CKi ein CNN-Model in C++

by Simeon Stix

21. Februar 2024

Version: 2.1

Betreuer: Sven Nüesch

Inhaltsverzeichnis

1 Analyse

1.1 Vorhaben

Die Aufgabe, die sich das Projekt CKi stellt, ist die Wissenserweiterung des Entwicklers. Dabei wird ein Programm geschrieben, welches einzelnen handgeschriebenen Ziffern erkennen kann. Dies geschieht mittels KI. Dabei wird die KI komplett vom Entwickler geschrieben. Da keine modernen externen Grundlagen verwendet werden, wird das Produkt, das Programm, nicht die Geschwindigkeit einer modernen KI erreichen. Dies spielt jedoch keine Rolle, da dieses Projekt nicht wegen des Endprodukts durchgeführt wird.

1.2 Zielgruppe

Das Projekt CKi ist als solches nicht ausgelegt einem realen Anwendungszweck zu entsprechen oder eine Lösung oder einen Lösungsansatz für einen solchen zu bieten. Diesbezüglich liegt der einzige Nutzen von CKi nicht in dessen Produkt, sondern nur im Wissensgewinn und Verständnisgewinn für den Entwickler in den Bereichen der künstlichen Intelligents oder genauer im Bereich des maschinellen Lernens mit einem *Convolutional Neural Network*, der Realisation von Anwendungen mit C++ und dessen Möglichkeiten Hardware direkt in die programminternen Abläufe einzubinden. Somit richtet sich CKi nicht nach dem Grundsatz ein bestmöglich nutzbares Produkt zu sein, sondern lediglich nach dem grössten Wissensgewinn für den Entwickler. Nach diesem Grundsatz ist die resultierende Zielgruppe der Entwickler und vereint so mehrere Rollen des Projektes CKi in einer Person.

1.3 Anforderungen

1.3.1 Must-haves

Die Must-haves wurden aus dem Themenblatt, welches am 15.09.2023 bei Walter Schnyder eingereicht wurde, übernommen und mit weiterführenden Elaborationen versehen.

- **Rückgabe in Prozentwerten, die die Wahrscheinlichkeit der Übereinstimmung mit dem digitalen Gegenstück der handgeschriebenen Zahl abbilden.** „Welche Zahl wurde (vermutlich) aufgeschrieben?“

  Erläuterung: Da ein simples neuronales Netzwerk für maschinelles Lernen durch ein „Netz“ aus Knotenpunkten gebaut wird und jeder dieser Knotenpunkte, auch die Knotenpunkte, welche bei einem solchen neuralen Netzwerk als Endschnittstellen fungieren, einzeln berechnet werden, erhält man, bei Anwendungsfall von CKi, eine multiple Anzahl von Prozentzahlen, welche zur Interpretation des gelieferten Endergebnisses verwendet werden können. Diese Rückgabe der einzelnen Prozentwerte erfolgt zum Beginn über eine Konsolenausgabe. Diese wird später, wie in 1.3.2 („Nice-to-haves“) unter GUI erläutert, in ein grafisches Nutzerinterface integriert und zu diesem Zeitpunkt evtl. auch interpretiert (wobei die einzelnen Prozentwerte weiterhin einsichtig bleiben sollten).

- **funktionales neuronales Netzwerk (trainiert auf Zahlenwert)**

  Erläuterung: Ein CNN-Algorithmus oder auch Convolutional Neural Network wird beim maschinellen Lernen oft bei der Interpretation von Bildern genutzt. Dabei wird das Bild in kleinere Abschnitte unterteilt und „einzeln“ an den gehirnähnlich aufgebauten Algorithmus weitergegeben.

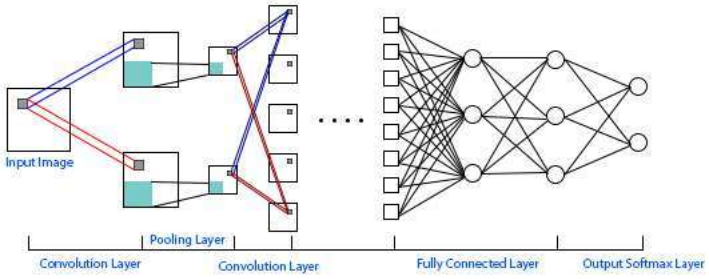


Abbildung 1.1: CNN Model Aufbau Grafik von researchgate.net

Im Projekt CKi wird dieser wie im Themenblatt beschrieben mit Bildern von einzelnen handgeschriebenen Ziffern trainiert, dem MNIST-Datensatz.

- **Nutzer-Input | Nutzer darf eine Zahl zeichnen**

  Erläuterung: Ein solches CNN-Model zu erbauen und zu trainieren ist zwar die Grundlage für dieses Must-have, jedoch sollte das Produkt auch erprobbar sein. Diesbezüglich muss der Nutzer in der Lage sein, eine handschriftliche Ziffer an das neurale Netzwerk zu liefern. Hierbei ist die minimale Anforderung, dass der Nutzer in einer anderweitigen Applikation ein solches Bild erstellt hat und es nun interpretieren lassen kann. Wie in 1.3.2 („Nice-to-haves“) unter GUI beschrieben, wird diese Eingabemöglichkeit (eine Ziffer zu zeichnen oder eine schon vorhandene grafische Abbildung zu verwenden) in einem weiteren Entwicklungsschritt direkt in die zukünftige grafische Nutzeroberfläche der Applikation integriert.

1.3.2 Nice-to-haves

Die Nice-to-haves wurden aus dem Themenblatt, welches am 15.09.2023 bei Walter Schnyder eingereicht wurde, übernommen und mit weiterführenden Elaborationen versehen. Zudem behalte ich mir als Verfasser dieses Dokumentes, als Entwickler des Projektes CKi und als Zielgruppe des Projektes CKi vor, diese Liste in gegebenen Fall zu erweitern.

- **GUI**

  Erläuterung: Ein GUI (oder auch grafisches User-Interface) ist die grafische Nutzeroberfläche der Applikation. Da die Applikation primär dem Wissensgewinn gewidmet ist

und dementsprechend nicht für den Nutzer optimiert wird, hat eine solche Erweiterung nur eine geringe Priorität. Diese Nutzeroberfläche wird selbst bei der Umsetzung in einer möglichst simplen Form gehalten. Dabei sollte es folgende Bestandteile beinhalten:

  - Eine Möglichkeit für den Nutzer eine anderweitig gezeichnete Ziffer interpretieren zu lassen
  - Eine Möglichkeit für den Nutzer eine Ziffer zu zeichnen.
  - Eine (weiter-) interpretierte Ausgabe der Interpretation des CNN.
  - Eine Ausgabe der nicht interpretierten Prozentwerte der Interpretation des CNN.

  Bis zu dem Punkt, wo ein GUI realisiert wurde und in den Einsatz gestellt wird, sind nicht alle dieser Funktionen über die Konsole (das Interface zum Programm, welches vor dem GUI zum Einsatz kommt) verfügbar.

- **GPU als Berechnungsplattform nutzen**

  Erläuterung: Die CPU in einem Computer ist eine sehr „fokussierte“ Hardware. So kommt es dazu, dass diese immer nur einen einzelnen Prozess berechnen kann. Dies ist für ein neuronales Netzwerk, welches Hunderte oder Tausende Knotenpunkte hat und alle einzeln berechnet werden müssen, äusserst hinderlich. Eine dezidierte Grafikkarte, wenn vorhanden, kann diesem Geschwindigkeitsverlust nachhelfen, da eine solche GPU in der Lage ist, tausende Berechnungen gleichzeitig zu tätigen. Da im Projekt mit C++, einer Hardware-nahen Programmiersprache, gearbeitet wird, kann eine Anbindung an das Rechenpotential der GPU implementiert werden. Da dies jedoch ein äusserst schwieriger Prozess ist, sich diese Anbindung bei den unterschiedlichen Herstellern von GPUs unterscheiden kann und nicht notwendig ist, um ein funktionierendes CNN zu erstellen, wurde diese Optimierung als nicht notwendig eingestuft.

1.3.3 Use Cases

Die hier aufgelisteten Use Cases entsprechen den Use Cases nach den Must-haves und sind diesbezüglich ohne die grafische Nutzeroberfläche. Dies kann dazu führen, dass ein endgültiges Produkt nicht mehr kohärent zu den Use Cases steht. Im Allgemeinen sollte aber selbst

eine solche Inkohärenz nicht in extremer Weise auftreten, da die Bedienung der Applikation als Konsole als auch als grafische Oberfläche in ähnlicher Weise auftreten sollte.

1. **Prediction**

    - **Akteur:** Nutzer, CNN
    - **Ablauf:** Siehe Ablaufdiagramm 1.3.5.1 („Interpretation“).
    - **Nachbedingungen:** Der Nutzer sollte in der Konsole eine Auflistung aller möglichen Ziffern (0–9) und deren entsprechende Wahrscheinlichkeit sehen. Zudem kann der Nutzer auch die kongruierende Zahl mit der höchsten Wahrscheinlichkeit auf einer speziellen Zeile in der Konsole ablesen.
    - **Ausnahmen:** Bei der Interpretation von benutzereigenen Abbildungen kann es zu multiplexen Fehlern kommen. Diese reichen von falscher Dateikodierung zu falscher Auflösung. Aufgrund dieser mannigfaltigen Möglichkeiten zu Fehlern können diese nicht alle hier erläutert werden.
    - **Anmerkungen:** Offiziell ist zwar „Training“ keine Vorbedingung von „Testen“, jedoch macht die Anwendung der Applikation keinen Sinn, wenn man nicht erwarten kann, dass man ein realistisches oder sinnvolles Ergebnis erhält.

2. **Training**

    - **Akteur:** Nutzer, CNN
    - **Ablauf:** Siehe Ablaufdiagramm 1.3.5.2 („Training“).
    - **Nachbedingungen:** Der Nutzer erhält nach der Beendigung der Schulung des CNN-Models eine kurze Benachrichtigung, dass das Training abgeschlossen ist. Zudem erhält der Nutzer nach jedem einzelnen Datensatz den Output, dass nun x von y Datensätzen bearbeitet wurden (kann auch als Lade-Balken oder Ähnliches (Ladetext) implementiert werden).
    - **Ausnahmen:** Hierbei kann es zu zwei Fehlern kommen. Dabei handelt es sich um Fehler bei den Datensätzen. Der eine Fehler ergibt sich aus der Möglichkeit, dass CKi die entsprechende Datei aus befindlichen Gründen nicht öffnen kann.

Der andere Fehler ergibt sich, wenn der Datensatz nicht dem Standard der MNIST-Datensätze im UByte-Format entspricht.
Natürlich können auch andere Fehler eintreten, diese können nicht zu diesem Zeitpunkt abgeschätzt werden.

3. **Testing**

   - **Akteur:** Nutzer, CNN
   - **Ablauf:** Siehe Ablaufdiagramm 1.3.5.3 („Testen“).
   - **Nachbedingungen:** Der Nutzer erhält eine Benachrichtigung, dass x von y Datensätzen bearbeitet wurden (kann auch als Lade-Balken oder Ähnliches (Ladetext) implementiert werden). Nach Beendigung erhält der Nutzer die Mitteilung, dass das Testen abgeschlossen ist und einen Prozentwert mit der Genauigkeit des Netzwerkes.
   - **Ausnahmen:** Hierbei kann es zu zwei Fehlern kommen. Dabei handelt es sich um Fehler bei den Datensätzen. Der eine Fehler ergibt sich aus der Möglichkeit, dass CKi die entsprechende Datei aus befindlichen Gründen nicht öffnen kann. Der andere Fehler ergibt sich, wenn der Datensatz nicht dem Standard der MNIST-Datensätze im UByte-Format entspricht. Natürlich können auch andere Fehler eintreten, diese können nicht zu diesem Zeitpunkt abgeschätzt werden.
   - **Anmerkungen:** Bei den Testdaten würde es evtl. Sinn ergeben, die einzelnen Konsolen Outputs auch in einer CSV-Datei festzuhalten, wobei dies mit Pipes in der Konsole dem Nutzer bereits offensteht. Offiziell ist zwar „Training“ keine Vorbedingung von „Testen“, jedoch ergibt das Testen nur begrenzt Sinn, wenn es noch nichts gibt, was sich zu testen lohnt.

