert und an die Main-Funktion zurückgegeben. Als Eingabeparameter wird die Bildgröße benötigt, da abhängig davon unterschiedliche Gewichte des CNNs geladen werden.

Innerhalb von **Modul B** werden eine oder mehrere Instanzen der Klasse *FuzzyARTMAP* mit den übergegebenen Parametern initialisiert und an die Main-Funktion zur weiteren Verwendung zurückgegeben. Parameter für die Initialisierung sind die Lernrate , der Vigilance-Parameter , der Choice-Parameter , der Parameter für das Match-Tracking sowie die Anzahl an Klassen und der zusätzliche Parameter für die Umsetzung des „Nothing I Know“-Konzepts.

Innerhalb der **Datenteilung- und Vorbereitung** wird der Datensatz für die folgenden Anwendungsfälle in die einzelnen Klassen aufgeteilt und die Features aus den Originaldaten extrahiert. Dafür benötigt diese Komponente als Eingangsdaten Modul A, die Anzahl an Klassen im Datensatz, die Trainings- und Testdaten sowie die per Parameter einstellbare Anzahl an Trainings- und Testbildern pro Klasse. Die Übergabe der Parameter erfolgt mithilfe des angelegten Parameter-Dictionary. An die Main-Funktion werden die Features sowie die Labels der einzelnen Bilder in Listen pro Klasse zurückgegeben, aufgeteilt in eine Trainings- und Testliste.

Im **Kontinuierlichen/Verteilten Lernen** wird abhängig vom gewählten Anwendungsfall kontinuierliches Lernen auf einem Gerät oder auf mehreren (verteilten) Geräten durchgeführt (in diesem SW-Prototyp durch mehrere Instanzen von Modul B umgesetzt). Es werden die extrahierten und separierten Trainings- und Testdaten mit dazugehörigen Labels sowie die Instanzen von Moduls B übergeben. In dieser Komponente findet das gesamte Training und Testen des inkrementellen Klassifikators statt. Es werden die finalen Prädiktionen des Netzwerks auf den Testdaten mit den dazugehörigen korrekten Labels sowie die erzielten Genauigkeiten über die Anzahl an erlernten Klassen zurückgegeben.

In der **Berechnung und Visualisierung der Metriken** werden die Ergebnisse visualisiert und gegebenenfalls weitere Metriken berechnet. Implementiert ist die Darstellung einer *Confusion Matrix* pro Instanz von Modul B sowie das Darstellen der Klassifikationsgenauigkeit über die Anzahl an erlernten Klassen.

# Beschreibung der Systemkomponenten

Die in Kapitel 1 eingeführten Komponenten werden im Folgenden detailliert hinsichtlich ihrer Aufgabe und Umsetzung beschrieben. Zudem werden die vorhandenen Funktionen/Programme/Klassen der SW den einzelnen Komponenten, soweit möglich, zugeordnet.

## **Main-Funktion**

**Diese Funktion steuert den Programmablauf. Um eine einfache Nutzung zu ermöglichen, werden alle relevanten Parameter hier zentral in einem *Dictionary* gebündelt und beschrieben. Die Komponente dient zum Verketten der einzelnen Komponenten und importiert Funktionen aus anderen Komponenten. Parameter, die hier gesetzt werden können sind im Folgenden aufgelistet:**

* **Save – Parameter zum Ein- und Ausschalten der Speicherung von Parametern, Ergebnisse und erstellten Graphiken**
* **Dir2save – Speicherpfad, in dem die Parameter, Ergebnisse und Graphiken des Tests gespeichert werden sollen (Nur relevant, wenn vorheriger Parameter auf *True* ist)**
* **No\_classes – Anzahl an Klassen des Datensatzes**
* **No\_groups – Anzahl an Gruppen, in welche die Klassen eingeteilt werden sollen (z.B. 5 Gruppen für Split-MNIST obwohl 10 Klassen vorhanden sind 🡪 2 Klassen pro Gruppe)**
* **Train\_img\_per\_class – Anzahl an Trainingsbildern pro Klasse**
* **Test\_img\_per\_class – Anzahl an Testbildern pro Klasse**
* **No\_edge\_devices – Anzahl an Endgeräten (relevant für den Anwendungsfall des verteilten Lernens)**
* **Test\_case – Gewünschter Testfall (zur Auswahl stehen „kontinuierlich“ und „verteilt“)**
* **No\_repetitions – Anzahl an Wiederholung des Testfalls, um eine statistische Aussagekraft zu erhalten**
* **Modul\_B\_xxx – Attribute für Instanzen der Klasse B (Mögliche Attribute siehe unter 2.4)**