1.3.4 Use Case Diagramme

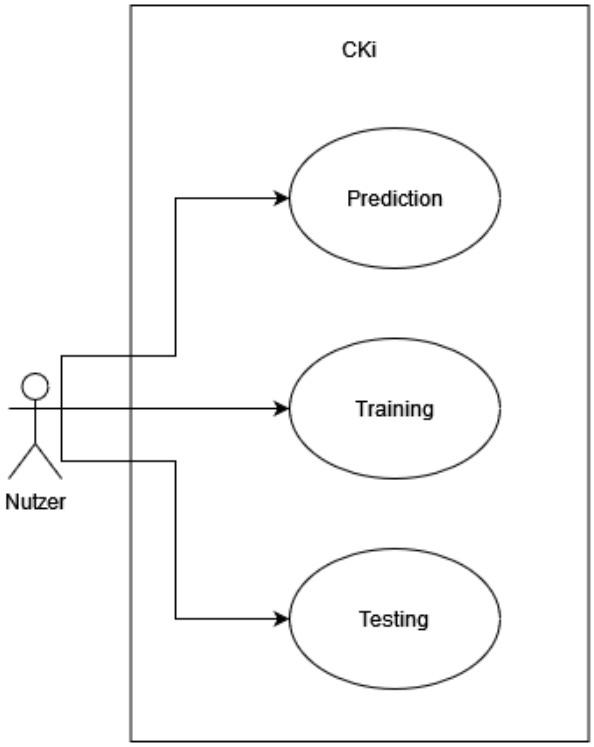


Abbildung 1.2: Use-Case-Diagramm

1.3.5 Ablaufdiagramme

1.3.5.1 Interpretation

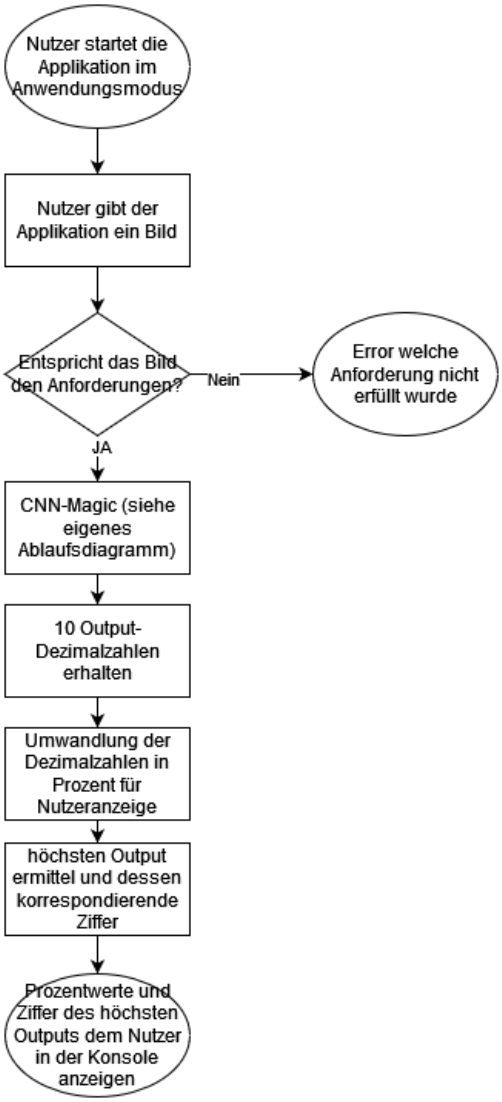


Abbildung 1.3: Ablaufdiagramm für die Interpretation von Nutzer-gelieferten Einzelbildern

1.3.5.2 Training

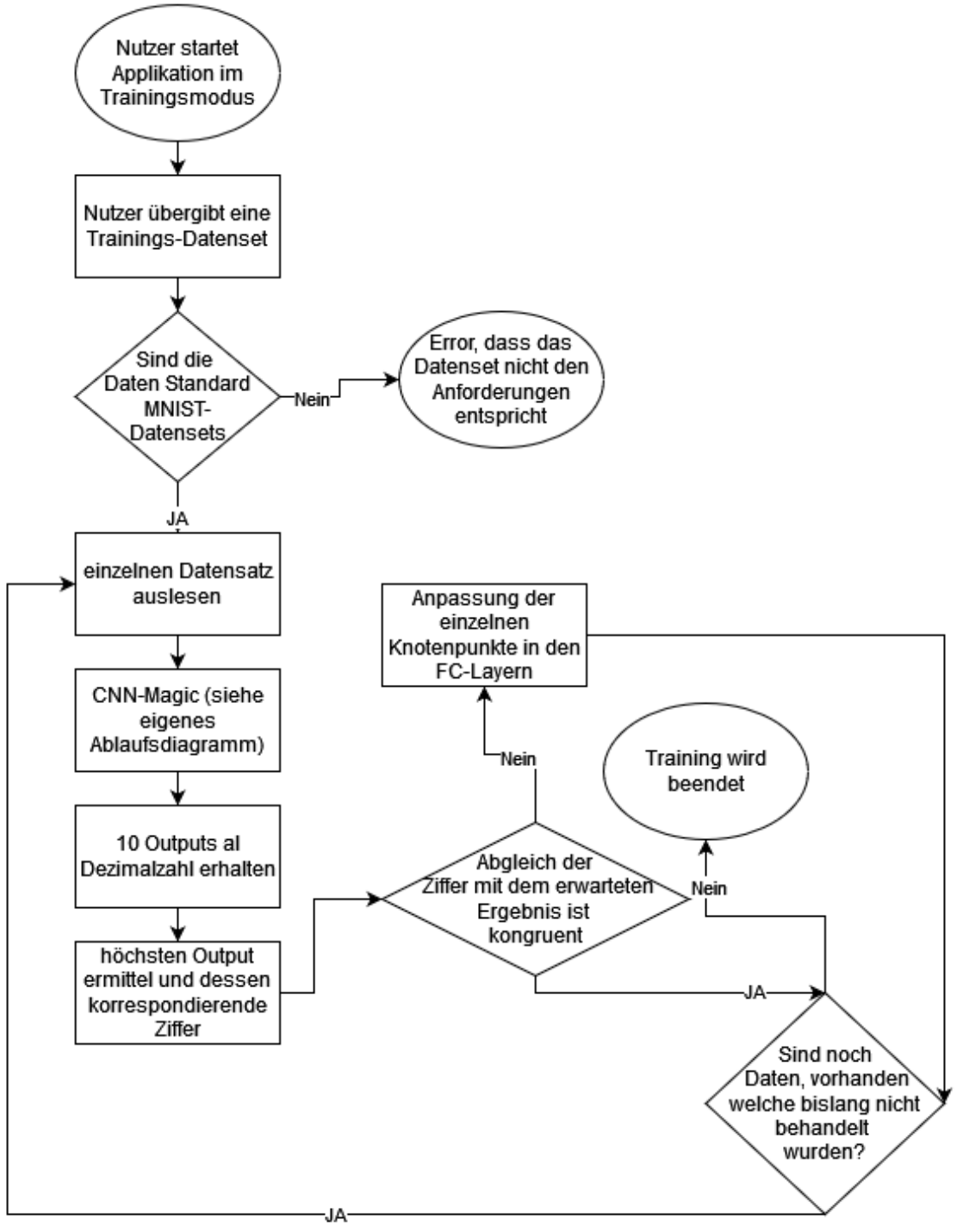


Abbildung 1.4: Ablaufdiagramm vom Training des CNN

1.3.5.3 Testen

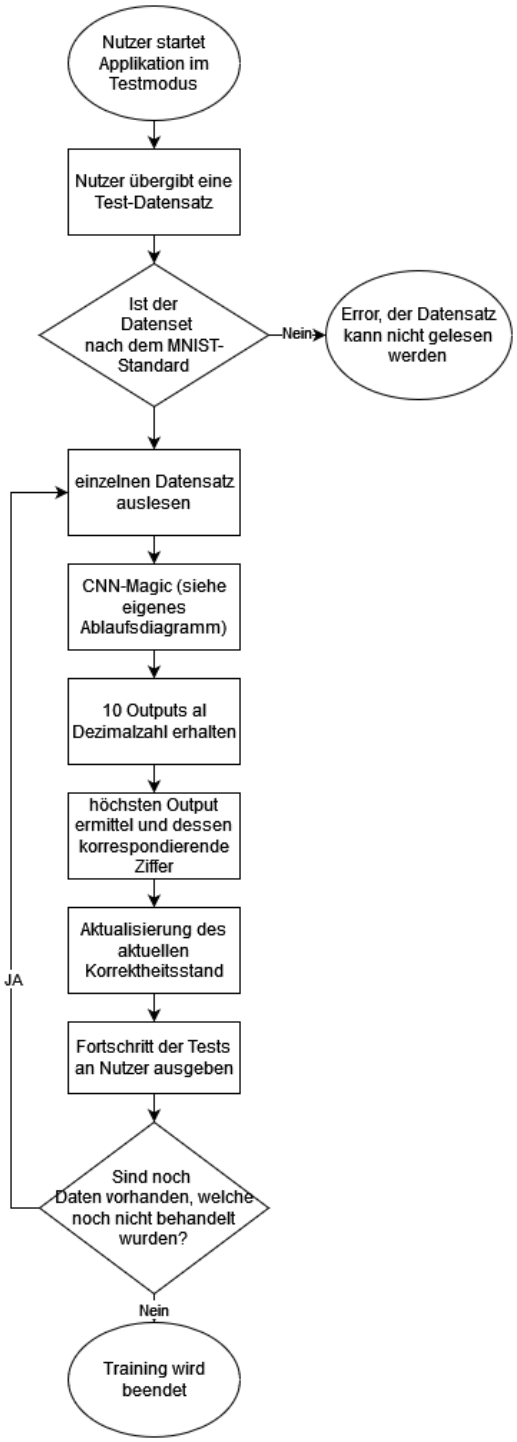


Abbildung 1.5: Ablaufdiagramm von einem Test des CNN

1.3.5.4 CNN-Magie

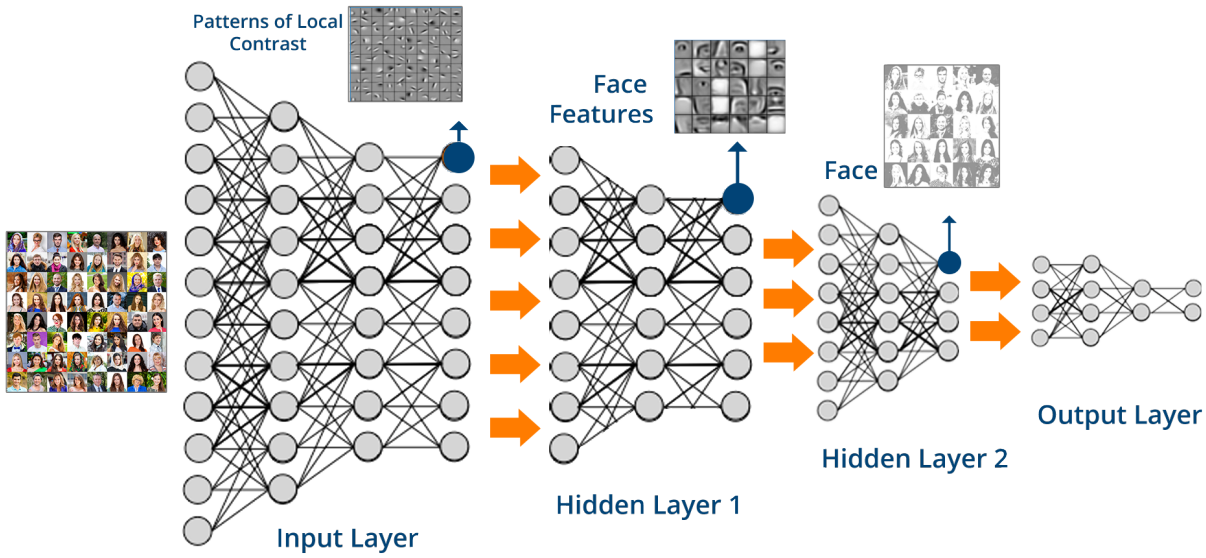


Abbildung 1.6: Visuelle Darstellung der Funktionsweise eines CNN von blog.goodaudience.com

1.4 Umsetzung

Wie im Themenblatt vom 15.09.2023 wird für die Umsetzung reines C++ verwendet.

1.4.1 Entwicklungsumgebung

Bei der Entwicklungsumgebung wird das JetBrains Produkt **CLion** zum Einsatz kommen. Dieses beinhaltet alle benötigten Funktionalitäten, die bei der Entwicklung des Produktes nötig sind. Gegebenen Falles könnte noch **SQLite Browser** zur Verwendung kommen, da dieses ein besseres (und für den Entwickler ein gewohntes) Interface zur Handhabung von lokalen Datenbanken bietet. (Dieses Produkt kam nicht zum Einsatz, da keine Datenbank verwendet wurde.) Für die Version Control des Projektes wird lokal **Git** verwendet und zur Sicherung in der Cloud wird sowohl **GitHub** als auch **GitLab** verwendet. Die Entwicklungsumgebung mit

IDE und Version Control wird primär auf dem Betriebssystem Windows verwendet. Es kann jedoch nicht garantiert sein, dass die Entwicklung nicht teilweise unter einer Distribution von Linux ablaufen wird. Hierbei sollten (evtl. abgesehen von der GUI) keine Kompatibilitätsprobleme auftreten.

1.4.1.1 Auflistung Versionen

- JetBrains Toolbox 2.1.1.18388, Windows 11, x64
- CLion (von JetBrains Toolbox) 2023.2.2
- Git 2.39.2.windows.1
- CMake 3.27.7
- GCC 13.2.0
- MiKTeX 23.10
- TeXnicCenter 2.02

1.4.2 CNN

1.4.2.1 Grundlagen

Der Kern des CNN lässt sich in drei unterschiedliche Arten von Schichten aufteilen. Zusätzlich gibt es noch zwei zusätzliche „Hilfs-“Schichten, welche für die Ein- und Ausgaben zuständig sind. Die *Eingabeschicht* ist eine dieser zwei erwähnten Schichten. Diese akzeptiert im Falle von CKi jeweils ein Bild von speziell definierten Grössen und nimmt jeden Pixel als Graustufen (Dies ist entscheidend, weil so der Farbwert des Pixels von drei zwei Byte grossen Zahlen zu nur einer zwei Byte grossen Zahl reduziert wird.) als Eingabe entgegen. Auf die Eingabeschicht folgt die *Convolutional Layers* oder auch *Faltungsschichten*. Diese Faltungsschichten sind einer der Schlüsselbestandteile eines CNN. Die Convolutional Layer dienen dazu, bestimmte Merkmale und Schlüsselelemente aus dem Bild zu extrahieren. Hierbei wird bei jedem der Convolutional Layer eine mathematische Faltungsoperationen durchführt. Nach

den Faltungsschichten kommen die simplen *Pooling-Schichten*. Bei einer Pooling-Schicht wird die Dimension eines Bildes reduziert (Downsampling). Dies kann durch zwei Arten geschehen, entweder durch Max-Pooling, dabei wird aus einer Liste von Werten nur der höchste übermittelt oder durch Durchschnitts-Pooling, wobei der Durchschnitt dieser Liste berechnet und übermittelt wird. Durch diese Datenreduktion wird Rechenleistung gespart und so das Ergebnis schneller und stabiler geliefert. Vor der Ausgabeschicht gibt es noch die *FC Layers* oder auch *Vollständig verknüpfte Schichten* (*Hidden Layers*). Bei diesen ist jeder Knotenpunkt mit jedem Knotenpunkt in der nächsten Schicht verbunden. Dies ist das Herz des gesamten Models. Dabei wird in jedem Knotenpunkt ein neuer Wert berechnet, um in der letzten Schicht, der *Ausgabeschicht* diese in Wahrscheinlichkeit zu „konvertieren“.[1]

1.4.2.2 Implementation

Bei der Implementation eines neuronalen Netzwerkes wurde für CKi der Entschluss getroffen, eine C++-Klasse für jede benötigte Art von Komponente, die die Applikation benötigt, zu kreieren. Konkret bedeutet dies, dass eine Klasse für das Neuron, eine für den Layer und eine für das Netzwerk entstehen muss. Dabei werden die Klassen aufeinander aufgebaut; ein Netzwerk hat multiple Layer, ein Layer hat multiple Neuronen. Dieses Netzwerk wird danach in die Main-Funktion, den Programmeinstieg, integriert. Dies erleichtert es, in einem späteren Schritt die grosse Umstellung von einer Konsolen-Anwendung hin zu einer grafisch basierten Anwendung (auch wenn immer noch grosse Teile der Main-Klasse ausgetauscht werden müssen). Für die Speicherung der Berechnungswerte für die FC-Layer ist die Abwägung zu treffen, ob es sinnvoll ist, diese in einer lokalen Datenbank zu speichern oder in eine konkret hierfür entworfene Datei zu schreiben.

1.4.3 Testdaten

Die Testdaten für CKi können in unzähliger Ausführung auf Kaggle gefunden werden. Es wird jedoch der Standard von `http://yann.lecun.com/exdb/mnist/` verwendet.

[1] vgl. Pathak, 2023; IBM, 2021; Saha, 2018; 3Blue1Brown, 2017

1.4.3.1 Speicherung

Im gegebenen Fall würde es für die Schulung, des CNN, Sinn ergeben, die einzelnen Bilder nicht über Ordner- oder Archivstrukturen zu speichern, sondern in eine oder mehrere lokale Datenbanken abzuspeichern.
Im Falle von `http://yann.lecun.com/exdb/mnist/` ist dies jedoch nicht notwendig, da diese im UByte-Format vorliegen, wie in 3.2.1 beschrieben.

1.5 Abwägung des Nutzens und der Alternativen

1.5.1 Alternativen

Die möglichen Alternativen zu einer kompletten Realisation eines CNN oder eines neuronalen Netzwerks in C++ ohne Bibliotheken oder Frameworks für neuronale Netze sind endlos. Wenn gleich man bereits bestehende Produkte begutachtet, wird man auf GitHub schnell fündig. Bei einer Suche auf GitHub mit dem Query „mnist digit recogniser“ findet man nach Stand 20.10.2023 02:13 178 Repositorien (je eines in C, C++, C#, Rust, drei in Java + Kotlin und 53 in Python). Auch unter dem Beschluss, dass man das CNN selbst bauen wolle, würde man für jede beliebige Sprache eine Bibliothek oder Framework finden, das einem diese Aufgabe erleichtern würde.

1.5.2 Kosten

Bezüglich der Kosten kann das Projekt CKi nicht mit den Alternativen mithalten. C++ ist keine Sprache, in welcher man schnell ein Produkt hervorbringt. Dies belegen etliche Studien und Artikel (Programming Languages Table, An Empirical Comparison of Seven Programming Languages, Programming Languages by Energy Efficiency, Development time in various languages). Zu dem, dass keine dezidierten Bibliotheken für maschinelles Lernen zum Einsatz kommen, erhöht den Entwicklungsaufwand, Zeitaufwand und so die Kosten. Wenn man die Stundenzahl aus der Planung mit einem Stundensatz von 50 CHF quantifizieren, käme man auf einen gesamten Kostenaufwand von 8’600.- + Lizenzkosten. Zuzüglich kämen noch Hard-

ware Anschaffungen für die Entwicklung hinzu, diese können jedoch im Falle vom Projekt CKi vernachlässigt werden.

Senior Developer	2h	100.00 Fr.
Junior Developer	89h	4’450.00 Fr.
Projektleitung	39h	1’950.00 Fr.
Hardware	-	0.00 Fr.
Software	Lizenz	827.00 Fr.
Möglicher Zusatz-Entwicklungskosten	42h	2’100.00 Fr.
Total		9’427.00 Fr.

1.5.3 Nutzen

Wie bereits im Abschnitt Zielgruppe 1.2 erwähnt, ist dieses Produkt auf den Wissensgewinn ausgerichtet und nicht auf das Produkt selbst. Dementsprechend ist in der geplanten Umsetzung des Projektes CKi der maximale Nutzen gewährleistet. Leider lässt sich dieser Nutzen nicht/schlecht quantifizieren.

1.5.4 Effizienz

Der Vergleich in Effizienz der Anwendung, der aus dem Projekt CKi resultiert und den professionell entwickelten Bibliotheken in Python, etc., wird CKi selbst mit GPU-Berechnung und dem Geschwindigkeitsvorteil von C++ gegen diese Bibliotheken in Bezug auf Run-Time-Length verlieren.

1.5.5 Vergleich

Bestehendes Produkt		Eigene Kreation	
Sektion	Kosten	Sektion	Kosten
Anschaffungskosten	0.- Fr.	Kreation	8’600.- Fr.

Wiederholende Kosten	0.- Fr.	wiederholende Kosten	0.- Fr.
Absolut	0.- Fr.	absolut	8’600.- Fr.

Eine kurze Zusammenfassung der Kosten-Nutzen-Analyse zeigt den Zustand von CKi.
Es werden Kosten für die Arbeitszeit von 8’600 CHF anfallen. Die Effektivität ist niedriger, als wenn dasselbe Produkt mit einer vorhandenen Bibliothek geschrieben werden würde (und die Entwicklungsdauer wäre ebenfalls geringer). Des Weiteren gibt es unzählige kostenlose Alternativen, die es nur zu verwenden gilt.
Der Nutzen des Endproduktes ist nichtig, da dieses nie zur wirklichen Anwendung kommen wird.

1.5.6 Nutzwertanalyse

Bei der Nutzwertanalyse werden unterschiedliche Alternativen systematisch und statistisch nach gewissen Kategorien verglichen. Konkret werden hier unterschiedliche Möglichkeiten der Implementierung miteinander verglichen, dabei wird eine Punkteskala von eins bis zehn verwendet. Wichtig bei der Auflistung ist es, dass eine höhere Punktzahl nicht bedeutet, wie hoch diese Kategorie ist. Eine Entwicklungsdauer von 10 bedeutet nicht, dass diese besonders lang ist, sondern dass diese Option im Bereich Entwicklungsdauer zehn Punkte erhalten hat (also besonders schnell zum implementieren ist).

Kriterien		Python mit TensorFlow		C++	
Entwicklungsdauer	10 %	10	1	2	0.2
Komplexität	15 %	10	1.5	4	0.6
Sprachkenntnisse	10 %	3	0.3	4	0.4
Frust	10 %	5	0.5	7	0.7
Typisierung	5 %	1	0.05	9	0.45
Run-Time-Length	10 %	10	1	9	0.9
Testing	5 %	6	0.3	7	0.35
Objektorientiert	5 %	5	0.25	9	0.45

Wissensgewinn	30 %	2	0.6	10	3
Total	100 %	52	5.5	61	7.05

2 Planung

2.1 Rollenverteilung

Rolle	Person
Leiter	Simeon Stix
Entwickler	Simeon Stix
Betreuer	Sven Nüesch

2.2 Aufgabenliste

1. ***Planung:***
    - Aufgabenliste erstellen
    - Zeiteinteilung
    - Meilensteine definieren
    - Gantt erstellen
2. ***Analyse:***
    - Recherche CNN
    - Recherche Testdaten
    - Recherche C++ Details
    - Anforderungen definieren

- Use Cases
- weitere Analysen
- Analyse schreiben

3. ***Design:***
    - GUI konzipieren
    - Wireframes zeichnen
    - Konsolenbefehle definieren
    - Konsolen-Output definieren
4. ***Realisation:***
    - Input Layer realisieren
    - Convolutional Layer realisieren
    - Pooling Layer realisieren
    - Dense Layer Klasse schreiben
    - Output Layer erstellen
    - CNN-Model erstellen
    - CNN-Model trainieren
    - GUI auf CNN-Model setzen
    - GPU-Anbindung implementieren
    - Kommentare schreiben
    - weitere Verbesserungen vornehmen
    - Puffer-Zeit fürs Programmieren
5. ***Dokumentieren:***
    - Schreiben
6. ***Testen:***

- Testliste kreieren
- Tests implementieren
- Tests durchführen

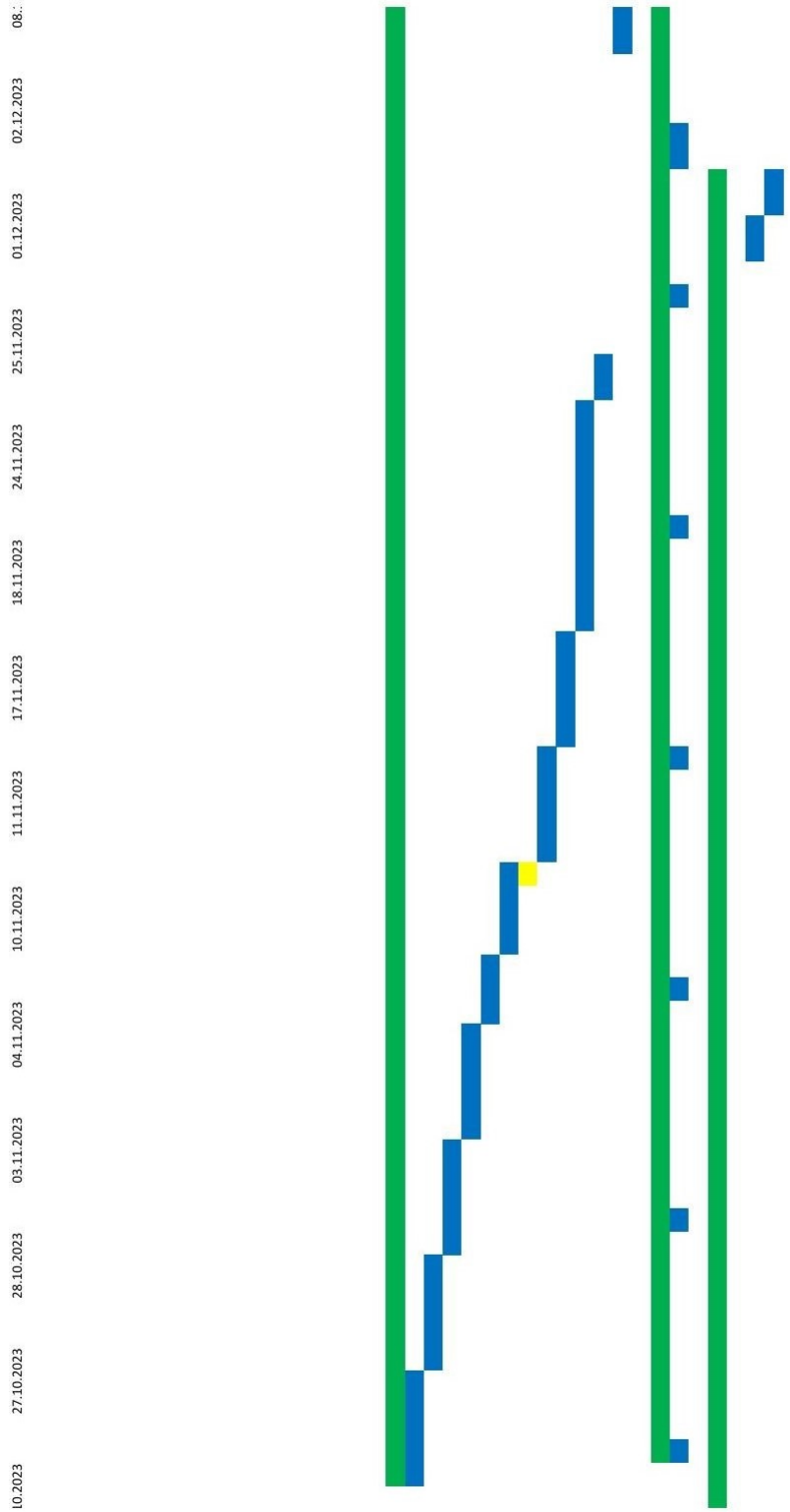
7. ***Deployment:***
   - Projekt als EXE bauen
   - Build-Anleitung kreieren
8. ***Präsentation:***
   - Präsentation erstellen