**Die gesetzten Parameter werden zur Nachvollziehbarkeit der Ergebnisse in einer *Json*-Datei gespeichert (sofern der „Safe“-Parameter auf *True* steht). Ebenfalls werden alle Metriken in einer *Json*-Datei gesammelt. Zudem kann über verschiedene Werte eines Parameters mithilfe eine *for-*Schleife iteriert werden, um eine Vielzahl an Untersuchungen zu diesem Parameter automatisiert durchlaufen lassen zu können. Dafür müssen die gewünschten Parameterwerte in einem Array gesammelt werden, und über dieses Array iteriert werden. Bei jedem Durchlauf wird der Parameter in dem Parameter-Dictionary überschrieben und die gewünschte Anzahl an Wiederholungen (no\_repetitions) mithilfe einer weiteren *for-*Schleife umgesetzt. Diese beschriebene Funktionalität ist in *Main.py* realisiert.**

## **Input-Funktion**

**Der Daten-Eingang besteht abhängig vom Datensatz aus mehreren Bereichen. Aus *Main.py* heraus wird die Funktion mit den Übergabeparametern Datensatz und einer boolschen Variable *„Visu“* aufgerufen*.* Die Aufgabe dieser Komponente besteht darin, den gewünschten Datensatz zur weiteren Verwendung zur Verfügung zu stellen. Die Umsetzung dieser Aufgabe ist in** Abbildung 2 **als Flussdiagramm dargestellt und wird im Folgenden erläutert.**



Abbildung : Flussdiagramm der Input-Funktion

**Wie in** Abbildung 2 **dargestellt, wird zunächst überprüft, ob dieser Datensatz in *Tensorflow* innerhalb des Paketes „tf\_dataset“ zur Verfügung steht und von dort genutzt werden kann. Wenn ja, kann der nächste Schritt übersprungen werden.**

**Wenn der Datensatz nicht in *Tensorflow* zur Verfügung steht, kann die Komponente aus „*tfrecord“*-Dateien den gespeicherten Datensatz herstellen. Dafür wird eine separate Extraktionsfunktion aufgerufen, um den Datensatz korrekt auszulesen.**

**Wenn der Datensatz geladen ist, findet eine Augmentation der Bilder statt. In dieser Arbeit wird das Bild in *float*-Variablen umgewandelt und in den Bereich zwischen 0 und 1 normalisiert. Dies ist ein typischer Vorverarbeitungsschritt in der Bildverarbeitung.**

**Zudem werden GrayScale Bilder in RGB-Bilder umgewandelt und Bilder mit einer Höhe oder Breite kleiner als 96 Pixel auf 96x96 Bilder umgewandelt. Diese Schritte sind notwendig, weil das vortrainierte CNN in Modul A lediglich für RGB-Bilder mit einer Mindestgröße von 96x96 trainiert wurde.**

**Mithilfe der „*Visu“*-Variable kann entschieden werden, ob ein Bild des Datensatzes vor und nach der Augmentation exemplarisch dargestellt werden soll. Dies dient der Veranschaulichung der Bild-Augmentation.**

**Die Aufgaben dieser Komponente werden innerhalb der Datei *Data\_Input.py* realisiert und an die Main-Funktion wird der Trainings- und Test-Datensatz zur weiteren Verwendung zurückgegeben.**