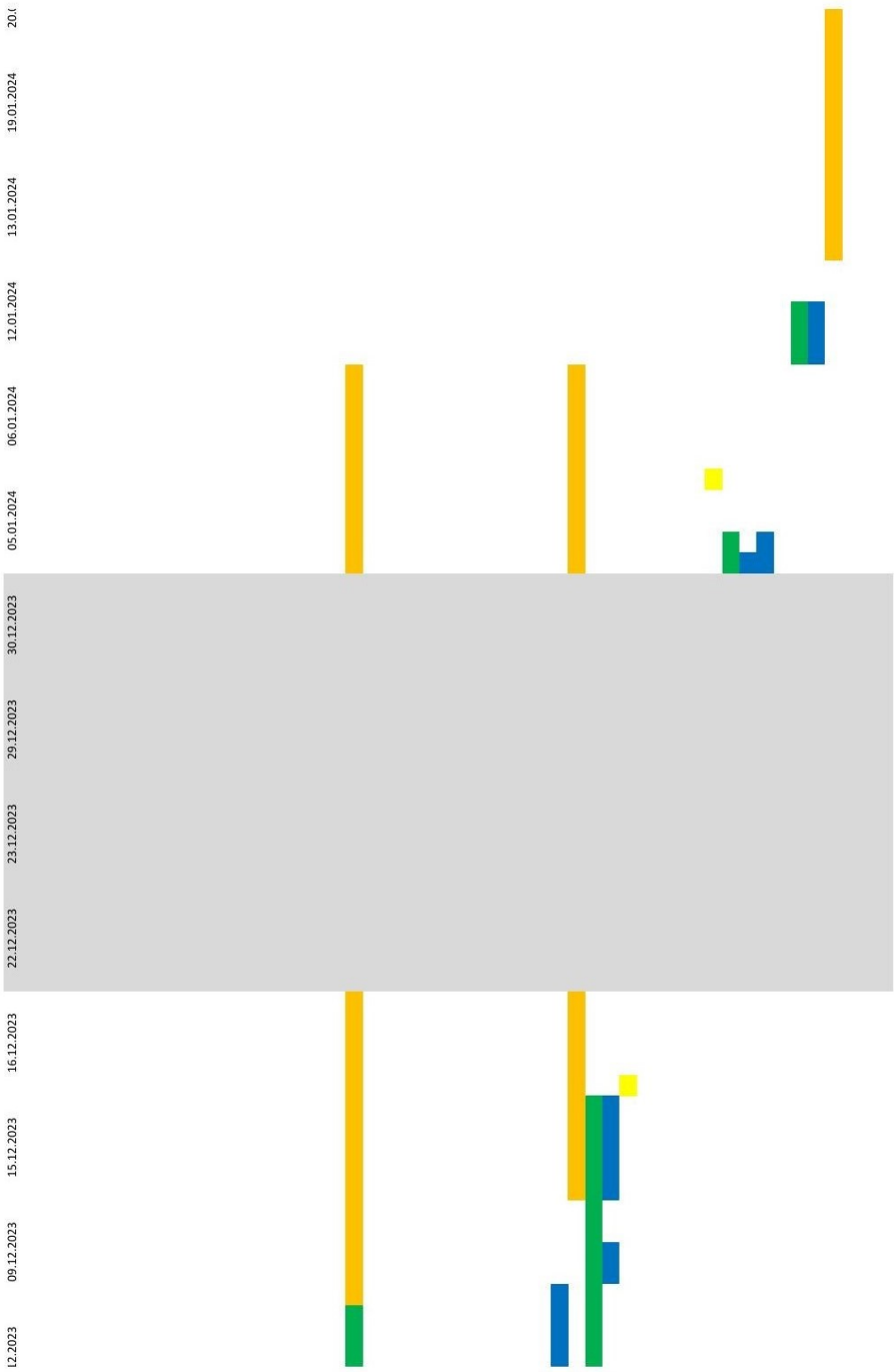
2.3 Meilensteine

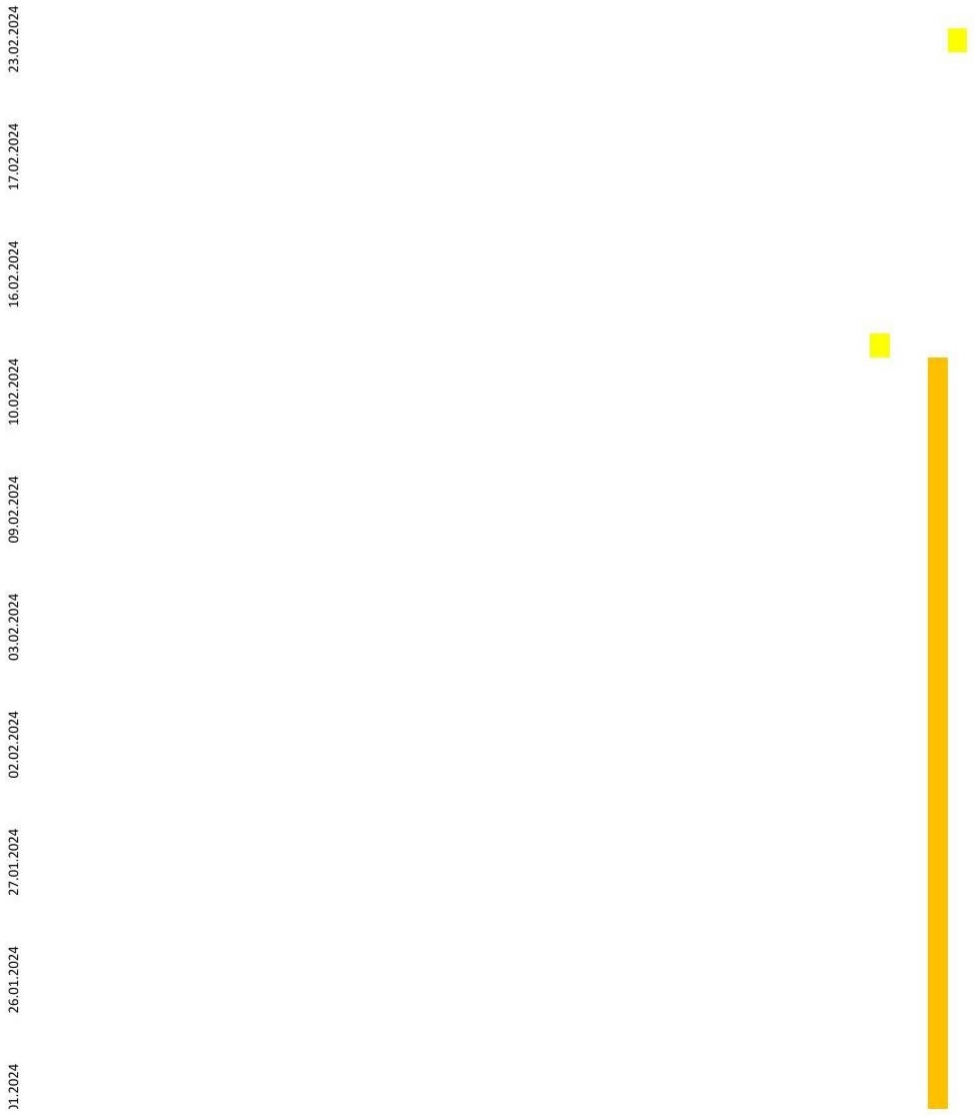
Meilenstein-Name	Datum
Vorbereitungen abgeschlossen	20.10.2023
CNN-Model Kreation abgeschlossen	10.11.2023
Dokumentation abgeschlossen	01.12.2023
Produkt abgeschlossen	05.01.2024
Projekt abgeschlossen	10.02.2024
Projekt abgegeben	23.02.2024

2.4 Gantt

Gross-Tasks	Tasks	Zeit [h]	Datum
Planung		5	
	Taskliste erstellen	1.5	
	Zeiteinteilung	1	
	Meilensteine definieren	0.5	
	Gantt erstellen	2	
Analyse		16	
	Reserche CNN	2	
	Recherche Testdaten	1	
	Recherche C++ Details	2	
	Anforderungen definieren	1	
	Use-Cases	4	
	weitere Analysen	3	
	Schreiben Analyse	3	
Design		5	
	Gui konsipieren	2	
	Wirefames zeichen	2	
	Konsolenbefehle definieren	0.5	
	Konsolen Output definieren	0.5	
Meilenstein	Vorbereitungen abgeschlossen		20.10.2320
Realisation		75	
	Input Layer realisieren	5	
	Convolutional Layer realisieren	5	
	Pooling Layer realisieren	5	
	Dense Layer Klasse schreiben	5	
	Output Layer erstellen	3	
	CNN-Model erstellen	4	
Meilenstein	CNN-Model kreation abgeschlossen		10.11.2023
	CNN-Model trainieren	5	
	GUI auf CNN-Model setzen	5	
	GPU-Anbindung implementieren	10	
	Kommentare schreiben	2	
	weitere Verbesserungen vornehmen	6	
	Bufferzeit Programmieren	20	
Dokumentieren		15	
	Schreiben	15	
Meilenstein	Dokumentation abgeschlossen		16.12.2023
Testen		7	
	Testliste kreieren	2	
	Tests implementieren	3	
	Tests durchführen	2	
Meilenstein	Produkt abgeschlossen		05.01.2024
Deployment		2	
	Projekt als EXE bauen	0.5	
	Build Anleitung kreieren	1.5	
Meilenstein	Projekt abgeschlossen		10.02.2024
Präsentation		3	
	Präsentation erstellen	3	
Buffer Allgemein		44	
Meilenstein	Projekt aggegeben		23.02.2024
Total		172	


08.
02.12.2023
01.12.2023
25.11.2023
24.11.2023
18.11.2023
17.11.2023
11.11.2023
10.11.2023
04.11.2023
03.11.2023
28.10.2023
27.10.2023
10.2023


20.(
19.01.2024
13.01.2024
12.01.2024
06.01.2024
05.01.2024
30.12.2023
29.12.2023
23.12.2023
22.12.2023
16.12.2023
15.12.2023
09.12.2023
12.2023


23.02.2024
17.02.2024
16.02.2024
10.02.2024
09.02.2024
03.02.2024
02.02.2024
27.01.2024
26.01.2024

3 Design

3.1 Konsole

Das Projekt CKi ist in seinem Grundaufbau für die Konsole konzipiert. Der grundlegende Befehl soll wie folgt aussehen:

```
$ ki [options] [file]
```

Dabei gibt es drei Formen von diesem Befehl.

3.1.1 Training

3.1.1.1 Input

```
$ ki --train -l [labels] -i [images] -e [epochs] -lr [learningrate]
$ #Dieser Befehl nimmt ubyte-Dateien.
$ #Die Optionen -e und -lr sind optional.
```

3.1.1.2 Output

Nach jedem Datensatz wird angezeigt, wie viele der zum Training gegebenen Daten bereits abgearbeitet wurden (Dies wird evtl. nicht implementiert.). Am Ende wird ein progressiv berechneter Fehler angezeigt.

3.1.2 Verify

3.1.2.1 Input

```
$ ki --verify -l [labels] -i [images]
$ #Dieser Befehl nimmt ubyte-Dateien.
```

3.1.2.2 Output

Nach jedem Datensatz wird angezeigt, wie viele der zum Testen bereitgestellten Daten bereits abgearbeitet wurden. Zudem wird angezeigt, wie viele der Datensätze richtig, genauer gesagt falsch erkannt wurden (Dies wird evtl. nicht implementiert.). Am Ende wird aber angezeigt, wie akkurat das Netzwerk die Test-Datensätze bewertet hat.

3.1.3 Prediction

3.1.3.1 Input

```
$ ki [file]
$ #Dieser Befehl nimmt jpg-Dateien & png-Dateien.
```

3.1.3.2 Output

Als Output werden alle Ziffern von 0 bis 9 mit den entsprechenden Prozentwerten zurückgegeben (Dies kommt daher, dass die KI die Wahrscheinlichkeit der Übereinstimmung für jede Ziffer berechnet.). Zudem wird auch angezeigt, welche Ziffer die höchste Übereinstimmung hat, da diese die erkannte Ziffer darstellt.
Eine interessante Erweiterung wäre es, bei diesem Befehl (oder in einer leicht abgewandelten Form) ein kleines Fenster zu öffnen, um dort direkt die Zahl zu zeichnen. Dies hätte den Vorteil, dass Parameter wie Grösse direkt bekannt und kontrolliert werden können.

3.1.4 Help

3.1.4.1 Input

```
$ ki --help
$ #Dieser Befehl liefert eine Liste an Befehlen.
```

3.1.4.2 Output

Dieser Befehl liefert eine Liste an Befehlen, die in der Konsole verwendet werden können.

3.2 Datenbank

Die Applikation benötigt keine Datenbank. Zur Speicherung der Test- und Trainings-Datensätze werden sogenannte ubyte-Dateien eingesetzt. Diese enden auf der Dateierweiterung „.ubyte“.

3.2.1 UByte

Die Bilddateien im MNIST-Datensatz werden im sogenannten „ubyte“ (unsigned byte) Format gespeichert. Dies bedeutet, dass die Pixelwerte der Bilder als Bytes gespeichert werden. Jedes Bild im Datensatz wird als Reihe von Bytes dargestellt, wobei jedes Byte den Graustufenwert eines Pixels repräsentiert. Hinzu kommt, dass auch die dazugehörigen Labels in einer „ubyte“-Datei gespeichert werden.
Aufgrund dieser Speicherung müssen die Datensets vor deren Verwendung eingelesen und interpretiert werden.
Für diese Interpretation muss die standardisierte Höhe und Weite für jedes auszulesende Bild multipliziert werden. Danach müssen so viele Bytes aus der entsprechenden „ubyte“-Datei ausgelesen und als vorzeichenlose ganze Zahlen gespeichert werden, denn jede dieser Zahlen stellt einen Pixel dar.
Für eine genaue Anleitung für das Einlesen und Interpretieren in Python ist die Webseite AndroidKT zu kontaktieren.[1]

[1] vgl. LeCun & Cortes & Burges, 2002; AndroidKT, 2023

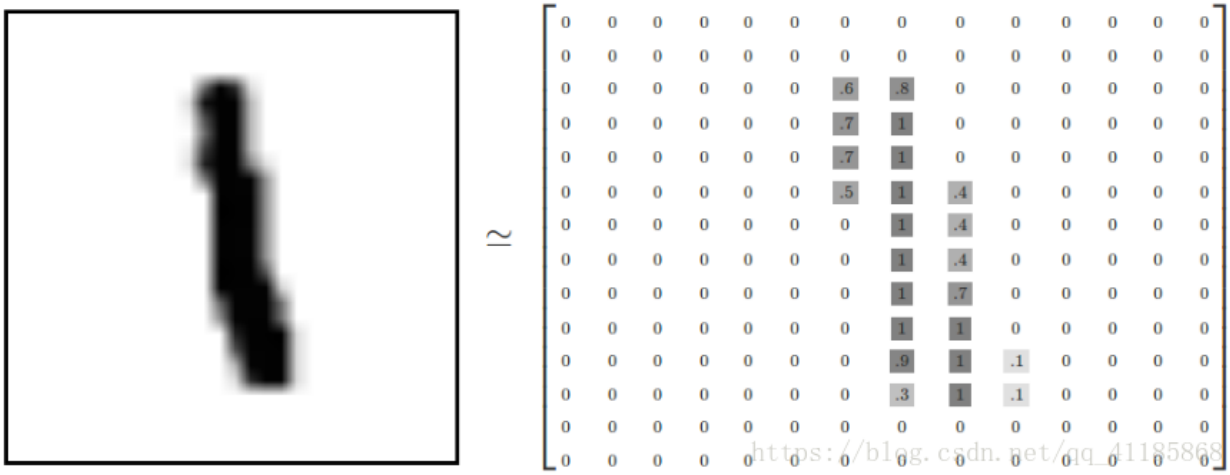


Abbildung 3.1: Darstellung einer Ziffer und deren Code von blog.csdn.net

3.3 Code

3.3.1 Klassendiagramm

Es gibt vier Klassen im Projekt CKi. Dabei handelt es sich um auf sich selbst aufbauende Strukturen. Als Code-Einstieg fungiert die Main-Klasse, welche die Main-Funktion bereitstellt. In dieser wird die Klasse „Network“ aufgesetzt. Es werden mehrere Hidden-Layer erstellt, die Datensätze der MNIST-Datenkollektion eingelesen und die Funktionen „Train“, „Verify“ und „Predict“ sowie „Load Network“ ausgeführt.

In der Network-Klasse werden mehrere Layer initialisiert und verwendet. Diese Layer sind in der Layer-Klasse definiert und besitzen wiederum multiple Neuronen.

Die Neuronen sind die Knotenpunkte, welche mit der mathematischen Sigmoid-Funktion einen Output-Wert berechnen. Zusätzlich ermitteln diese auch die Fehler-Abweichung bei der Back-propagation, um die einzelnen Werte anzupassen.

Um Sigmoid und die Ableitung von Sigmoid zu berechnen, gibt es noch eine Utility-Klasse „Util“. Diese besteht nur aus statischen Funktionen, welche an anderen Orten benötigt werden.

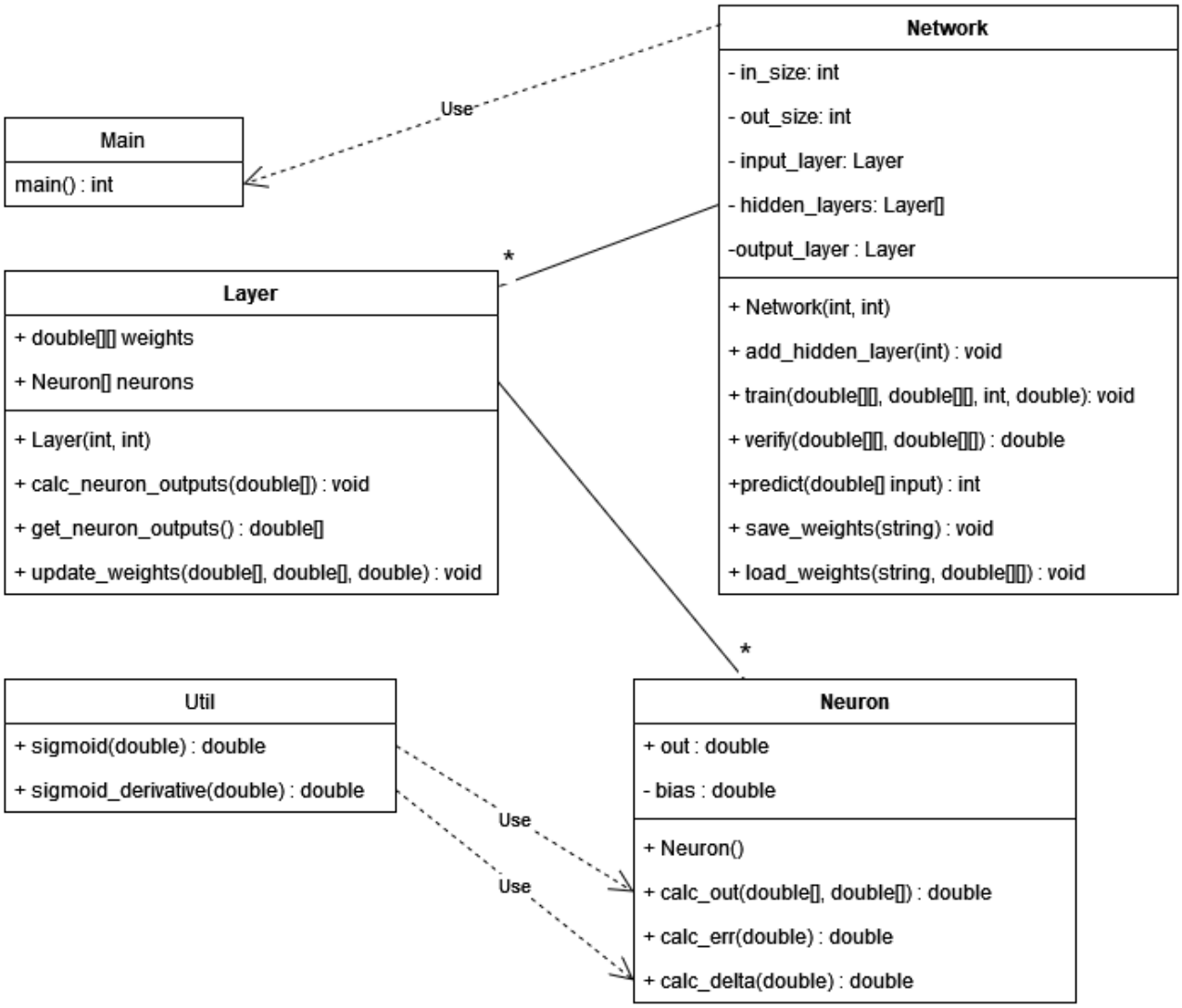


Abbildung 3.2: Das Klassendiagramm, Grundaufbau der Applikation (überholt)

3.3.2 Trainingsdaten

Die Trainingsdaten sind das MNIST-Datenset mit den handschriftlichen Zahlen. (`http://yann.lecun.com/exdb/mnist/`)

3.3.3 Tests

Im Projekt CKI gibt es drei Arten von Tests. Es gibt das simple Ausprobieren. Da es nicht allzu viele Nutzerschnittstellen gibt, kann man diese ausführen und begutachten, ob diese mit der Beschreibung übereinstimmen. Bei einer dieser Schnittstellen wird die KI getestet. Dies geschieht, indem die KI ihr unbekannten Datensätze zu sehen bekommt und das Endergebnis mit einem vordefinierten Ergebnis abgeglichen wird. Egal, ob das Ergebnis korrekt oder inkorrekt erkannt worden ist, wird es statistisch aufgenommen. Am Ende wird dem Nutzer eine Prozentzahl der korrekten Erkenntnisse präsentiert. Diesbezüglich ist dies kein Test, in dem die Applikation versagen könnte, es ist eine reine Leistungsüberprüfung, ob mehr Training vonnöten ist. Zusätzlich zu diesen zwei Testmöglichkeiten gibt es noch die Unit-Tests. Diese werden dazu genützt, um einzelne Funktionen und Klassen noch vor deren Verwendung zu überprüfen.
Zusätzlich zu diesen drei Tests wäre es überaus interessant zu begutachten und zu vergleichen, wie hoch die Leistungsunterschiede zwischen CKi und einem in Python mit TensorFlow gebauten Programm mit vergleichbarer Grundstruktur sind. Diesbezüglich wird eventuell neben CKi auch ein kleines vergleichbares Modell in Python realisiert, um diese Daten zu sammeln.

4 Realisation

4.1 Allgemein

Die Realisation des Projekts CKI zeichnet sich durch die Implementierung eines modularen neuronalen Netzwerks aus, das für Aufgaben wie die Erkennung handschriftlicher Ziffern konzipiert wurde. Zum Einsatz kamen dabei Standard-C++-Technologien sowie eine CMake-basierte Projektstruktur für das Build-Management. Die Kernstruktur besteht aus Klassen für Neuron, Layer, und Network, die die Basis des Netzwerks bilden, ergänzt durch eine Util-Klasse für Hilfsfunktionen, wie Aktivierungsfunktionen und Datenverarbeitung. Zusätzlich gibt es im Ordner „Test“ Unit-Tests mit den entsprechenden Attrappen für die MNIST-Datensätze. Die Unit-Tests sind mit Google-Tests implementiert.

4.2 Abhängigkeiten

Das Projekt CKI kommt mit nur wenigen Abhängigkeiten aus. Die, die es dennoch gibt, sind umso wichtiger für das erfolgreiche Ausführen der Applikation.
Abhängigkeiten:

1. **googletest:**

   Version (Tag): v1.14.0

   Git: `https://github.com/google/googletest`

   Beschreibung: GoogleTest ist ein C++-Framework für Unit-Tests, das Assertions für die Überprüfung von Code und Funktionen für die Organisation und Ausführung von Tests bietet.

2. **nlohmann/json:**

Version (Tag): v3.11.3

Git: `https://github.com/nlohmann/json`

Beschreibung: Nlohmann/json ist eine moderne, header-only C++ Bibliothek für die Verarbeitung von JSON-Daten, die einfache Integration und intuitive Nutzung bietet.

3. **wichtounet/mnist:** (nicht mehr in Verwendung)

   Version (Commit): 3b65c35

   Git: `https://github.com/wichtounet/mnist`

   Beschreibung: Wichtounet/mnist ist ein einfacher C++-Reader für den MNIST-Datensatz, der es ermöglicht, Trainings- und Testbilder sowie Labels zu lesen und zu verwenden.

4. **nothings/stb:**

   Version (Commit): 5736b15

   Git: `https://github.com/nothings/stb`

   Beschreibung: STB ist eine Sammlung plattformübergreifender Header-Dateien für C, die umfassende Funktionen für Grafik, Audio und Textverarbeitung ohne externe Abhängigkeiten bereitstellt.

4.3 Architektur

4.3.1 Klassen

4.3.1.1 Klassendiagramm

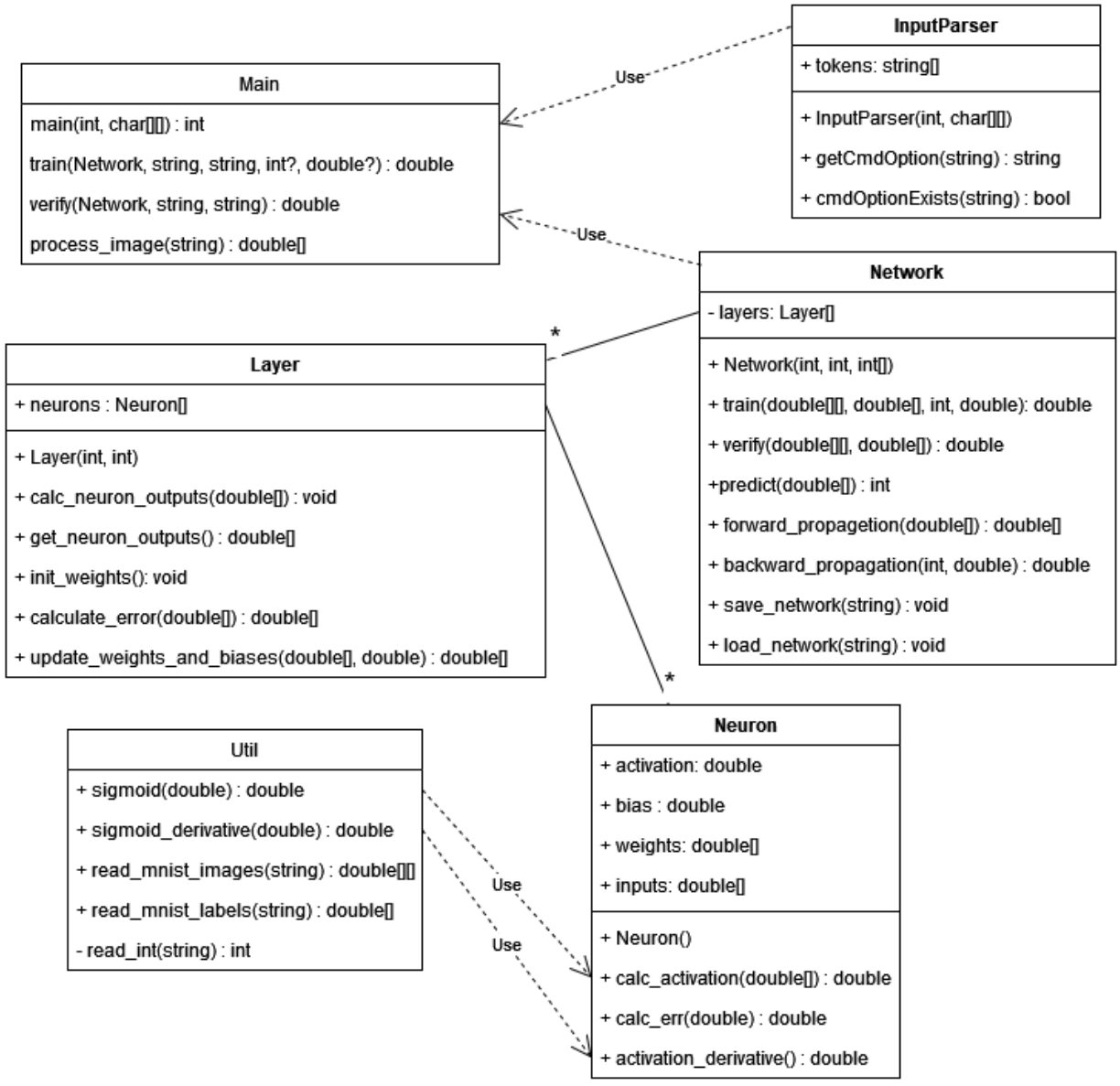


Abbildung 4.1: Klassendiagramm nach der Realisation

Das ist das Klassendiagramm, wie die Applikation des Projektes CKI nach der Implementation aussieht. Somit sind die nötigen Anpassungen, die seit der Designphase zu treffen waren, ersichtlich.
Wie zu sehen ist, gibt es etliche markante Änderungen in den Klassen Netzwerk und Layer. Dies kommt daher, dass in der Designphase aufgrund fehlendem oder unzureichendem Wissen die Struktur angepasst werden musste. Wie diese Klassen funktionieren sowie die wichtigsten Funktionen sind in den Abschnitten 4.3.1, bis 4.4.3 zu finden.
Interessant ist die Klasse InputParser, welche beim ursprünglichen Entwurf nicht anzutreffen war. Diese Klasse ist zuständig, einzelne Attribute aus dem Aufruf der CKI-Main-Funktion auszulesen und eine einfache Verwendung dieser zu gewährleisten. Dabei ist diese Klasse nicht selbst implementiert, sondern stammt von `https://stackoverflow.com/questions/865668/parsing-command-line-arguments-in-c`. Dementsprechend wird im weiteren Verlauf nicht weiter auf diese Funktion eingegangen.