## **Modul A**

**Modul A dient im L DNN Algorithmus zur Feature-Extraktion aus den ursprünglichen Eingangsbildern. Dieses Modul ist fixiert, d.h. es findet kein weiteres Training dieses Netzwerkes statt. Die Komponente wird mit der Bildgröße (*image\_size*) aufgerufen, da abhängig von der Bildgröße die Dimensionen des vortrainierten Netzwerkes ausgewählt werden. Mit dieser Information können vortrainierte CNNs in *Tensorflow* geladen werden. Gesammelt sind diese vortrainierten Modelle in *tf.keras.applications* (Doku unter** <https://www.tensorflow.org/tf/keras/applications>**). Wie in der Konzeption beschrieben, wird das CNN MobileNet-V2 ohne die Klassifikationsschichten genutzt. Es werden die Gewichte übernommen, welche auf Basis des gesamten ImageNet-Datensatzes trainiert wurden. Im Programm wird eine Instanz der MobileNet-V2 Klasse gebildet, die unterschiedliche Methoden und Attribute bereithält. Das Attribut *trainable* dieser Instanz wird in dieser Arbeit auf *False* gesetzt, um die erlernten Gewichte des CNNs zu fixieren.**

**Für die weiteren Aufgaben ist im Kontext dieser Arbeit lediglich die *predict*-Funktion relevant, welche auf Basis von Eingangsdaten die Features für die spätere Klassifikation extrahiert. Zusätzlich kann über die *summary*-Funktion eine Übersicht über Modul A in der Konsole ausgegeben werden mit dem Aufbau/den Dimensionen der einzelnen Schichten sowie der Anzahl an Parametern innerhalb des Netzwerkes.**

**Das Erstellen der Instanz sowie das Fixieren des Netzwerks und eine mögliche Ausgabe der Informationen über das Netzwerk ist innerhalb von *ModulA.py* implementiert. Das Extrahieren der Features aus den Eingangsdaten findet in der Komponente zur Daten-Teilung und Vorbereitung statt.**

## **Modul B**

**In der Komponente Modul B ist der inkrementell lernende Klassifikator (in dieser Arbeit ein Fuzzy ARTMAP-Netzwerk) realisiert. Dafür wurde eine Klasse *FuzzyARTMAP* implementiert, die alle relevanten Attribute und Funktionen bereithält, um Modul B als inkrementellen Klassifikator einsetzen zu können.** Abbildung 3 **stellt die Klasse *FuzzyARTMAP* in einem Klassendiagramm für eine erste Übersicht dar.**



Abbildung : Klassendiagramm der Klasse *FuzzyARTMAP*

**Im Folgenden werden die dargestellten Attribute und Funktionen der Klasse weiter erläutert.**

**Zur Initialisierung einer Instanz müssen die unterschiedlichen Attribute initial gesetzt werden. Eine Instanz hat dabei folgende Attribute:**

* **(Alpha) – Lernrate:**

**Zur Bildung einer neuen Repräsentation auf Basis einer alten Repräsentation und der neuen Eingangsdaten nach Formel (1):**

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

* **(Gamma) – Choice Parameter:**

**Verhindert eine Division durch 0 bei der Berechnung der Aktivierung der Eingangsdaten mit den Repräsentationen**

* **(Rho) – Vigilance Parameter:**

**Schwellwert, der vom Matching (der Ähnlichkeit) zwischen Eingang und Repräsentation erreicht werden muss, um die Eingangsdaten einer Repräsentation und damit Klasse zuordnen zu können**

* **(Epsilon) – Parameter für das Match-Tracking:**

**Bei Fehlklassifizierung während dem Training wird dieser Mechanismus eingesetzt um sicherzustellen, dass eine Repräsentation mit höherem Matching der korrekten Klasse vorhanden ist oder neu abgelegt wird**

* **– „Nothing I Know“-Parameter:**

**Faktor zur Bestimmung des Schwellwertes für das Nothing I Know-Konzept um eine eindeutige Gewinnerkategorie zu erhalten und neue Kategorien erkennen zu können (Details siehe Konzeption)**