4.3.1.2 Netzwerk

Die Network-Klasse repräsentiert das neuronale Netzwerk. Es unterstützt die Initialisierung des Netzwerks mit einer bestimmten Anzahl von Eingabe- und Ausgabeneuronen sowie eine variable Anzahl von versteckten Schichten und deren Grössen. Die Klasse bietet Funktionen für das Training des Netzwerks mit gegebenen Eingaben und Zielwerten, die Überprüfung der Netzwerkleistung und Vorhersagen für neue Eingaben. Dies ist möglich wegen der Durchführung von Vorwärts- und Rückwärtspropagation, die wichtigsten Algorithmen im Bereich des maschinellen Lernens (Mehr dazu unter Algorithmen 4.4.2).
Zusätzlich existieren noch zwei Funktionen in der Klasse, welche zuständig sind für die Speicherung und das Laden des Netzwerks in bzw. aus einer Datei.

4.3.1.3 Layer

Die Layer-Klasse definiert die Layer im neuronalen Netzwerk, bestehend aus mehreren Neuronen. Sie bietet Funktionen zum Initialisieren und Setzen von Gewichten, Berechnen der Ausgaben der Neuronen basierend auf Eingaben, Berechnen der Fehler im Vergleich zu den Zielwerten und der Aktualisierung von Gewichten und Biases auf Basis von Fehlern und Lern-

rate.Jedes Layer-Objekt enthält eine Liste von Neuron-Objekten, die die Neuronen in diesem Layer repräsentieren.

4.3.1.4 Neuron

Die Neuron-Klasse stellt ein einzelnes Neuron dar, inklusive seiner Gewichte, Bias, Eingaben, Aktivierungsfunktion und Summe der gewichteten Eingaben. Sie bietet Funktionen zur Berechnung der Aktivierung basierend auf den Eingaben, der Ableitung der Aktivierungsfunktion, der Fehlerberechnung im Vergleich zu einem Zielwert, sowie zur Speicherung der Gewichte und des Biases.
Zur Berechnung der Aktivierung und deren Ableitung werden die statischen Utility-Funktionen Sigmoid und die Ableitung von Sigmoid aus der Utility-Klasse genutzt (Siehe auch 4.4.3.7).

4.3.1.5 Utility

Die Util-Klasse bietet statische Hilfsfunktionen für das neuronale Netzwerk, darunter die Sigmoid-Aktivierungsfunktion und ihre Ableitung sowie Funktionen zum Lesen von MNIST-Bilddaten und Labels aus Dateien. Diese Hilfsfunktionen sind essenziell für die Vorverarbeitung von Eingabedaten und die Implementierung der Lernmechanismen im Netzwerk.

4.3.2 Ordnerstruktur

Im Kern des Projekts stehen die Hauptdateien, welche die essenziellen Klassen wie Neuron, Layer, Network und Util enthalten. Diese sind grundlegend für die Funktionalität des neuronalen Netzwerks. Die CMakeLists.txt unterstützt das Build-Management, vereinfacht die Kompilierung und Konfiguration. Der .gitignore sorgt dafür, dass unnötige Dateien und Ordner nicht in die Versionskontrolle einfliessen. Ein speziell dafür vorgesehener Test-Ordner beinhaltet Tests zur Überprüfung der Funktionalität, was die Zuverlässigkeit des Systems sicherstellt. Zusätzlich runden Dummy-Files und ein eigener Ordner für die MNIST-Datasets das Projekt ab, indem sie das Vorhandensein der Datensets für Training und Tests des Netzwerks garantieren.
Das CKI-Projekt ist strukturiert in:

- **Hauptdateien:** (kein Ordner, Rootverzeichnis) Enthalten die Klassen Neuron, Layer, Network, und Util für die Kernlogik des neuronalen Netzwerks.
- **CMakeLists.txt:** (Im Rootverzeichnis) Für das Build-Management erleichtert das Kompilieren und die Konfiguration des Projekts.
- **.gitignore:** (Im Rootverzeichnis) Definiert Dateien und Ordner, die von Git-Versionierung ausgeschlossen sind, wie Build-Artefakte und IDE-spezifische Dateien.
- **Test-Ordner:** Beinhaltet Testfälle zur Überprüfung der Funktionalität einzelner Komponenten und des Gesamtsystems.
  - **Dummy-Files:** Zusätzliche Dateien für Testzwecke.
  - **Unittests:** Tests zur Sicherstellung korrekter Funktionalität, Isolation von Fehlern.
- **MNIST-Datasets Ordner:** Speichert die Datensätze für das Training und Testen des Netzwerks, insbesondere für die Erkennung handschriftlicher Ziffern.

4.4 Code

4.4.1 Konzept

Das Projekt CKI nutzt grundlegende Konzepte des maschinellen Lernens wie Vorwärts- und Rückwärts-Propagierung, Aktivierungsfunktionen (hier Sigmoid) und die Anpassung von Gewichten während des Trainingsprozesses. Die modulare Struktur, unterteilt in Schichten, Neuronen und Hilfsfunktionen, sowie die reine objektorientierte Implementation in C++ ohne externe Bibliotheken, abgesehen von Bibliotheken für Tests und Datenhandling, zeichnen die Architektur der Applikation aus.
Um mehr über die hier erwähnten Funktionen zu erfahren, ist der Abschnitt 4.4.2 zu konsultieren.

4.4.2 Algorithmen

4.4.2.1 Forward-Propagation

Die Forward-Propagation ist ein grundlegender Prozess in neuronalen Netzwerken, der es ermöglicht, Vorhersagen auf Basis von Eingabedaten zu treffen. Dabei werden die Eingabedaten durch das Netzwerk von der Eingabeschicht über eine oder mehrere versteckte Schichten bis zur Ausgabeschicht vorwärts geleitet. Jede Schicht besteht aus Neuronen, die über Gewichte mit den Neuronen der vorherigen Schicht verbunden sind. Die Daten werden in jedem Neuron durch eine Summationsfunktion verarbeitet, die die gewichteten Eingaben aufsummiert und einen Bias-Wert hinzufügt hat. Das Ergebnis dieser Summation wird dann durch eine Aktivierungsfunktion geleitet, um die Ausgabe des Neurons zu bestimmen.
Die Aktivierungsfunktion bestimmt, wie Neuronen ihre Eingaben in Ausgaben umwandeln und ist entscheidend für die Fähigkeit des Netzwerks, komplexe Muster in den Daten zu erkennen. Beliebte Aktivierungsfunktionen sind die Sigmoid-, Tanh- und ReLU-Funktion.[1]
Sobald die Eingabedaten durch das Netzwerk propagiert worden sind und die Ausgabeschicht erreicht haben, wird das Ergebnis mit dem tatsächlichen Wert verglichen, um den Fehler der Vorhersage zu bestimmen. Dieser Fehler wird dann in einem separaten Prozess, der als Backpropagation bekannt ist, verwendet, um die Gewichte im Netzwerk anzupassen und die Vorhersagegenauigkeit zu verbessern.[2]

4.4.2.2 Back-Propagation

Die Backpropagation, kurz für „backward propagation of errors“, ist ein Schlüsselmechanismus im Training neuronaler Netzwerke. Dieser Algorithmus ermöglicht es, die Gewichte des Netzwerks so anzupassen, dass der Gesamtfehler bei der Vorhersage minimiert wird. Backpropagation wird nach der Forward-Propagation angewendet, nachdem eine Vorhersage durch das Netzwerk gemacht und der Fehler zwischen der Vorhersage und dem tatsächlichen Wert berechnet wurde.
Der Prozess der Backpropagation besteht aus zwei Hauptphasen: der Berechnung des Gradi-

[1] vgl. Li & Johnson & Yeung, 2017; databasecamp, 2023
[2] vgl. Nüesch, 2023; 3Blue1Brown, 2017;

enten des Fehlers bezüglich aller Gewichte im Netzwerk und der anschliessenden Anpassung dieser Gewichte in die Richtung, die den Fehler minimiert. Der Gradient gibt an, in welche Richtung die Gewichte verändert werden müssen, um den Fehler zu verringern, und die Grösse der Anpassung wird durch die Lernrate bestimmt.
Backpropagation nutzt die Kettenregel der Differenzialrechnung, um die Fehlergradienten für die Gewichte jeder Schicht vom Ausgang zurück zum Eingang effizient zu berechnen. Der berechnete Fehlergradient für jede Gewichtung zeigt, wie eine kleine Änderung in diesem Gewicht den Gesamtfehler beeinflusst. Durch die systematische Anpassung der Gewichte basierend auf diesen Gradienten kann das Netzwerk schrittweise verbessert werden, um genauere Vorhersagen zu liefern.
Insgesamt ermöglicht Backpropagation das effiziente Training tiefer neuronaler Netzwerke, indem es systematisch die Netzwerkgewichte anpasst, um den Fehler zwischen den Vorhersagen des Netzwerks und den tatsächlichen Werten zu minimieren, was zu einer verbesserten Modellleistung führt.[3]

4.4.3 Schlüsselpassagen & Snippets

Die wichtigsten Funktionen in einem neuronalen Netzwerk sind die beiden Propagation, vorwärts als auch rückwärts. Wobei diese das Grundgerüst für die drei Hauptfunktionen, Train, Verify & Predict bilden.

4.4.3.1 Train

```
1 double Network::train(std::vector<std::vector<double>> &inputs, std::vector<↩
    ↪ double> &labels, int epochs, double learning_rate){
2     double total_error = 0;
3     for(int epoch = 0; epoch<epochs; epoch++ ){
4         for(std::size_t i = 0; i < inputs.size(); i++){
5             std::vector<double> outputs = Network::forward_propagation(inputs[i↩
                ↪ ]);
6             total_error += Network::backward_propagation(labels[i], ↩
                ↪ learning_rate);
```

[3] vgl. Nüesch, 2023; 3Blue1Brown, 2017; Karpathy, 2016; Roy, 2022; Li & Johnson & Yeung, 2017;

```
            total_error /= 2;
        }
    }
    return total_error;
}
```

Die Funktion, welche für das Lernen des neuronalen Netzwerks verantwortlich ist, ist simpel aufgebaut. Sie erhält primär zwei Listen: eine Liste aus Bildern und eine Liste aus Beschriftungen für die Bilder. Danach nutzt Train die Funktionen der Forward- und Back-Propagation, um zuerst das Bild durch das Netzwerk bewerten zulassen und dann diese Bewertung mithilfe der Beschriftungen zu korrigieren. Dies tut sie für den gesamten Datensatz oder durch eine zukünftige Erweiterung in einzelnen Blöcken (für höhere Effektivität) und der Anzahl Epochen entsprechend.

4.4.3.2 Verify

```
double Network::verify(const std::vector<std::vector<double>> &inputs, const std↩
    ↪ ::vector<double> &labels){
    int correct = 0;
    for (std::size_t i = 0; i < inputs.size(); ++i){
        if (Network::predict(std::vector<double>(inputs[i])) == static_cast<int↩
            ↪ >(labels[i])){
            correct++;
        }
    }
    return static_cast<double>(correct) / static_cast<double>(inputs.size());
}
```

Die Funktion ist eigentlich gleich aufgebaut wie „Train“, jedoch mit dem markanten Unterschied, dass keine Back-Propagation zum Einsatz kommt. So werden die Bewertungen des neuronalen Netzwerks nur als richtig oder falsch bewertet und nicht korrigiert.

4.4.3.3 Predict

```
int Network::predict(const std::vector<double> &input){
```

```
    std::vector<double> outputs = Network::forward_propagation(input);
    return static_cast<int>(std::distance(outputs.begin(), std::max_element(↩
        ↪ outputs.begin(), outputs.end())));
}
```

Diese Funktion erhält nur ein einziges Bild als Attribut. Dieses Bild wird durch die Forward-Propagation bewertet. Die Bewertung wiederum besteht nur aus einer Liste von zehn Prozentzahlen (für die Ziffern von 0 bis 9) und damit diese besser für einen Menschen lesbar wird, muss die höchste Prozentzahl aus der Liste gesucht werden. Deren Index wird dann als errechnete Zahl zurückgegeben.
(Diese Funktion ist ein Zusammenzug aus der Predic-Funktion und Logic aus der main.cpp-Datei.)

4.4.3.4 Forward-Propagation

```
std::vector<double> Network::forward_propagation(const std::vector<double>& ↩
    ↪ input){
    layers[0].calc_neuron_outputs(input);

    for (int i = 1; i < layers.size(); ++i){
        std::vector<double> inputs = layers[i - 1].get_neuron_outputs();
        layers[i].calc_neuron_outputs(inputs);
    }

    return layers[layers.size()-1].get_neuron_outputs();
}
```

In dieser Funktion wird eine Liste von grossen Fliesskommazahlen übernommen. Diese sind die einzelnen Pixel, in dem zu bearbeiteten Bild, welche nur einen Helligkeitswert beinhalten. Diese Pixel werden an den ersten Layer im Netzwerk übergeben. Dieser kalkuliert durch die Sigmoid-Funktion neue Ausgabewerte. Die Ausgabewerte werden mit der For-Schleife durch alle Schichten des Netzwerks weitergereicht, bis diese am Ende ausgelesen und als Rückgabewert dieser Funktion genutzt werden. Die Ausgabewerte haben zwar das gleiche Datenformat, unterscheiden sich jedoch anhand ihrer Grösse (Länge der Liste) und den tatsächlichen

Werten.

4.4.3.5 Back-Propagation

```
double Network::backward_propagation(const int& target, double learning_rate){
    std::vector<double> inputs (10, 0);
    inputs[target] = 1;
    std::vector<double> error = layers[layers.size()-1].calculate_error(inputs);

    double total_error = 0;
    for(double& e : error) { //remove for better performance
        total_error += e;
    }

    total_error/=error.size();

    //std::cout << total_error << std::endl;

    for (int i = layers.size() - 1; i >= 0; --i){
        error = layers[i].update_weights_and_biases(error, learning_rate);
    }
    return total_error;
}
```

Im Gegensatz zur Forward-Propagation wird in der Back-Propagation ein Wert nicht von der ersten Schicht zur letzten übergeben, sondern Rückwerts von dem letzten Layer bis zur Eingabe zurück.
Dabei muss zuerst die Abweichung von der erhaltenen Ausgabe (aus der Forward-Propagation) mit der erwarteten Ausgabe abgeglichen und der Fehler berechnet werden. Nun wird ähnlich zur Forward-Propagation dieser Fehler durch die einzelnen Schichten geschickt und dort verwendet, um die Gewichtungen und Biases für jedes Neuron anzupassen. Dies passiert in einer eigenen Funktion im Layer.

4.4.3.6 Anpassung von Gewichtungen und Biases

```
1  std::vector<double> Layer::update_weights_and_biases(const std::vector<double>& ↩
       ↪ error, double learning_rate){
2      std::vector<double> prev_layer_error(Layer::neurons[0].weights.size(), 0.0);
3
4      for (size_t i = 0; i < Layer::neurons.size(); ++i){
5          Neuron &neuron = Layer::neurons[i];
6          const double neuron_error = error[i];
7          const double activation_derivative = neuron.activation_derivative();
8
9          for (size_t j = 0; j < neuron.weights.size(); ++j){
10             prev_layer_error[j] += neuron.weights[j] * neuron_error;
11
12             const double delta_weight = neuron_error * activation_derivative * ↩
                   ↪ neuron.inputs[j];
13             neuron.weights[j] -= learning_rate * delta_weight;
14         }
15         neuron.bias += learning_rate * neuron_error * activation_derivative;
16     }
17     return prev_layer_error;
18 }
```

Dies ist die Funktion, die in der Back-Propagation aufgerufen wird.

Dabei wird für jedes Neuron für jede Gewichtung eine eigene Anpassung gesucht. Um dies zu erreichen, werden mehrere Werte benötigt: Der Fehler des Neurons, der Input des Neurons, die Ableitung der Aktivierung und die Learningrate. Nun können die Änderungen bei der Gewichtung und dem Bias berechnet werden. Die Abweichung der Gewichtung ist ein Produkt des Inputs in diesem Neuron, dem Fehler, der Ableitung der Sigmoid-Funktion und der Learningrate. Die Korrektion für das Bias ist fast identisch zu der Abweichung der Gewichtung. Es ist ebenfalls ein Produkt der gleichen Faktoren, abgesehen von der Learningrate und dem Input.

Zusätzlich kann der Fehler für den nächsten Layer berechnet werden. Dieser wird benötigt, um die Backpropagation auf den nächsten Layer auszudehnen und dort dieselben Schritte auszuführen. Dabei setzt sich der Fehler für den nächst oberen Layer zusammen, aus dem Fehler und dem entsprechenden Gewicht für das Neuron. Dabei wird der Fehler neu gewich-

tet und zusammen mit allen anderen neu-gewichteten Fehlern für dieses eine Neuron auf der nächsten Ebene aufsummiert.

4.4.3.7 Sigmoid & Ableitung

```
double Util::sigmoid(double x){
  return 1.0 / (1.0 + exp(-x));
}

double Util::sigmoid_derivative(double x){
  double s = sigmoid(x);
  return s * (1.0 - s);
}
```

Das sind die mathematischen Implementierungen der Sigmoid-Funktion und deren mathematischen Abweichungen. Diese Funktionen sind zuständig für die einzelnen Aktivierungen der Neuronen und verschieben eine Zahl in den Wertebereich zwischen 0 und 1.

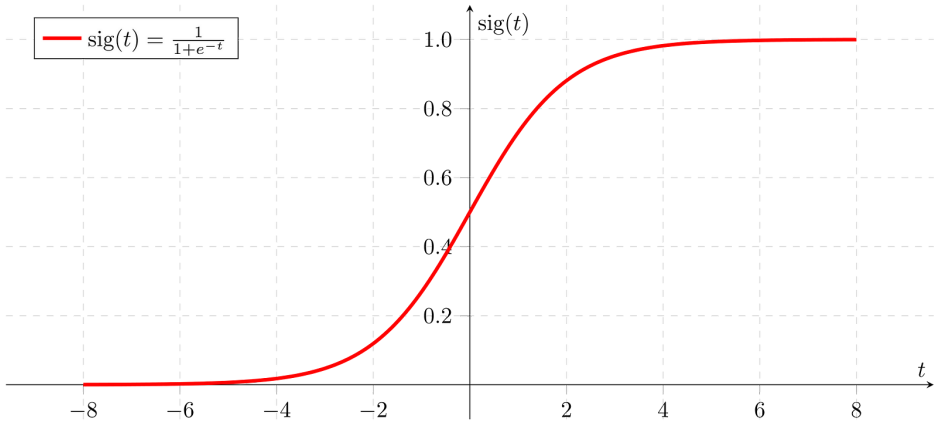


Abbildung 4.2: Sigmoid Funktion

4.4.3.8 MNIST-Reader

```
std::vector<std::vector<double>> Util::read_mnist_images(const std::string &↩
    ↪ filename){
    std::ifstream file(filename, std::ios::binary);
    if (file.is_open()){
```

```
        int magic_number = read_int(file);
        int number_of_images = read_int(file);
        int number_of_rows = read_int(file);
        int number_of_columns = read_int(file);

        std::vector<std::vector<double>> images(number_of_images, std::vector<↩
            ↪ double>(number_of_rows * number_of_columns));

        for (int i = 0; i < number_of_images; ++i){
            for (int r = 0; r < number_of_rows * number_of_columns; ++r){
                unsigned char temp = 0;
                file.read(reinterpret_cast<char*>(&temp), sizeof(temp));
                images[i][r] = (double)temp / 255.0; // Normalizing pixel values↩
                    ↪ to [0, 1]
            }
        }
        return images;
    }else{
        throw std::runtime_error("Cannot open file: " + filename);
    }
}
```

Um die Daten aus den von `http://yann.lecun.com/exdb/mnist/` stammenden UByte-Dateien auszulesen, müssen gewisse Byte-Werte aus den Dateien ausgelesen werden. Die hier aufgeführte Funktion dient dazu Bilder aus diesen Dateien auszulesen und ist beispielhaft auch für die Labels.

Die Implementierung ist nach dem Muster von `http://yann.lecun.com/exdb/mnist/` implementiert. So sind in den ersten vier Byte gespeichert, wie viele Bilder/Beschriftungen im Datensatz zu finden sind. Um dies auszulesen, ist in einer eigenen Funktion implementiert (read_int). Nach den ersten vier Byte müssen die restlichen Bytes in eine Liste eingetragen, in eine ganze Zahl interpretiert (zwischen 0 und 255) und auf den Wert zwischen 0 und 1 gebracht werden. Man spricht auch von der Normalisierung der Pixel-Werte.

4.4.3.9 Ersten 4 Bytes

Dies ist die oben angesprochene „read_int“-Funktion.

```
int Util::read_int(std::ifstream &file){
    unsigned char bytes[4];
    file.read(reinterpret_cast<char*>(bytes), sizeof(bytes));
    return (int)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | bytes↩
        ↪ [3]);
}
```

Die Funktion wurde mit der Hilfe von Stack Overflow unter dem Link `https://stackoverflow.com/questions/604431/c-reading-unsigned-char-from-file-stream` erstellt.
Zuerst wird ein Array in der Grösse von vier Bytes erstellt und vier Bytes aus einer Datei eingelesen. Danach werden die vier Bytes zu einer einzigen ganzen Zahl zusammengesetzt. Dies geschieht, indem die Bytes um jeweils 8 Bits nach links verschoben werden und somit das erste Byte die acht höchsten Stellen der ganzen Zahl symbolisiert, danach das zweite Byte und so folgend die restlichen Bytes.

4.4.3.10 Bild Vorverarbeitung

```
std::vector<double> process_image(const char* filename) {
    int width, height, channels;
    unsigned char* img = stbi_load(filename, &width, &height, &channels, 0);
    if (img == nullptr) {
        std::cerr << "Error in loading the image" << std::endl;
        exit(1);
    }

    unsigned char* resized_img = new unsigned char[28 * 28 * channels];
    stbir_resize_uint8(img, width, height, 0, resized_img, 28, 28, 0, channels);

    std::vector<double> image_data(28 * 28);
    for (int i = 0; i < 28 * 28; ++i) {
        int j = i * channels;
```

```
15          double gray = 0.299 * resized_img[j] + 0.587 * resized_img[j + 1] + ↩
                ↪ 0.114 * resized_img[j + 2];
16          image_data[i] = gray / 255.0; // Normalize to [0, 1]
17      }
18
19      stbi_image_free(img);
20      delete[] resized_img;
21
22      return image_data;
23  }
```

Dies Funktion lädt ein Bild von der angegebenen Datei, skaliert es auf 28x28 Pixel herunter und konvertiert es in Graustufen. Zuerst wird das Bild geladen, wobei Breite, Höhe und Kanäle des Bildes erfasst werden. Falls das Laden fehlschlägt, wird eine Fehlermeldung ausgegeben und das Programm beendet. Anschließend wird das Bild auf die Größe 28x28 Pixel verkleinert, wobei die Anzahl der Kanäle beibehalten wird. Danach wird für jeden Pixel ein Graustufenwert berechnet, indem eine gewichtete Summe seiner RGB-Werte gebildet und durch 255 geteilt wird, um den Wert auf den Bereich [0, 1] zu normalisieren. Diese Werte werden gespeichert und am Ende der Funktion zurückgegeben. Zum Schluss werden der Speicher des ursprünglichen und des skalierten Bildes freigegeben, um Speicherlecks zu verhindern.
Diese Funktion wird verwendet, um in der Ausführung von Predict das Bild des Nutzers zu übernehmen und auf das geforderte Format zu verkleinern.

5 Testing

5.1 Unittests

Beim Unit-Testing wird Code durch Code überprüft. So kann die Integrität des Projektes von einer einzelnen Funktion zu ganzen Klassen vom Grund auf überprüft werden.
In CKI werden die Klassen für das Neuron, den Layer und das Netzwerk getestet. Zusätzlich gibt es Tests zu den meisten Funktionen der Utility-Klasse.
Dabei werden Unit-Tests in CKI mit der „Google-Tests“ Bibliothek von Google implementiert.

5.1.1 Util (Utility-Klasse)

5.1.1.1 Sigmoid-Funktionstests

... prüfen, ob die Sigmoid- und ihre Ableitungsfunktion die erwarteten Werte zurückgeben.

5.1.1.2 Dateilese-Tests

... überprüfen das korrekte Lesen eines Integers aus einer Binärdatei und das korrekte Verhalten beim Versuch, nicht vorhandene Dateien zu lesen.

5.1.1.3 MNIST-Daten-Tests

... überprüfen das Einlesen von MNIST-Bild- und Label-Daten, einschliesslich der Überprüfung der Anzahl und der Normalisierung von Bildern sowie der Überprüfung der Gültigkeit von Labels.

5.1.2 Network

5.1.2.1 Index in akzeptiertem Bereich

... überprüft, ob die predict-Methode einen gültigen Index innerhalb des erwarteten Bereichs zurückgibt.

5.1.2.2 Training beeinflusst das Verhalten.

... testet, ob das Training des Netzwerks dessen Verhalten ändert, indem es die Ausgaben vor und nach dem Training vergleicht. Es wird festgestellt, dass sich die Ausgaben verändern sollten, was auf eine Modifikation des Netzwerks hinweist.

5.1.2.3 Netzwerkspeicher-Tests

... überprüfen die Funktionalität zum Speichern und Laden des Netzwerks. Nach dem Speichern und Laden wird erwartet, dass das Netzwerk wie vorgesehen funktioniert.

5.1.3 Layer

5.1.3.1 Initialisierung

... überprüft die korrekte Initialisierung der Neuronen in der Schicht mit der richtigen Anzahl von Gewichten.

5.1.3.2 Gewichtungen setzen

... testet das Setzen und Abrufen der Gewichte eines Neurons.

5.1.3.3 Output kalkulieren

... überprüft die Berechnung der Ausgänge der Neuronen basierend auf gegebenen Eingaben und gesetzten Gewichten.

5.1.3.4 Fehler kalkulieren

... testet die Fehlerberechnung der Schicht basierend auf den Ausgängen und den erwarteten Werten.