* **Anzahl Klassen:**

**Die Anzahl der maximal vorkommenden Klassen. Notwendig um ein korrektes Mapping der Kategorie-Knoten auf die Klassen durchführen zu können**

* **Anzahl Training True Positives:**

**Attribut zum Erfassen der korrekt klassifizierten Trainingsdaten um während dem Training eine Information über die Performanz zu erhalten**

* **– Top-Down Gewichte:**

**Diese Gewichte sind die Repräsentation des FuzzyARTMAP-Netzwerks. Mit diesen werden die Aktivierungen und Matching-Werte der Eingangsdaten errechnet und darauf aufbauend die Klassen bestimmt.**

* **– Mapping Array:**

**Dieses Array wird genutzt um das korrekte Mapping zwischen der Ausgabe Kategorie Schicht und der Kategorie Schicht durchzuführen. Der Index einer Repräsentation innerhalb des Netzwerks wird den konkreten Klassenlabels zugeordnet (z.B. Repräsentation 3 entspricht Klasse 1).**

**Zudem besitzt die Klasse *FuzzyARTMAP* einige Member-Funktionen, deren Aufgaben und Umsetzung im weiteren Verlauf erläutert werden.**

\_init\_weights(self, x, y)

**Diese Funktion initialisiert die Top-Down Gewichte und das Mapping , wenn diese noch nicht initialisiert wurden. Dafür wird das Eingangssample als initiales Gewicht der Klasse abgespeichert. Im Mapping Array findet dafür ein entsprechender Eintrag mithilfe des Labels statt.**

\_add\_category(self, x, y)

**Mit dieser Funktion werden neue Kategorien hinzugefügt. Dabei wird das Eingangssample der Klasse als neues Gewicht zu hinzugefügt. In dem Mapping Array wird ebenfalls mithilfe des Labels ein neuer Eintrag hinzugefügt.**

\_match\_category(self, x, y=None)

**Hier wird das Eingangssignal mit den bekannten Repräsentationen des Netzwerks, , verglichen und „gematched“. Es werden die Aktivierungen und Matchingwerte berechnet und mit dem eingestellten Schwellwert verglichen. Während des Trainings ist ein Label verfügbar, welches genutzt wird um das Eingangssignal der passenden Repräsentation zuzuordnen. Wenn das Eingangssignal keine Repräsentation ausreichend „matched“, gibt die Funktion „-1“ (Unbekannt) zurück und sonst den internen Index der Gewinner-Repräsentation.**

**Im Testfall ist kein Label vorhanden. Hier wird der Index der Gewinner-Repräsentation zurückgegeben. Falls der Schwellwert nicht erreicht wird, wird ebenfalls ein „-1“ zurückgegeben und das „Nothing I Know“ findet Anwendung.**

Train(self, x, y, epochs=1)

**Das Training des Netzwerks wird über diese Funktion durchgeführt. Dabei stellt den Datensatz im Format (samples, features) dar. Die Labels sind in enthalten. Über die Variable *epochs* kann die Anzahl an Epochen für das Training mit den gegebenen Daten und festgelegt werden, was in dieser Arbeit standardmäßig auf „1“ ist. Der Ablauf der Trainings-Funktion ist in** Abbildung 4 **graphisch als Flussdiagramm dargestellt und wird im weiteren beschrieben.**



Abbildung : Flussdiagramm der Trainings-Funktion der Klasse *FuzzyARTMAP*

**Wenn keine Gewichte vorhanden sind, wird zunächst mit dem ersten Sample ein Gewicht initialisiert. Daraufhin findet das Training für die gegebene Anzahl an Epochen in zufälliger Reihenfolge der Samples statt.**

**Es wird über die Samples iteriert und mit dem aktuellen Sample *\_match\_category* aufgerufen. Wenn die Kategorie „-1“ (unbekannt) zurückkommt, wird eine neue Kategorie mithilfe von *\_add\_category* angelegt. Ansonsten wird das Gewicht der erhaltenen Kategorie mithilfe von Formel (1) angepasst und das nächste Sample wird gezogen.**

***Test(self, x)***

**Mit dieser Funktion wird der Testfall, beziehungsweise die spätere Anwendung im Betrieb umgesetzt. Dabei stellt die Testdaten dar, welche bei den vorhandenen Datensätzen vom Format (sample, features) sind. Es können jedoch auch einzelne Samples übergeben werden. Die Funktion gibt die prädizierten Klassen der Testdaten zurück. Es wird wiederum für jedes Sample die Funktion *\_match\_category* aufgerufen. Wenn „-1“ zurückkommt, wird das Sample der Kategorie „Unknown“ zugeordnet und der Nutzer wird aufgefordert die Kategorie einzugeben, um einen neuen Kategorie-Knoten über *\_add\_category* anzulegen.**

Consolidation(self)

**Mit dieser Funktion können die Top-Down Gewichte, und damit die Repräsentationen der einzelnen Kategorien, konsolidiert werden. Es werden dabei alle Repräsentationen einer Kategorie (Klasse) zu einer Repräsentation für diese Instanz zusammengefasst. Dies kann in realen Anwendungen relevant sein, wenn der Speicher begrenzt ist und viele Kategorien auftreten. Diese Funktion kann zu jedem Zeitpunkt von einer Instanz aufgerufen werden.**

Melding(self, modul\_b\_additional)

**Mithilfe dieser Funktion können mehrere (verteilte) Instanzen der Klasse FuzzyARTMAP zusammengefasst werden und deren erlerntes Wissen ausgetauscht werden. Dies ist für den Anwendungsfall des verteilten Lernens relevant, wenn unterschiedliche Instanzen unterschiedliche Klassen (und damit verschiedenes Wissen) erlernt haben. Durch das *melding* der Instanz mit einer zusätzlichen Instanz () enthält die aufrufende Instanz die Repräsentationen und das dazugehörige Mapping der zusätzlichen Instanz. Dafür werden die Repräsentationen und die Maps der zusätzlichen Instanz zu den passenden Attributen angehängt.**

## **Datenteilung und -Vorbereitung**

**Innerhalb dieser Komponenten werden zum einen mithilfe von Modul A Features aus den ursprünglichen Eingangsbildern extrahiert. Zum anderen werden die Daten auch in Bezug auf ihre Klassen separiert, um später das inkrementelle Klassenlernen einfach implementieren und umsetzen zu können. Diese Aufgaben werden innerhalb der Funktionen *data\_separation\_train* und *data\_separation\_test* in der Datei *Helper.py* umgesetzt.**

**Der Funktion wird *Modul A*, die Trainings- oder Test-Datensätze sowie das Parameter-Dictionary übergeben. Aus dem Parameter-Dictionary werden die Anzahl an Klassen und an gewünschten Trainings- oder Testbildern pro Klasse innerhalb der Funktion benötigt.**

**Es wird über den Trainings- und Test-Datensatz iteriert, bis ausreichend Trainingsbilder (*train\_img\_per\_class*) oder Testbilder (*test\_img\_per\_class*) pro Klasse verfügbar sind. Wenn für die Klasse des ausgewählten Samples noch nicht genügend Daten vorhanden sind, werden die Features des Bildes mithilfe der Komponente *Modul A* extrahiert. Die Features werden vektorisiert, und dieser Feature-Vektor mit Elementen wird mithilfe von Formel (2) auf die Länge 1 normiert.**

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

**Pro Klasse wird eine separate Liste angelegt, die final in eine verkettete Liste zusammengefügt werden. Mit dieser Konstruktion können später im Training und Testfall des kontinuierlichen und verteilten Lernens die einzelnen Klasse einfach inkrementell dem Netzwerk zugeführt werden. Für die Labels wird ebenfalls eine verkettete Liste für den Trainings- und Test-Datensatz angelegt. Die Komponente gibt vier verkettete Listen zurück: Die Feature-Listen für die Trainings- (*feature\_list\_train*) oder Test-Daten (*feature\_list\_test*), sowie die dazugehörigen Label-listen (*label\_list\_train* und *label\_list\_test*). Durch das Aufsplitten der Datenteilung in zwei Funktionen, können die einzelnen Datensätze einfach zu den gewünschten Zeitpunkten erstellt werden. Wenn zum Beispiel für eine hohe Anzahl an Wiederholungen jeweils zufällig unterschiedliche Trainingsdaten benötigt werden, kann bei jeder Wiederholung eine neue *feature\_list\_train* erzeugt werden aus zufällig gezogenen Trainingsdaten. Wenn die Testdaten jeweils die selben sind, kann die *feature\_list\_test* einmalig vor Beginn der Wiederholungen erzeugt werden, wodurch die Rechenzeit erheblich reduziert werden kann bei einer hohen Anzahl an Testdaten.**

## **Kontinuierliches Lernen**

**In dieser Komponente wird der Anwendungsfall des kontinuierlichen Lernens ausgeführt. Das kontinuierliche und das verteilte Lernen wird in der SW-Umsetzung getrennt, jedoch wird es in einer graphischen Darstellung wie** Abbildung 1 **als eine Komponente zusammengefasst, da immer nur eine von beiden Anwendungen ausgewählt werden kann. Das kontinuierliche Lernen ist innerhalb von *Continual\_Learning.py* mit der Funktion *continual\_learning\_fun()* umgesetzt. Als Übergabeparameter erhält die Funktion Werte aus den zuvor beschriebenen Komponenten: Eine Instanz von Modul B, die Feature-Listen der Trainings- und Testdaten, die dazugehörigen Labels, die Anzahl an Klassen des Datensatzes und die Anzahl an Gruppen, in welche die vorhandenen Klassen eingeteilt werden sollen.**

**Für die Umsetzung des kontinuierlichen Lernens wird über die Anzahl an festgelegten Gruppen iteriert. Dafür wird zu Beginn die Gruppengröße berechnet, um die korrekte Anzahl an Klassen für das Training bereit zu stellen und alle Klassen zu trainieren. Mithilfe der Gruppengröße werden dann aus der Liste der Klassen die benötigte Anzahl an Klassen gezogen und für das Training von Modul B benutzt. Über das Hinzufügen einer einzelnen Zeile kann ausgewählt werden, ob die Reihenfolge der Klassen zufällig angeordnet sein soll (z.B. 9, 3, 5, 1, 0, 2, 4, 7, 8, 6) oder die „klassische“ Reihenfolge eingehalten werden soll (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Abhängig vom Datensatz ist eine zufällige oder feste Reihenfolge sinnvoll.**

**Zudem wird eine zusammengefügte Test-Liste (*feature\_list\_test\_merged*) erstellt, in der alle bereits trainierten Klassen für die Test-Funktion zusammengefasst sind. Nach dem Training einer Gruppe von Modul B wird mit dieser Test-Liste die *test*-Funktion von Modul B aufgerufen und die Klassifikationsgenauigkeit des Netzwerks zu diesem Zeitpunkt bestimmt und gespeichert. Nachdem jede Gruppe trainiert und getestet wurde, werden am Ende die Prädiktionen und Labels der gesamten Testdaten zurückgegeben. Zusätzlich wird ein Vektor mit der Klassifikationsgenauigkeit über die einzelnen Iterationen zurückgegeben.**

## **Verteiltes Lernen**

**Die Umsetzung des Verteilten Lernens findet innerhalb von *Distributed\_Learning.py* in der Funktion *distributed\_learning\_fun()* statt. Die Übergabeparameter sind identisch zu der Funktion des kontinuierlichen Lernens (siehe oben). Der grundlegende Aufbau und Ablauf sind ähnlich zu dem Fall des kontinuierlichen Lernens, jedoch mit einigen Erweiterungen und Änderungen um die verteilten Instanzen (Geräte) unabhängig voneinander zu trainieren. Zu Beginn werden die Gruppen gleichmäßig auf die Anzahl an Geräten aufgeteilt. Weiter wird wie beim kontinuierlichen über die unterschiedlichen Gruppen iteriert.**

**Wenn die berechnete Anzahl an Gruppen für das eine Gerät trainiert wurden, wird ein Geräte-Index erhöht und die nächste Instanz von Modul B wird für das Training ausgewählt. Das Training pro Instanz folgt dem Ablauf des kontinuierlichen Lernens, lediglich mit einer geänderten Anzahl an Gruppen (Klassen).**

**Wenn alle Gruppen durchlaufen sind wird das erlernte Wissen der unterschiedlichen Geräte mithilfe der *melding*-Funktion von Modul B kombiniert, ohne dabei die Trainingsdaten austauschen zu müssen. Zum Schluss findet mit diesen zusammengefügten Netzwerken ein weiteres Testen auf Basis aller Testdaten statt. Die Funktion gibt schließlich die Labels, Prädiktionen und die Genauigkeiten der einzelnen Iterationen zurück. Hier werden jeweils Listen mit separaten Einträgen für die einzelnen Geräte und des fusionierten Netzwerks zurückgegeben, um die einzelnen Geräte und den Einfluss der Fusion bewerten zu können. Zusätzlich wird die Klassifikationsgenauigkeit der zusammengefügten Netzwerke zurückgegeben.**

## **Berechnung und Visualisierung der Metriken**

**In dieser Komponente werden finale Metriken erstellt und visualisiert. Dafür werden die zuvor bestimmten Genauigkeiten und Prädiktionen genutzt. Es wird eine *Confusion Matrix* mithilfe der Funktion *confusion\_matrix\_plot()* (in *Helper.py*) aus den erhaltenen Label-Listen und den Prädiktionen des Netzwerks erstellt. Pro initialisierte Instanz wird eine Confusion Matrix erstellt. Bei dem Anwendungsfall des verteilten Lernens wird eine zusätzliche Matrix des kombinierten (*melded*) Netzwerks erstellt.**

**Zusätzlich wird die Genauigkeit des Netzwerks über die Anzahl an erlernten Klassen visualisiert, um das Verhalten bezüglich *Catastrophic Forgetting* untersuchen zu können. Bei mehreren Geräten werden die einzelnen Kurven innerhalb des gleichen Plots mit unterschiedlichen Farben dargestellt. Dies wird mit der Funktion *accuracy\_plot()* (in *Helper.py*) realisiert. Als weitere Metrik wird der belegte Speicher der einzelnen Module A und B ermittelt und in der Konsole ausgegeben. Die Genauigkeit über die einzelnen Schritte sowie der Speicherbedarf werden für eine mögliche spätere Auswertung in einer *Json*-Datei gesichert. Die erstellten Graphiken werden in demselben Ordner abgelegt. In** Abbildung 5 **und** Abbildung 6 **sind beispielhaft die Metriken für einen Continual Learning Fall dargestellt.** Abbildung 5 **zeigt eine Confusion Matrix auf Basis des MNIST-Datensatzes mit den möglichen Klassen 0 bis 9. Die prädizierten Klassen sind entlang der x-Achse und die tatsächlichen Klassen entlang der y-Achse aufgetragen.** Abbildung 6 **zeigt die Klassifikationsgenauigkeit über die Anzahl an erlernten Klassen für denselben Testfall.**

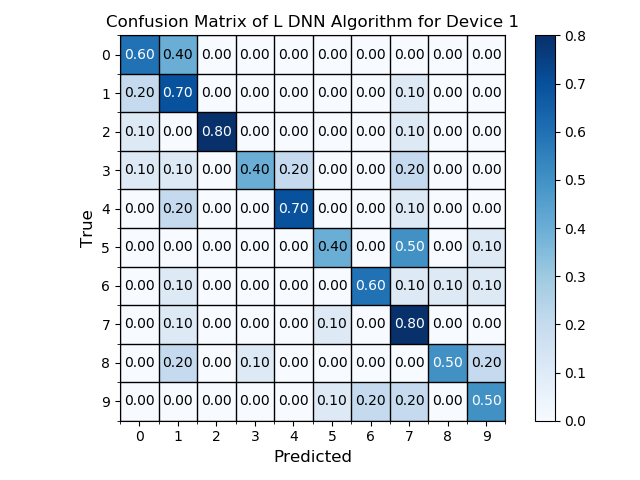


Abbildung : Beispielhafte Confusion Matrix

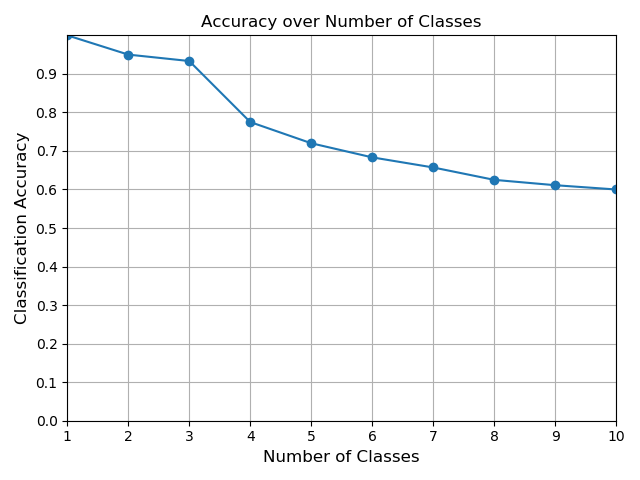


Abbildung : Beispielhafter Plot der Genauigkeit über die Anzahl an erlernten Klassen

# Installations- und Benutzungsanleitung

Zur Benutzung des SW-Prototypens müssen Programme/Pakete installiert werden, welche im Folgenden genannt werden. Als Grundbedingung ist eine funktionierende Version von Python3.6 erforderlich. Darauf aufbauend werden folgenden Pakete mit der angegebenen Versionsnummer innerhalb des SW-Prototypen genutzt:

* Pympler-Package (v0.7)
* Numpy (v1.16.4)
* Tensorflow (v2.0.0b1)
* Matplotlib (3.1.1)
* Tensorflow\_datasets (1.0.2)

Dies sind die Pakete, welche speziell installiert wurden und explizit in der Software importiert werden, da aus diesen Paketen Funktionalitäten genutzt werden. Einige dieser Pakete haben wiederum Abhängigkeiten mit weiteren Paketen, weshalb diese automatisch mit installiert werden bei dem Benutzen eines entsprechenden Installationsprogramms (z.B. über PyCharm). Diese werden hier nicht speziell aufgelistet. Wenn die notwendigen Pakete installiert sind, muss lediglich der vorhandene Code lokal gespeichert werden (verfügbar unter <https://github.com/simonbuerste/Masterarbeit/tree/master/Code>) und das Programm kann durch Ausführen von *Main.py* gestartet werden.

Zur Adaption des Testfalls oder des Prototypens müssen lediglich Parameter innerhalb der Funktion *Main.py* angepasst werden. Die Bedeutung der Parameter wurde im Rahmen von Kapitel 2 beschrieben. Auch können einzelne Module ausgetauscht werden, wenn z.B. ein anderer inkrementeller Klassifikator für Modul B genutzt werden soll.

Beachtet werden muss, dass die Anzahl an Klassen durch die Anzahl der Gruppen beziehungsweise Geräten teilbar ist, um eine korrekte Ausführung zu ermöglichen.

In diesem Prototyp wurde keine spezielle Logik entwickelt, um diesen Fall abzudecken, da im Rahmen der Arbeit kein Anwendungsfall definiert ist, der auf solch einen Testfall (nicht gleichverteilte Gruppen) abzielt. So führt zum Beispiel im Fall des MNIST-Datensatzes (10 Klassen) eine Anzahl von 8 Gruppen dazu, dass lediglich die ersten 8 Klassen trainiert und getestet werden aufgrund der Integer-Division . Dieses Verhalten muss bekannt sein, um keine falschen Schlüsse aus solch einem Test zu ziehen.