5.1.3.5 Änderung der Gewichtungen und Biases

... überprüft das Aktualisieren der Gewichte und Biases der Neuronen basierend auf einem berechneten Fehler.

5.1.4 Neuron

5.1.4.1 Aktivierungsfunktion

... überprüft, ob das Neuron die richtige Ausgabe liefert, wenn eine bestimmte Eingabe gegeben wird.

5.1.4.2 Ableitung der Aktivierungsfunktion

... untersucht, ob die Ableitung der Aktivierungsfunktion korrekt berechnet wird.

5.1.4.3 Fehlerberechnung

Der Fehler wird durch den Unterschied zwischen dem erwarteten und dem tatsächlichen Ausgang des Neurons bewertet.

5.2 Integrationstests

Es gibt keine Integrationstests, da keine (externen) Komponenten verwendet werden, auf denen das Projekt beruht und deren Verwendung nicht über Unittests abgedeckt werden.

5.3 Deployment-Tests

Es gibt keine Deployment-Tests, da dieses Produkt nicht ausgelegt wurde an die breite Öffentlichkeit weitergegeben zu werden. Somit werden keine Deployment-Tests benötigt, da jeder, der dieses Projekt verwenden möchte, dieses selbst „bauen“ muss.

6 Deployment

6.1 Building

Die Erstellung des Projektes CKI lässt sich in drei Schritte unterteilen. Es gibt jedoch einige Anforderungen.

6.1.1 Anforderungen

Das Projekt CKI ist zwar plattformunabhängig, kann also unter allen gängigen Betriebssystemen verwendet werden, muss jedoch erst für die entsprechende Plattform erstellt werden. Für diesen Erstell-Prozess sind folgende Anforderungen (Programme) unabdingbar:

- CMake
- C++ Compiler
- Internet

Je nach Betriebssystem oder IDE können schon alle oder einige dieser Anforderungen erfüllt sein.

6.1.1.1 CMake

CMake ist ein plattformübergreifendes Tool zur Automatisierung des Build-Prozesses, das es ermöglicht, Makefiles und Projekte für verschiedene Entwicklungsumgebungen zu generieren. Es verwendet eine Konfigurationsdatei „(*CMakeLists.txt*)“, um den Build-Prozess zu steuern. Unter Windows kann CMake von der offiziellen Website heruntergeladen und entweder über einen grafischen Installer oder über die Kommandozeile installiert werden.

Unter Linux und Mac kann CMake über den Package Manager der Wahl installiert werden. Die häufigsten sind Pacman, APT oder RPM unter Linux und Brew unter Mac.
Für das Projekt CKI wird CMake benötigt, um die Build-Konfiguration zu erstellen, externe Abhängigkeiten zu verwalten und das Projekt für verschiedene Entwicklungsumgebungen vorzubereiten, wodurch eine konsistente und effiziente Entwicklung ermöglicht wird. Dabei ist bei der Installation notwendig, dass die CMake Version kompatibel mit der **Version 3.26**, wie in der CMakeLists.txt spezifiziert, ist.

6.1.1.2 C++-Compiler

Ein C++-Compiler ist ein Software-Tool, das C++-Code in maschinenlesbaren Code übersetzt, sodass Programme ausgeführt werden können.
Unter Windows, Mac und Linux kann ein C++-Compiler durch die Installation einer Entwicklungsumgebung wie Visual Studio oder durch direkte Installation von GCC oder Clang eingebunden werden.
Für das Projekt CKI ist ein Compiler notwendig, da CMake, das für das Erstellen des Projektes verwendet wird, auf einen Compiler angewiesen ist, um den C++-Code in ausführbare Dateien zu übersetzen. CMake generiert Build-Konfigurationen, aber der eigentliche Kompilierungsprozess benötigt einen C++-Compiler. Dabei ist bei der Installation notwendig, dass der C++-Compiler den **C++-Standard 17** unterstützt, da das Projekt CKI auf diesen Standard aufgebaut wurde und moderne C++-Features nutzt.

6.1.1.3 Internet

Das Projekt CKI benötigt Internet, um externe Abhängigkeiten wie *googletest* für UnitTests und *nlohmann_json* für JSON-Verarbeitung automatisch herunterzuladen und zu integrieren. Diese Bibliotheken sind essenziell für das Funktionieren und Testen des Projektes. CMake verwaltet diesen Prozess, indem es die benötigten Pakete aus dem Internet lädt, was eine effiziente und konsistente Set-up-Umgebung über verschiedene Entwicklungsplattformen hinweg ermöglicht.

6.1.2 Ausführung

Um das Projekt CKI zu bauen und auszuführen, gibt es verschiedene Wege, die wegen unterschiedlicher Entwicklungsumgebungen stark variieren können. Dementsprechend ist hier nur beschrieben, wie CKI mithilfe von CMake, einem C++ Compiler und Internet in der Konsole gebaut werden kann.
Es folgen nun die notwendigen Schritte, um dies zu bewerkstelligen:

1. Erstellen Sie ein Build-Verzeichnis im Root-Verzeichnis des Projektes. Dies kann über ein grafisches Interface passieren oder über die Konsole. Der entsprechende Befehl unter Windows, Linux und Mac sollte „*mkdir*“ sein.

2. Wechseln Sie in das Build-Verzeichnis (Der entsprechende Befehl unter Windows, Linux und Mac sollte „*cd*“ sein.) und führen Sie „*cmake ..*“ aus, um die Build-Konfiguration zu generieren. Dieser Befehl variiert nicht unter den unterschiedlichen Betriebssystemen, setzt allerdings voraus, dass CMake installiert wurde. Siehe mehr unter Anforderungen 6.1.1.

3. Führen Sie anschliessend „*cmake –build .*“ aus, um das Projekt zu kompilieren. Dies setzt voraus, dass ein C++ Compiler installiert wurde. Siehe mehr unter Anforderungen 6.1.1.

4. Wenn Sie nun den Inhalt des Build-Verzeichnisses begutachten, sollten Sie unterschiedlichste Dateien und Verzeichnisse vorfinden, aber auch eine Datei namens CKI mit einer Endung einer ausführbaren Datei (unter Windows würde die Datei *CKI.exe* heissen). Diese ausführbare Datei ist das compilierte und fertige Produkt des Projektes CKI.

6.2 Installation

Dieser Abschnitt setzt voraus, dass die Schritte aus dem Abschnitt Building 6.1 erfolgreich ausgeführt wurden und das Produkt des Projekts CKI bereit zur Ausführung im Build-Verzeichnis zu finden ist.

6.2.1 Installation mit CMake

Nach dem Build-Vorgang können Sie das Projekt CKI installieren, indem Sie einen Installationsbefehl über CMake ausführen. Dieser Vorgang kopiert die erforderlichen ausführbaren Dateien, Bibliotheken und eventuell weitere Ressourcen in vordefinierte Verzeichnisse. Normalerweise wird dies durch den Befehl „*cmake –install .*“ im Build-Verzeichnis erreicht. Stellen Sie sicher, dass Sie über die erforderlichen Berechtigungen verfügen, um Installationen in den Zielverzeichnissen durchführen zu können. Die genaue Konfiguration des Installationspfades und anderer Parameter kann in der „*CMakeLists.txt*“ festgelegt werden. Diese Konfiguration muss jedoch vom Installierenden vorgenommen werden, da das Projekt CKI nicht für eine langfristige Installation gedacht ist, da es per se keinen Mehrwert für einen Endnutzer bringt.

6.2.2 Installation ohne CMake

Achtung! Dieser Vorgang wurde nur flüchtig unter Windows getestet und kann daher nicht für Linux oder Mac garantiert werden.

Nach dem Build-Vorgang kann die Datei, welche verantwortlich für die Ausführung ist (unter Windows *CKI.exe*), in ein beliebiges anderes Verzeichnis kopiert werden. Unter Windows sollte das Produkt des Projekts CKI (*CKI.exe*) ausführbar bleiben; für Linux und Mac kann aufgrund fehlender Versuchen keine Aussage getroffen werden.

6.2.3 Hinzufügung von vortrainierten Gewichtungen und Biases

Um im Projekt CKI, einer Installation oder einem Build, vorhandene Gewichtungen und Biases für das neuronale Netzwerk hinzuzufügen, wird eine JSON-Datei benötigt, welche dem Netzwerk als Speicher für die Gewichtungen und Biases dient. Diese Datei liegt standardmässig im Root-Verzeichnis der installierten oder gebauten Applikation (also im gleichen Verzeichnis wie die ausführbare Datei/Applikation).
Somit können alle Gewichtungen und Biases zwischen unterschiedlichen neuronalen Netzwerken (welche für die gleiche Aufgabe trainiert wurden) ausgetauscht werden. Dabei gilt allerdings zu beachten, dass das Netzwerk gleich aufgebaut sein muss. Das heisst, die Netzwerke

benötigen die gleiche Anzahl an Layer und Neuronen pro Layer. Es muss die „ load_network“ -Funktion ausgeführt werden, vor der Verwendung des Netzwerkes und die JSON-Datei muss richtig benannt sein.
Sollte die Standard-Version der Main-Datei verwendet werden, heisst diese Datei „network.json“ und wird bei Nichtvorhandensein automatisch beim ersten Trainingsvorgang erstellt. Somit wird mindestens beim Erststart ein Fehler erscheinen, dass die JSON-Datei nicht gelesen werden konnte. Dieser kann allerdings ignoriert werden, wenn keine vorhandenen Gewichtungen und Biases manuell zum Netzwerk hinzugefügt worden sind.

6.3 Verwendung

Das Projekt CKI nutzt ein neuronales Netzwerk, um mit dem MNIST-Datensatz zu arbeiten, einem Standarddatensatz für Handschrifterkennung. Dabei bietet es eine flexible Anwendung für maschinelles Lernen mit Fokus auf Bilderkennung. Es ermöglicht Benutzern, ein Convolutional Neural Network (CNN) mit dem MNIST-Datensatz für Handschrifterkennung zu trainieren, zu verifizieren und Vorhersagen für einzelne Bilder zu treffen. Durch Kommandozeilenargumente kann der Benutzer wählen, ob das Netzwerk trainiert, dessen Genauigkeit überprüft oder ein Bild klassifiziert werden soll.

6.3.1 Interaktion

Um über die CLI mit CKI zu interagieren, muss der Benutzer das Programm mit spezifischen Kommandozeilenargumenten ausführen. Diese Argumente steuern die Aktionen des Programms, wie das Trainieren des Netzwerks, die Überprüfung seiner Genauigkeit oder die Klassifizierung eines spezifischen Bildes. Eine genaue Definition der Befehle ist im Abschnitt Design 3.1 zu finden. Der Übersichtlichkeit sind diese aber hier nochmals zusammengefasst:

1. Training

   - **Befehl:**

```
$ cki --train -l [labels] -i [images] -e [epochs] -lr [learningrate]
```

- **Verwendung:** Trainiert das Netzwerk mit Bildern und Labels. Die Optionen für Epochen (-e) und Lernrate (-lr) sind optional. -l und -i benötigen einen Dateipfad zu den entsprechenden UByte-Dateien.

2. Verifizierung

   - **Befehl:**

```
$ cki --verify -l [labels] -i [images]
```

   - **Verwendung:** Überprüft die Genauigkeit des trainierten Netzwerks mit einem Testdatensatz.-l und -i benötigen einen Dateipfad zu den entsprechenden UByte-Dateien.

3. Vorhersage

   - **Befehl:**

```
$ cki [file]
```

   - **Verwendung:** Macht eine Vorhersage für ein einzelnes Bild. Das Bild kann im JPG- oder PNG-Format sein.

4. Hilfe

   - **Befehl:**

```
$ cki --help
```

   - **Verwendung:** Zeigt eine Liste aller verfügbaren Befehle an.

Literaturverzeichnis

[1] 3Blue1Brown. Backpropagation-kalkül | kapitel 4, deep learning. 3Blue1Brown, 03.11.2017. Accessed on 19 February 2024.

[2] 3Blue1Brown. Aber was ist ein neuronales netzwerk? | kapitel 1, deep learning. 3Blue1Brown, 15.10.2017. Accessed on 19 February 2024.

[3] AndroidKT. Extract images from mnist idx3 ubyte file format in python. AndroidKT, 11.06.2023. Accessed on 23 November 2023.

[4] databasecamp. Was ist die relu-funktion (rectified linear unit)? Database-Camp, 26.04.2023. Accessed on 19 February 2024.

[5] stimms Dave Jarvis. Development time in various languages. Stack Overflow, 12.12.2009. Accessed on 12 February 2024.

[6] ibm. What are convolutional neural networks? IBM, 16.12.2021. Accessed on 23 November 2023.

[7] Paul Hudak; Mark P. Jones. Haskell vs. ada vs. c++ vs awk vs. ... an experiment in software prototyping productivity. Yale Edu, 10.1994. Accessed on 12 February 2024.

[8] Andrej Karpathy. Yes you should understand backprop. Medium, 19.12.2016. Accessed on 19 February 2024.

[9] Sven Nüesch. *Artificial Intelligence (AI)*. KS Frauenfeld, 07.04.2023. Only access over Sven Nüesch.

[10] Amrita Pathak. Convolutional neural networks (cnns): Eine einführung. Geekflare, 24.09.2023. Accessed on 23 November 2023.

[11] Abhijit Roy. An introduction to gradient descent and backpropagation. Towards Datascience, 14.06.2020. Accessed on 19 February 2024.

[12] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. Towards Datascience, 15.12.2018. Accessed on 23 November 2023.

[13] Ph.D. Stephen F. Zeigler. Comparing development costs of c and ada. Ada vs C, 30.03.1995. Accessed on 12 February 2024.

[14] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. MNIST, 22.06.2002. Accessed on 23 November 2023.

[15] Fei-Fei Li; Justin Johnson; Serena Yeung. Backpropagation and gradients. Standford, 13.04.2017. Accessed on 19 February 2024.

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis