Ernst-Göbel-Schule Abitur 2022

Besondere Lernleistung

im Fach Informatik

Thema:

Modellierung und Implementierung künstlicher neuronaler Netze am Beispiel von Erkennung handgeschriebener Ziffern mit Hilfe von C++

Verfasser: Eugen Bondarev

Betreuende Lehrer: Herr Voith, Herr Garrels

Bearbeitungszeit: Oktober 2021 - April 2022

Abgabetermin: 25. April 2022

Inhaltsverzeichnis

1 Einleitung

1.1 Über die Struktur dieser BLL

Diese Arbeit ist der Hauptteil der von mir erstellten besonderen Lernleistung, die außerdem ein in der Programmiersprache C++ entwickeltes Programm und ein Arbeitstagebuch beinhaltet.

1.2 Was diese Arbeit umfasst

- Erklärung der wichtigen Grundbegriffe
- Problemstellung
- Beschreibung der benötigten mathematischen Formeln, sowie deren Implementierung
- Die Hauptaspekte meiner Implementierung der Abstraktionen, mit deren Hilfe sich künstliche neuronale Netze erstellen und trainieren lassen (solche Klassen wie z. B Network und Layer, sowie ihre Beziehungen)
- Demonstration der von mir durchgeführten Tests, die sich auf die Problemstellung beziehen, sowie ihre Bedingungen und Resultate

2 Grundbegriffe und allgemeine Theorie

2.1 Künstliche Intelligenz

Künstliche Intelligenz (KI) ist ein Teilgebiet der Informatik, in dem es darum geht, dem Computer intelligentes Verhalten zu verleihen, also solche Komponenten der menschlichen Intelligenz wie Lernen und Denken.[1] Es ist außerdem erwähnenswert, dass das Ziel künstlicher Intelligenz meistens darin besteht, diejenigen Aufgaben zu lösen, welche normalerweise menschliche Intelligenz erfordern. Darauf gehe ich im Kapitel 3. „Verbreitung der KI und Anwendungsgebiete“ genauer ein.

[1] https://de.wikipedia.org/wiki/Künstliche_Intelligenz (Einleitung)

2.2 Biologische neuronale Netze

Ein weiterer wichtiger Begriff ist der eines *biologischen neuronalen Netzes/Netzwerks*. Biologische neuronale Netze sind Kompositionen von Nervenzellen (in den Neurowissenschaften auch als Neuronen bezeichnet), die miteinander vernetzt sind.[2] Durch diese Vernetzung besitzen biologische Neuronen die Fähigkeit, elektrische Signale zu übertragen,[3] aus denen dann Information entsteht.

2.3 Künstliche neuronale Netze

Künstliche neuronale Netze sind eine Methode der künstlichen Intelligenz, mit deren Hilfe intelligentes Verhalten oder Informations- und Datenverarbeitung durch eine bestimmte Komposition von miteinander verbundenen *künstlichen Neuronen* zustande kommt.
Das ganze Modell der neuronalen künstlichen Netze ist durch das entsprechende biologische Modell inspiriert[4], das im menschlichen und tierischen Gehirn enthalten ist.[5]

2.3.1 Künstliche Neuronen

Während es sich bei den biologischen Neuronen um Nervenzellen handelt, die elektrische Signale übertragen und sich unmittelbar im menschlichen bzw. tierischen Körper befinden, sind künstliche Neuronen nichts Anderes als programmierte Objekte einer Abstraktion (einer Klasse, oder Datenstruktur), die sich unbedingt durch folgende Attribute auszeichnen[6]:

- Sie sind mit den anderen Objekten dieser Art vernetzt.
- Sie können Signale in Form von Zahlen übertragen.
- Sie können Signale (Zahlen) verändern, wie eine mathematische Funktion.[7]

Wenn man die drei oben genannten Eigenschaften miteinander kombiniert, kann man künstliche Neuronen als eine mathematische Funktion betrachten, die von *m* Variablen (Eingaben, Inputs) abhängt und eine Ausgabe produziert, die über die Vernetzung weiter propagiert wird.[3]
Es ist außerdem erwähnenswert, dass sich künstliche neuronale Netze von biologischen u. a. dadurch unterscheiden, dass die Ersteren über eine klare Trennung von Eingabeneuronen und

[2] https://de.wikipedia.org/wiki/Neuronales_Netz (Einleitung)
[3] https://de.wikipedia.org/wiki/Neuronales_Netz (Die Vernetzung von Neuronen)
[4] https://en.wikipedia.org/wiki/Artificial_intelligence#Artificial_neural_networks
[5] https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz (Einleitung)
[6] https://en.wikipedia.org/wiki/Artificial_neuron#Biological_models (Einleitung)
[7] https://en.wikipedia.org/wiki/Artificial_neuron (Basic structure)

Ausgabeneuronen verfügen und die in sie eingehenden Signale unmittelbar „nach vorn“ propagieren, während es bei den Letzteren nicht der Fall ist.[8]

2.3.2 Schichten von künstlichen Neuronen

Aufgrund der im Kapitel 2.3.1. beschriebenen klaren Trennung von Eingabe- und Ausgabeneuronen, die künstliche neuronale Netze aufweisen, ist es von praktischer Bedeutung, sie in Schichten aufzuteilen (engl *layers*).[9]
Schichten von künstlichen Neuronen sind also diejenige Abstraktion, die sich zwischen der Abstraktion eines künstlichen neuronalen Netzes und der eines Neurons befindet.
Die Abstraktion von Schichten definiert sich über folgende Eigenschaften[10]:

- Jede Schicht beinhaltet *n* Neuronen, die miteinander *nicht* verbunden sind.
- Diejenige Schicht, die die Signale nur zu der nächsten Schicht befördert, aber sie nicht von einer vorigen Schicht bekommt, heißt die *Eingabeschicht*.
- Diejenige Schicht, die die Signale von einer vorigen Schicht bekommt, aber sie nicht weiter befördert, heißt die *Ausgabeschicht*.
- Diejenigen Schichten, die die Signale sowohl von einer vorigen Schicht bekommen, als auch sie weiter befördern, heißen *verdeckte Schichten* (engl *hidden layers*).

2.3.3 Abstraktion von künstlichen neuronalen Netzen

Aus den in den Kapiteln 2.3.1 und 2.3.2 beschriebenen Abstraktionen lässt sich leicht die Abstraktion eines künstlichen neuronalen Netzes ableiten, besonders wenn man bedenkt, dass die Abstraktion eines künstlichen neuronalen Netzes nichts Anderes ist, als dasjenige, das die Schichten miteinander *vernetzt*. Aufgrund dessen ist ein künstliches neuronales Netz nichts Anderes als eine Liste, die *l* geordnete Schichten beinhaltet.

2.3.4 Signale

Wie im Kapitel 2.3.1. bereits erwähnt wurde, haben künstliche Neuronen die Rolle, die in sie eingehenden Signale in Form von Zahlen zu verändern und zu übertragen. Meistens kommt diese Veränderung durch den Einfluss der sogenannten Gewichtungen und Bias (engl *weights and biases*) zustande. Wenn man ein künstliches Neuron als eine mathematische Funktion

[8] https://de.wikipedia.org/wiki/Neuronales_Netz (Die Vernetzung von Neuronen, siehe: *Divergenz, Konvergenz*)
[9] https://en.wikipedia.org/wiki/Artificial_neural_network (Models - Organization)
[10] https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz (Topologie der Verbindungsnetze)

betrachtet, die von m Eingaben abhängt[11], kann man die Gewichtungen und Bias als Parameter dieser Funktion betrachten.
Die Menge aller Signale in einer Schicht l wird häufig als *Aktivierungen* (engl *activations*) bezeichnet und in Matrizen gespeichert.

Die Matrix aller Aktivierungen einer Schicht l wird in dieser Arbeit als $a^{(l)}$ bezeichnet.
Die Gewichtungen einer Schicht l werden auch häufig in einer Matrix gespeichert, die ich in dieser Arbeit als $w^{(l)}$ kennzeichnen werde. Und die Matrix der Bias einer Schicht l wird im Rahmen dieser Arbeit als $b^{(l)}$ gekennzeichnet.

2.4 Eingabe und Ausgabe eines künstlichen neuronalen Netzes

Das Kapitel 2.3.2. beschreibt künstliche Neuronen vereinfachend, verallgemeinernd und zusammenfassend als mathematische Funktionen. Das Kapitel 2.3.3. beschreibt die Struktur von Schichten, welche geordnete Gruppen von Neuronen bilden. Ein neuronales Netz macht also eine Komposition von Schichten aus, die miteinander verbunden sind. Schlussendlich beschreibt das Kapitel 2.3.4. jene Einheiten, mit denen die Neuronen operieren (Signale oder Zahlen).
Da ein künstliches neuronales Netz aus mathematischen Funktionen komponiert ist, lässt es sich auch auf das Konzept einer recht komplexen mathematischen Funktion reduzieren, diese Funktion wird in dieser Arbeit als $N_{w,\,b}(Eingabe) = Ausgabe$ definiert. Da es sich bei den künstlichen neuronalen Netzen um mathematische Funktionen handelt, erben sie auch den für die mathematischen Funktionen charakteristischen Determinismus, d.h. für jede mögliche Eingabe gibt es jeweils nur eine mögliche Ausgabe.
Mit $w,\ b$ ist die Menge aller Gewichtungen bzw. Bias des gegebenen neuronalen Netzes gemeint. Da die Ausgabe des Netzwerks nicht nur von der Eingabe, sondern auch von diesen Parametern abhängt, kann man diese Parameter als den *Zustand* des Netzes definieren.

2.5 Training

Bei der künstlichen Intelligenz wird davon ausgegangen, dass sie intelligentes Verhalten aufweisen kann, d.h. z. B. handgeschriebene Ziffern erkennen, oder ein Auto steuern, oder bestimmen, welche Werbungen der einen oder anderen Person angezeigt werden sollten. Dafür muss sie in der Lage sein, eine Information zu analysieren und darauf basierend eine

[11] vgl. Kapitel 2.3.1.

eindeutige Entscheidung zu treffen.[12] Wenn man ein künstliches neuronales Netz modelliert, gibt es zwei Möglichkeiten, wie sein Zustand[13] initialisiert werden soll:

1. *w* und *b* werden zufällig initialisiert.
2. *w* und *b* werden einer bestimmten Zahl gleichgesetzt.

Sowohl bei der ersten, als auch bei der zweiten Option handelt es sich im Endeffekt wohl kaum um ein intelligentes Verhalten, eher um ein chaotisches. Eine Möglichkeit, um die es in dieser Arbeit gehen wird, ist, diese Parameter des Netzes anzulernen, es zu trainieren.

Beim *Training* eines künstlichen neuronalen Netzes handelt es sich um die Suche nach denjenigen Parametern *w* und *b*, bei denen das Netz intelligentes Verhalten aufzuweisen beginnt.[14] Der Trainingsprozess wird ausführlich im Kapitel 8. „Theorie des Trainings“ beschrieben.

Es ist aber wichtig zu erwähnen, dass der Trainingsprozess meistens einen riesigen Datensatz erfordert.[15]

2.5.1 Lerndatensatz

Ein Lerndatensatz ist ein Array, das Paare von Matrizen beinhaltet, wobei sich in jedem Paar eine Input-Matrix und eine für dieses Input gewünschte Output-Matrix befinden. Es werden hierbei Matrizen in ihrem mathematischen Sinne verwendet.[16]

Beispiel eines Datensatzes: In dieser Arbeit wird der MNIST-Datensatz verwendet. Dieser besteht aus 70 000 Beispielen von handgeschriebenen Ziffern. Dabei ist jeder Input-Matrix, die Pixel einer Ziffer beschreibt, eine Beschriftung zugeordnet (s. Anhang, Material 2.).[17]

3 Verbreitung der KI und Anwendungsgebiete

Jedes Jahr gewinnt die künstliche Intelligenz an Bedeutung. Immer mehr Unternehmen finden neue Anwendungen für sie. In der Medizin wird künstliche Intelligenz beispielsweise in Form von konvolutionellen neuronalen Netzen (CNNs) erfolgreich eingesetzt, um früh Erkrankungen zu erkennen.[18] Die größten Internetunternehmen verwenden künstliche Intelligenz in Form von neuronalen Netzen, um zu erkennen, welche Art Content ihren

[12] s. Kapitel 2.4, Eingabe-Ausgabe-Determinismus.
[13] s. Kapitel 2.4.
[14] https://en.wikipedia.org/wiki/Artificial_neural_network (Chapter “Training”)
[15] https://en.wikipedia.org/wiki/Artificial_neural_network (Chapter “Training”)
[16] https://de.wikipedia.org/wiki/Überwachtes_Lernen (Kapitel „Definitionen“)
[17] http://yann.lecun.com/exdb/mnist
[18] https://link.springer.com/article/10.1007/s12065-020-00540-3 (Chapter “Medical image understanding”)

Benutzern am besten gefällt, oder welche Werbungen die eine oder die andere Zielgruppe am häufigsten anklickt.[19] Die Anwendungsgebiete der künstlichen Intelligenz sind vor allem[20]:

- Bilderkennung
- Spracherkennung
- Schrifterkennung
- Frühwarnsysteme
- Medizinische Diagnostik

4 Modellierung eines neuronalen Netzes

Laut vielen Quellen sei die Programmiersprache Python die erste Wahl, wenn es um Modellierung und Verwendung von künstlichen neuronalen Netzen geht. Der Grund dafür seien einerseits die zahlreichen mathematischen und statistischen Bibliotheken, andererseits viele Frameworks, mit deren Hilfe sich künstliche neuronale Netze leicht und schnell erstellen und trainieren lassen.[21]

Laut einigen Quellen sei die einfache und vielseitige Syntax von Python einer der wichtigsten Gründe, warum Python so häufig für KI-Projekte verwendet wird. Auch in Forschung, in der künstliche Intelligenz involviert sei, sei Python die erfolgreichste und am weitesten verbreitete Programmiersprache.[22]

Außerdem sind die meisten fertigen Projekte zur Erstellung von neuronalen Netzen lediglich in Python verfügbar, wie bspw. TensorFlow[23] oder scikit-learn[24]. Diese Gegebenheit kann die Benutzer, die auf die Schnelle gute Resultate erzielen möchten, dazu zwingen, sich für Python zu entscheiden.

Schlussendlich sind diejenigen Aufgaben, die in den kompilierten Programmiersprachen schwierig und umständlich gestaltet sind, meistens recht einfach in Python, wo der Entwickler z. B. nie direkt mit dem Memory-Management oder langer und aufwendiger Kompilierung konfrontiert ist, unter anderem deshalb, weil Python eine höhere und interpretierte Programmiersprache ist.[25]

[19] https://www.facebook.com/business/news/good-questions-real-answers-how-does-facebook-use-machine-learning-to-deliver-ads
[20] https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185
[21] https://www.computerwoche.de/a/die-besten-coding-sprachen-fuer-ki
[22] https://www.westhouse-group.com/ki-programmiersprachen-python-c-java-lisp-prolog
[23] https://www.tensorflow.org/install
[24] https://scikit-learn.org/stable/install.html
[25] https://de.wikipedia.org/wiki/Python_(Programmiersprache)

4.1 Ein mögliches Problem

Wenn man bedenkt, dass Python eine interpretierte Programmiersprache ist, deren Skripte von jener Sprache interpretiert werden müssen, in der der Python-Interpreter geschrieben ist (Die Programmiersprache C)[26], kann man zum Schluss kommen, dass Python durch diese zusätzliche Interpretationsstufe eigentlich relativ leistungsschwach sein sollte, wenn man Python mit den kompilierten Programmiersprachen vergleicht, deren Code direkt in den Assembly-Code konvertiert wird, der anschließend vom Prozessor unmittelbar ausgeführt wird, wie es z. B. bei C oder C++ der Fall ist.[27]

Hier ist ein Diagramm abgebildet, das diese Hypothese untermauert:

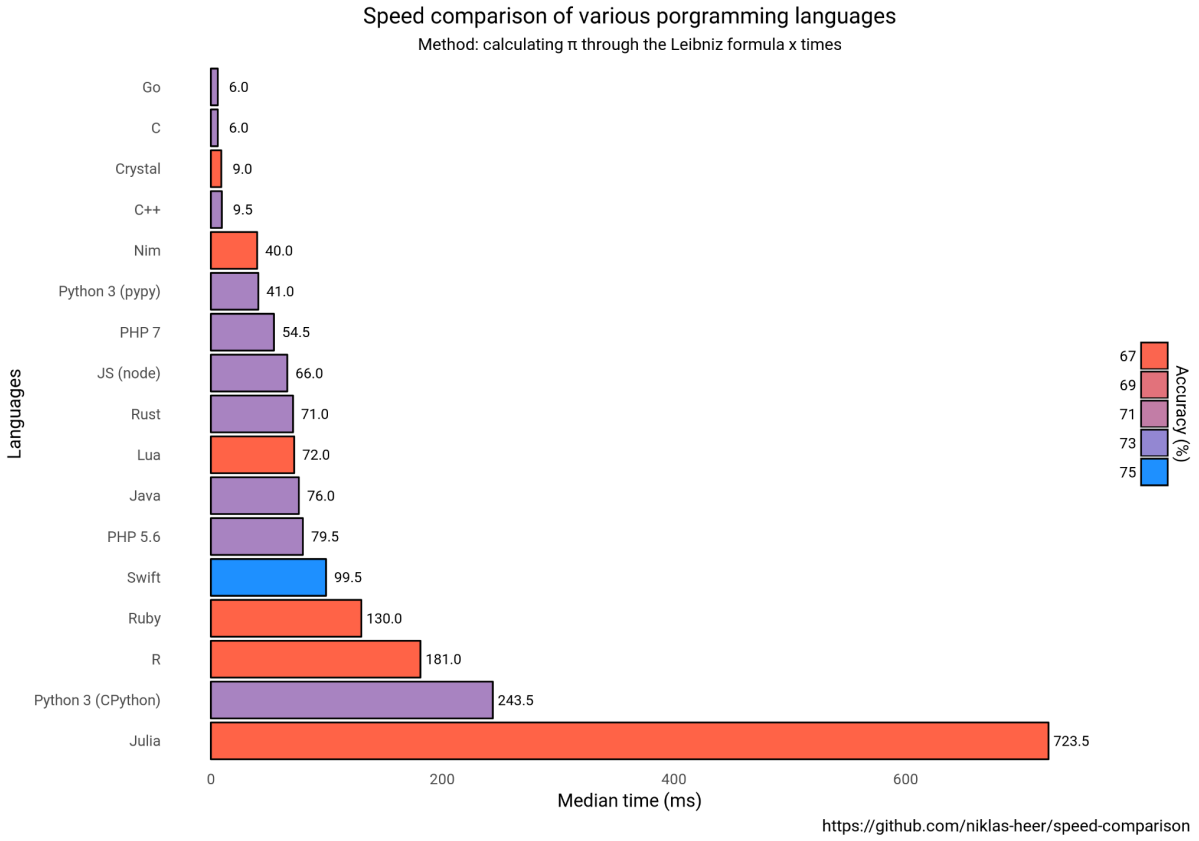


Abbildung 1. Quelle: https://github.com/niklas-heer/speed-comparison

Das Diagramm stellt dar, wie präzise und schnell verschiedene Programmiersprachen dieselbe rechenintensive Aufgabe erledigen, nämlich die Konstante π mit Hilfe der Leibniz-Reihe[28] zu berechnen.

Wie man auf dem Diagramm sehen kann, liegt Python weit unten mit einem durchschnittlichen Resultat von *243.5 ms*, wohingegen die kompilierten

[26] https://en.wikipedia.org/wiki/Python_(programming_language) (Chapter "Implementations")
[27] https://de.wikipedia.org/wiki/C_(Programmiersprache), https://de.wikipedia.org/wiki/C%2B%2B
[28] https://de.wikipedia.org/wiki/Leibniz-Reihe

Programmiersprachen wie Go, C oder C++ weit oben platziert sind und lediglich *6.0 ms*, *6.0 ms*, bzw *9.5 ms* gebraucht haben.

Diese Gegebenheit stellt ein Problem dar: wie es im Kapitel 4. „Modellierung eines neuronalen Netzes“ beschrieben ist, sind viele neuronale Netze in Python implementiert und modelliert und die Wahl der Programmiersprache mag sich sehr negativ auf die Leistung[29] des neuronalen Netzes auswirken.

4.2 Eine mögliche Lösung

In dieser Arbeit wird versucht, ein künstliches neuronales Netz von Grund auf mit Hilfe der Programmiersprache C++ zu erstellen. Die Leistung dieses Netzes wird dann mit der Leistung des von Michael Nielsen[30] erstellten Projekts[31] verglichen, das in Python geschrieben ist.

Dieses Projekt habe ich aus zwei Gründen für den Vergleich gewählt:

1. Es ist möglichst einfach implementiert, somit ist es gut für Bildungszwecke geeignet.
2. Es ist Open-Source und man kann leicht auf den Quellcode zugreifen.

Hypothese H_0: Es wird davon ausgegangen, dass ein in C++ implementiertes neuronales Netz viel bessere Leistung aufweisen müsse.

5 Design des Projekts

5.1 Externes Design und Planung des Programmierprojekts

Der Code dieses Projekts wird auf *GitHub* unter dem Link https://github.com/eugen-bondarev/BLL gespeichert. Das sich im Anhang befindende *Material 1*. gibt einige Hinweise über die Gestaltung des Quellcodes, falls sich der Leser dafür interessiert.

Das Betriebssystem, für das das Programm kompiliert und getestet wird, ist *Windows 10*, der Code sollte sich aber auch problemlos für andere Plattformen kompilieren lassen. Als Compiler wird *Visual Studio Community 2019* verwendet und für das Build-Management – *CMake*.

---

[29] Mit „Leistung“ meine ich ausschließlich die Zeit, die das neuronale Netz braucht, bis es eine gegebene Aufgabe beherrscht. Da es sich bei neuronalen Netzen um reine mathematische Einheiten handelt (Funktionen), sollte es keinen direkten Zusammenhang zwischen der gewählten Programmiersprache und der Genauigkeit der erzielten Ergebnisse geben.

[30] https://michaelnielsen.org

[31] https://github.com/mnielsen/neural-networks-and-deep-learning

5.2 Internes Design und eingebundene Bibliotheken

Für dieses Projekt werden außerdem *OpenGL* (in Form von solchen Bibliotheken wie *GLEW* und *GLFW*), *ImGui* (eine Bibliothek, um Benutzerschnittstellen (GUI) im OpenGL-Kontext zu rendern), *Simple C++ reader for MNIST dataset* und *eigen* (für die Mathematik) verwendet.

6 Modell eines neuronalen Netzes

Untersuchen wir genauer das Modell eines neuronalen Netzes. Wie bereits erwähnt, besteht ein neuronales Netz von *Neuronen*, die geordnete Schichten *(*engl *Layers)* bilden.[32] Die Anzahl von Neuronen in jeder einzelnen Schicht definiert das Modell des Netzwerks. Zum Beispiel verfügt das in der Abbildung 1. dargestellte Netzwerk über zwei Neuronen in der Eingabeschicht, vier Neuronen in der ersten und in dem Fall einzigen verdeckten Schicht und ein Neuron in der Ausgabeschicht.

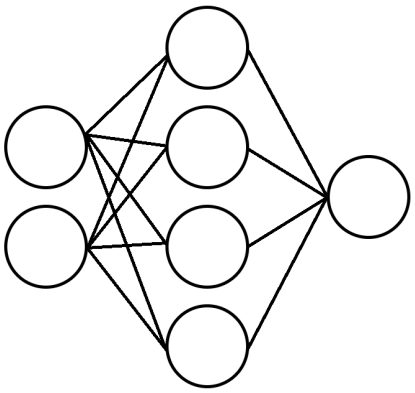


Abbildung 1.

Wie man in der Abbildung sehen kann, sind alle Neuronen in der Schicht *l* mit allen Neuronen in der Schicht *l-1* verbunden. So hat zum Beispiel jedes Neuron der mittleren Schicht Kontakt zu jedem Neuron der Eingabeschicht und so weiter. Die Anzahl der Neuronen in jeder Schicht sowie die Anzahl der Schichten kann selbstverständlich variieren, was einen großen Raum für Experimente gibt. Die Ausgabe des Netzes ist nichts Anderes als die Matrix von Werten der Ausgabeschicht.

Der Wert jedes einzelnen Neurons kann mit Hilfe der folgenden Funktion beschrieben werden:

[32] https://youtu.be/aircAruvnKk?t=215 „But what is a neural network?“, ab 3:35

$a^{(l)} = g^{(l)}(z^{(l)})$, wobei l der Index der Schicht, $z^{(l)}$ – Gewichtete Summe[33] oder Multiplikation von den Gewichtungen mit allen Werten der vorigen Schicht und $g^{(l)}$ die Aktivierungsfunktion (normalerweise eine nichtlineare Funktion) sind. Die Ausgabe des Netzes ist also nichts Anderes als $a^{(L)}$, wobei L der Index der letzten Schicht ist. Das heißt, um die Ausgabe des Netzes zu berechnen, muss man den Wert von $z^{(l)}$ finden, den ich oben als „gewichtete Summe“ bezeichnet habe. Tatsächlich ist es lediglich die Summe aller gewichteten Inputs für das gegebene Neuron plus dessen Bias. Dies lässt sich folgendermaßen veranschaulichen:

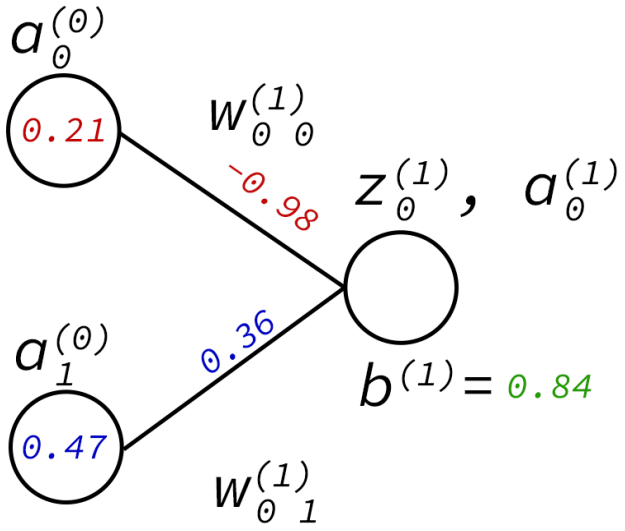


Abbildung 2.

In diesem Beispiel ist

$a_0^{(1)} = g^{(1)}(z_0^{(1)}), \quad z_0^{(1)} = 0.21 \cdot (-0.98) + 0.47 \cdot 0.36 + 0.84$

$z_0^{(1)} = 0.8034, \quad a_0^{(1)} = g^{(1)}(0.8034) = 0.6907,$

wobei $g^{(l)}(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

Wie man sehen kann, enthält jede Schicht die Werte $w^{(l)}$, $b^{(l)}$, $z^{(l)}$, $a^{(l)}$. Wenn man diese jedoch als Matrizen betrachtet, lässt sich die Berechnung der gewichteten Summe und anschließend der Aktivierung folgendermaßen verallgemeinern:

(1) $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$

[33] https://www.golem.de/news/deep-learning-maschinen-die-wie-menschen-lernen-1510-116468-3.html (Abbildung 2.)

(2) $a^{(l)} = \sigma(z^{(l)})$

Die Abbildung 3. zeigt, wie die Berechnung der Aktivierungen aller Neuronen in einer Schicht *l* aussehen würde (angenommen, diese besteht aus *m* Neuronen, während die Schicht *l-1* aus *n* Neuronen besteht):

$$z^{(l)} = \begin{bmatrix} w_{1\,1}^{(l)} & w_{1\,2}^{(l)} & \dots & w_{1\,n}^{(l)} \\ w_{2\,1}^{(l)} & w_{2\,2}^{(l)} & \dots & w_{2\,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m\,1}^{(l)} & w_{m\,2}^{(l)} & \dots & w_{m\,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_n^{(l)} \end{bmatrix}$$

Abbildung 3.

Anschließend können wir das Output des Netzwerks berechnen, indem wir für $a^{(0)}$ gewünschtes Input einsetzen und die oben abgebildeten Formeln auf jede einzelne Schicht anwenden, angefangen mit der Schicht 2. An dieser Stelle möchte ich noch einmal betonen, dass das infolge dieses Vorgangs ausgegebene Output chaotisch aussehen wird, weil das Netzwerk zu diesem Zeitpunkt noch nicht trainiert worden ist.[34]

7 Abstraktion eines neuronalen Netzes

In diesem Kapitel wird erläutert, wie sich die früher beschriebene Funktionsweise eines neuronalen Netzes in ein Programm übertragen lässt. Dabei wird der Code im *objektorientierten Paradigma* geschrieben, weil dieses sich ausgezeichnet für Hierarchien eignet. Fangen wir in diesem Sinne noch einmal mit dem Begriff eines neuronalen Netzes, also von Oben der Hierarchie von Klassen an. Diesmal berücksichtigen wir jedoch, welche Attribute und Methoden die Klasse *Network* beinhalten würde, also über welche Daten deren Objekt verfügt und welche Aktionen es ausführen muss. Fangen wir mit dem Konstruktor oder mit der Erstellung des Objekts eines Netzwerks an. Wie es im Kapitel 6. „Modell eines neuronalen Netzes“ beschrieben wurde, definiert sich ein neuronales Netz über eine Komposition von Schichten. Um eine Schicht zu beschreiben, brauchen wir nur die Anzahl von Neuronen in dieser Schicht und die vom Benutzer gewählte Funktion $g^{(l)}(x)$. Die Struktur, die die Schichten beschreibt, sieht dann folgendermaßen aus:

```
 9 // Exzerpt 1, Commit: 31797e3, Datei: src/AI/LayerDescriptor.h
10 struct LayerDescriptor
11 {
12     size_t numNeurons;
```

[34] s. Kapitel 2.5 „Training“

```
13     // Mathematische funktion f(x).
14     Activation function;
15 };
```

Dann ist der das ganze Netzwerk beschreibende Typ nichts Anderes als:

```
10 // Exzerpt 2, Commit: 31797e3, Datei: src/AI/NetworkDescriptor.h
11 using NetworkDescriptor = Vec<LayerDescriptor>;
```

Diesen Typ verwenden wir dann im Konstruktor der Klasse *Network*:

```
10 // Exzerpt 3, Commit: 31797e3, Datei: src/AI/Network.h
11 class Network
12 {
13 public:
14     Network(const NetworkDescriptor& descriptor);
15 };
```

Mit dessen Hilfe kann man dann eine Instanz dieser Klasse, also ein Netzwerk erstellen:

```
 4 // Exzerpt 4, Commit: 7b5e09e, Datei: src/Main.cpp
 5 AI::Network network({
 6     { 784 }, // Eingabe der Pixel, (28 · 28 = 784 Neuronen).
 7     { 16 },  // Die erste verdeckte Schicht (16 Neuronen).
 8     { 16 },  // Die zweite verdeckte Schicht (16 Neuronen).
 9     { 10 }   // Die Ausgabeschicht (10 Neuronen).
10 });
```

Dem Konstruktor der Klasse *Network* übergeben wir die Struktur *[784, 16, 16, 10]*, jedoch verwenden wir diese Daten im Konstruktor noch nicht. Um dies zu tun, muss zuerst die Klasse *Layer* beschrieben werden. Eine Struktur, die die Schichten beschreibt, ist bereits vorhanden (*LayerDescriptor*). Diese muss der Konstruktor der Klasse *Layer* als Parameter annehmen und sich um die Erstellung solcher Matrizen wie $a^{(l)}$, $z^{(l)}$, $w^{(l)}$, $b^{(l)}$ kümmern. Die Daten, die ich in der Klasse *Layer* in Form von Attributen deklariere, sind wie folgt:

```
19 // Exzerpt 5, Commit: e12d74e, Datei: src/AI/Layer.h
20 Activation g;       // Aktivierungsfunktion
21 Matrix a;           // Aktivierungen (Werte von den Neuronen)
22 Matrix z;           // Gewichtete Summe
23 Matrix w;           // Gewichtungen
24 Matrix b;           // Bias
```

Noch ein wichtiges Attribut, das für die Berechnung von $z^{(l)}$ relevant ist, ist $a^{(l-1)}$. Dieses lässt sich in folgender Form darstellen:

```
// Exzerpt 6, Commit: e12d74e, Datei: src/AI/Layer.h
Matrix* previousLayerActivation{ nullptr };
```

Für die erste Schicht sind die Attribute $a^{(l-1)}$ und $w^{(l)}$ offensichtlich irrelevant, sie bleiben *nullptr*, bzw. leer. Da eines der Attribute der Klasse *Layer* ein Zeiger ist, lässt sich diese Klasse und somit die Klasse *Network* schlecht kopieren, weswegen ich den Kopierkonstruktor der Klasse *Network* überlade. Die Implementierung der Symbole, in denen all die oben deklarierten Attribute initialisiert werden, sieht folgendermaßen aus:

```
// Exzerpt 7, Commit: e12d74e, Datei: src/AI/Layer.cpp
Layer::Layer(const LayerDescriptor& descriptor)
{
    g = descriptor.function;
    a = Matrix(descriptor.numNeurons, 1);
    z = Matrix(descriptor.numNeurons, 1);
    b = Matrix(descriptor.numNeurons, 1);
}

void Layer::ConnectWithPreviousLayer(Matrix* previousLayerActivation)
{
    this->previousLayerActivation = previousLayerActivation;
}

void Layer::InitWeights()
{
    const Matrix& prevA = *previousLayerActivation;
    w = Matrix(a.GetRows(), prevA.GetRows(), GenRandomWeight);
}
```

Diese Funktionen werden dann im Konstruktor der Klasse *Network* aufgerufen:

```
// Exzerpt 8, Commit: e12d74e, Datei: src/AI/Network.cpp
Network::Network(const NetworkDescriptor& descriptor)
{
    layers.reserve(descriptor.size());
    // Die erste Schicht, es gibt weder
    // Gewichtungen noch eine Verbindung zur vorigen Schicht.
    layers.emplace_back(descriptor[0]);
    // Alle weiteren Schichten enthalten
    // Gewichtungen und einen Zeiger auf die vorigen Aktivierungswerte.
    for (size_t i = 1; i < descriptor.size(); ++i)
    {
```

```
15        Layer& layer = layers.emplace_back(descriptor[i]);
16        layer.ConnectWithPreviousLayer(&layers[i - 1].a);
17        layer.InitWeights();
18    }
19 }
```

Zusammenfassend ist die Aufgabe des Konstruktors der Klasse *Network*, eine Liste mit Objekten der Klasse *Layer* zu füllen. Die Klasse *Layer* initialisiert in ihrem Konstruktor die für die weiteren Berechnungen benötigten Matrizen.

Jetzt verfügt die Klasse *Network* über alle notwendigen Daten, um eine Prognose machen zu können. Die Methode, die die Aktivierungen[35] einer beliebigen Schicht berechnet, lässt sich jetzt in einer sehr einfachen und kurzen Memberfunktion der Klasse *Layer* darstellen:

```
28 // Exzerpt 9, Commit: ea7f072, Datei: src/AI/Layer.cpp
29 void Layer::Evaluate()
30 {
31    z = w * (*previousLayerActivation) + b;
32    a = z.Apply(g.function);
33 }
```

Wie man sehen kann, sind das genau die zwei einfachen Formeln (*1*, *2*), die ich im Kapitel 6. „Modell eines neuronalen Netzes“ hergeleitet habe.

Da die Ausgabe des Netzes nichts Anderes ist, als lediglich der Wert aller Aktivierungen *a* der letzten Schicht, ist die für die Prognosen verantwortliche Funktion folgendermaßen definiert:

```
35 // Exzerpt 10, Commit: ea7f072, Datei: src/AI/Network.cpp
36 Matrix Network::Feedforward(const Matrix& input)
37 {
38    GetFirstLayer().a = input;
39    for (size_t i = 1; i < layers.size(); ++i)
40    {
41       layers[i].Evaluate();
42    }
43    return GetLastLayer().a;
44 }
```

[35] s. Kapitel 2.3.4. „Signale“

8 Theorie des Trainings

Die Implementierung ist nun bereit zum Training. Das Training eines neuronalen Netzwerks bedeutet nichts Anderes als Verringerung des Fehlers des Netzwerks[36], welcher durch eine Verlustfunktion (engl *cost function*, *loss function*)[37] beschrieben werden kann.

Der Fehler des Netzwerks für ein Sample *n* lässt sich mathematisch zum Beispiel folgendermaßen darstellen: $C_n(a^{(L)}, y_n) = (a^{(L)} - y_n)^2$, wobei $a^{(L)}$ die Ausgabe des Netzwerks und y_n die gewünschte Ausgabe für ein Sample *n* und C_n die Verlustfunktion sind.

Unser Ziel ist es, den Fehler des Netzes oder die Ausgabe dieser Verlustfunktion zu verringern. Dabei haben wir keinen Einfluss auf den Wert von y_n, weil es genau derjenige Wert ist, den wir erreichen möchten, d.h. es muss der Wert von $a^{(L)}$ geändert werden, welcher wiederum eine Funktion ist: $a^{(L)}(w^{(L)}, b^{(L)}, a^{(L-1)})$. Wenn wir die mathematischen Formeln aus dem Kapitel 6. „Modell eines neuronalen Netzes“ verwenden, kann der Fehler der letzten Schicht wie folgt umgeformt werden:

$$C_n(w^{(L)}, a^{(L-1)}, b^{(L)}, y_n) = (\sigma(w^{(L)}a^{(L-1)} + b^{(L)}) - y_n)^2$$

Beim Training handelt es sich um die Suche nach dem lokalen Minimum dieser Funktion (weil es nahezu unmöglich ist, das globale Minimum zu finden)[38]. Je kleiner $C_n(a^{(L)}, y_n)$ im Durchschnitt ist, desto besser ist das Netzwerk trainiert.

Wie man sehen kann, kann man die Funktion $C_n(w^{(L)}, a^{(L-1)}, b^{(L)}, y_n)$ partiell nach $w^{(L)}$, $a^{(L-1)}$ und $b^{(L)}$ ableiten. Infolge dieser partiellen Ableitung bekommen wir die Werte $\frac{\partial C_n}{\partial w^{(L)}}$, $\frac{\partial C_n}{\partial b^{(L)}}$, und $\frac{\partial C_n}{\partial a^{(L-1)}}$. Diese sind die sogenannten Gradientenvektoren, die in die Richtung des *steilsten Anstiegs* der Funktion $C_n(a^{(L)}, y_n)$ zeigen.[39]

[36] M. Nielsen, “Neural Networks and Deep Learning”, Chapter “Learning with gradient descent”
[37] https://en.wikipedia.org/wiki/Loss_function
[38] https://youtu.be/IHZwWFHWa-w?t=181 “Gradient descent, how neural networks learn”, Chapter “Cost functions”, ab 5:30
[39] https://youtu.be/IHZwWFHWa-w?t=505 “Gradient descent, how neural networks learn”, Chapter “Gradient descent”, ab 8:10

Wenn wir also genau diese Gradientenvektoren multipliziert mit einer Konstante *r* von den aktuellen Parametern $w^{(L)}$ und $b^{(L)}$ subtrahieren, dann hieße das, dass die Ausgabe der Funktion C_n um *r* Einheiten in die Richtung des *steilsten Abstiegs* gegangen ist.[40]

Somit verringert sich der Fehler, was wiederum bedeutet, dass das Netzwerk nächstes Mal bei einer *ähnlichen Eingabe* eine *ähnliche Ausgabe* erzielen wird.

8.1 Backpropagation

Nach der Anpassung der letzten Schicht offenbart sich allerdings ein Problem: Der Wert $a^{(L)}$ hängt nicht nur von $w^{(L)}$ und $b^{(L)}$ ab, welche wir vor Kurzem erfolgreich reduziert haben, sondern auch von $a^{(L-1)}$, was man den im Kapitel 6. hergeleiteten Formeln 1 und 2 entnehmen kann. Auf den Wert $a^{(L-1)}$ haben wir keinen direkten Einfluss, weil er schlussendlich auch nur von den Werten $w^{(L-1)}$, $b^{(L-1)}$ und $a^{(L-2)}$ abhängt. Äquivalente Abhängigkeiten weisen natürlich auch alle vorigen Schichten auf, bis die allererste Schicht erreicht ist. Eine natürliche und gerechtfertigte Frage an dieser Stelle wäre: „Während wir den Fehler der letzten Schicht dadurch definiert haben, dass wir die eigentliche Ausgabe unseres Netzes mit der gewünschten verglichen haben, wie lässt sich der Fehler der vorigen Schichten definieren, also womit können wir all die weiteren Aktivierungsmatrizen vergleichen?“. Die Antwort ist *Backpropagation*.

Außer den Ableitungen der Gewichtungen und Bias habe ich auch folgende Ableitung erwähnt: $\frac{\partial C_n}{\partial a^{(L-1)}}$. Diese gibt die Richtung des steilsten Anstiegs der Funktion C_n, wenn nur Änderungen von $a^{(L-1)}$ berücksichtigt werden. Das heißt, wenn man diesen Wert von dem Wert $a^{(L-1)}$ subtrahiert, kann man diese Differenz $(a^{(L-1)} - \frac{\partial C_n}{\partial a^{(L-1)}})$ als jenen Fehler interpretieren, den wir für die weiteren Anpassungen benötigen.

Jetzt, wo wir auch den Fehler der vorigen Schicht definiert haben, kann auch diese angepasst werden, sodass sich ihr Beitrag zum Fehler $C_n$verringert. Die Tatsache, dass wir den Einfluss von $a^{(L-1)}$ auf $a^{(L)}$ berechnet und den entsprechenden Fehler des Neurons $a^{(L-1)}$ kalkuliert haben, heißt nichts Anderes als Backpropagation. Wir kalkulieren den Fehler einer Schicht *l*

[40] Als „Konstante r“ ist hier die so g. Lernrate bezeichnet (engl *learning rate*), s.: https://en.wikipedia.org/wiki/Learning_rate

und propagieren ihn weiter zur Schicht *l-1*. Dieser Vorgang wird wiederholt, bis die erste Schicht erreicht ist.

8.2 Partielle Ableitungen

Nun müssen diejenigen Ableitungen, die im vorigen Kapitel vorgeführt worden sind ($\frac{\partial C_n}{\partial w^{(L)}}$, $\frac{\partial C_n}{\partial b^{(L)}}$, $\frac{\partial C_n}{\partial a^{(L-1)}}$), berechnet werden. Dafür muss die Verlustfunktion C_n jeweils nach $w^{(L)}$, $b^{(L)}$ und $a^{(L-1)}$ abgeleitet werden.

Zur Erinnerung: Die Formel der Verlustfunktion lautet:

$$C_n(a^{(L)}, y_n) = (\sigma(w^{(L)}a^{(L-1)} + b^{(L)}) - y_n)^2$$

So können die Ableitungen mit Hilfe der Kettenregel dargestellt werden[41]:

$$(1) \quad \frac{\partial C_n}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_n}{\partial a^{(L)}}$$

$$(2) \quad \frac{\partial C_n}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_n}{\partial a^{(L)}}$$

$$(3) \quad \frac{\partial C_n}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_n}{\partial a^{(L)}}$$

Wenn man sich die oben abgebildeten Formeln anschaut, fällt sofort auf, dass sich die letzten zwei Terme nicht im Geringsten unterscheiden. Von dieser vereinfachenden Eigenschaft wird im Kapitel 9.4. Gebrauch gemacht, wenn das Backpropagation-Verfahren implementiert wird, was sich schlussendlich positiv auf die Leistung auswirkt, da das Programm dann diesen Term nur einmal kalkulieren muss, anstatt dreimal.

Nun werden die Ableitungen berechnet:

$$(I) \quad \frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$(II) \quad \frac{\partial C_n}{\partial a^{(L)}} = 2(a^{(L)} - y_n)$$

$$(III) \quad \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_n}{\partial a^{(L)}} = \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y_n)$$

$$(IV) \quad \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$(4) \quad \frac{\partial C_n}{\partial w^{(L)}} = a^{(L-1)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y_n)$$

$$(V) \quad \frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$$

$$(5) \quad \frac{\partial C_n}{\partial b^{(L)}} = \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y_n)$$

[41] https://youtu.be/tIeHLnjs5U8?t=230 “Backpropagation calculus”, Chapter “Computing relevant derivatives”, ab 3:50

$$(VI)\ \frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)} \qquad (6)\ \ \frac{\partial C_n}{\partial a^{(L-1)}} = w^{(L)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y_n)$$

8.3 Engpass des Algorithmus

In dem neuronalen Netz, das in dieser Arbeit implementiert und modelliert wird, hat die erste Schicht 784 und die zweite Schicht 16 Neuronen. Daher besteht die erste Matrix der Gewichtungen aus 12544 reellen Zahlen. Wenn wir diese riesige Matrix mit der Matrix $a^{(1)}$ multiplizieren, die 16 Zeilen und eine Spalte beinhaltet, kommen wir zum Schluss, dass diejenige For-Schleife, in der die Matrizen kalkuliert werden, 12544 Iterationen machen muss. Und diese kleine Operation ist nur ein winziger Bruchteil des riesigen Algorithmus. Es müssen dann die weiteren Schichten berechnet werden, es müssen dann all die Ableitungen kalkuliert und von den aktuellen Parametern abgezogen werden, um den Fehler des Netzes zu verringern. All diese Schritte machen eine enorme Anzahl mathematischer Operationen aus. Daher gehört das Training von neuronalen Netzen zu äußerst *rechenintensiven Algorithmen*.

8.4 Stochastisches Gradientenabstiegsverfahren

Eine mögliche Verbesserung des Algorithmus des Trainings ist das sogenannte *stochastische Gradientenabstiegsverfahren*.[42] Die Idee dieses Algorithmus besteht darin, dass die Anpassungen der Gewichtungen und Bias nicht nach der Bearbeitung jedes einzelnen Samples stattfinden, sondern erst nach *m* Samples (*m* ist eine vom Benutzer gewählte natürliche Zahl). Die Matrizen, die diese Anpassungen für die Gewichtungen und Bias beinhalten, werden dann durch *m* geteilt, um den Durchschnitt der Anpassungen zu finden, weswegen dieses Gradientenabstiegsverfahren stochastisch ist. Der Parameter *m* ist eine natürliche Zahl, die die Größe der Mini-Batches bezeichnet, nach deren Bearbeitung die Parameter angepasst werden.

9 Implementierung des Trainings

9.1 Vorbereitung

Zunächst einmal muss der Trainingsdatensatz heruntergeladen und im Programm mit Hilfe einer zusätzlichen Bibliothek gelesen und angezeigt werden. Nachdem ich einige neue

[42] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Klassen und Funktionen hinzufügen musste, die sich um das Laden und Anzeigen der Trainingsdaten kümmern, teste ich diese hiermit:

```
16 // Exzerpt 11, Commit: 12dd521, Datei: src/Main.cpp
17 const AI::TrainingData testData{ MNIST::Load(
18    "dataset/test-images",
19    "dataset/test-labels"
20 ) };
21 Window window{ 350, 350 };
22 while (window.IsRunning())
23 {
24    window.BeginFrame();
      ...
31    ImGui::Begin("%i. Sample, label: %i", randomSample, label);
32       ImGui::RenderMatrix(testData[randomSample].input);
33    ImGui::End();
34    window.EndFrame();
35 }
```

Der in diesem Exzerpt vorgeführte Code produziert folgende Ausgabe:

Abbildung 4.

Eine solche Ausgabe zeugt davon, dass das Programm in der Lage ist, einen gewünschten Lerndatensatz zu lesen, zu laden, und ein zufälliges Element anzuzeigen. Nun kann die für das Training verantwortliche Funktion implementiert werden.

9.2 Die SGD Funktion

Für das Training des Netzwerks ist die Memberfunktion *SGD* verantwortlich, deren Deklaration folgendermaßen aussieht:

```
18 // Exzerpt 12, Commit: 12dd521, Datei: src/AI/Network.h
19 void SGD(
20    const TrainingData& trainingData,
21    const size_t numEpochs,
22    const size_t miniBatchSize,
23    const Num eta
24 );
```

Der erste Parameter ist selbsterklärend, jedoch ist die Bedeutung der Anderen nicht so offensichtlich. Sie haben folgende Rollen:

- *numEpochs* bestimmt, wie viele Durchgänge das Netzwerk trainiert werden muss.
- *miniBatchSize* ist jener *m*-Wert, der im Kapitel 8.4. definiert wurde.
- *eta* ist ein für das Gradientenabstiegsverfahren notwendiger Parameter, die Lernrate, die im Kapitel 8. definiert wurde.

Von all den oben erwähnten Parametern müssen die letzten drei empirisch bestimmt werden. Das heißt, die „perfekten“ Werte für diese hängen von der Aufgabe, dem Datensatz und sogar voneinander ab. Diese Parameter werden häufig als *Hyperparameter* bezeichnet.[43]

Der Funktionsaufruf in *Main.cpp* kann bspw. folgendermaßen aussehen:

```
57 // Exzerpt 13, Commit: 12dd521, Datei: src/Main.cpp
58 network.SGD(trainingData, 3, 10, 1.5f);
```

Und implementiert wird die Funktion *SGD* wie folgt:

```
104 // Exzerpt 14, Commit: 12dd521, Datei: src/AI/Network.cpp
105 void Network::SGD(
106    const TrainingData& trainingData,
107    const size_t numEpochs,
108    const size_t miniBatchSize,
109    const Num eta)
110 {
111    for (size_t epoch = 0; epoch < numEpochs; ++epoch)
112    {
113       const TrainingData shuffled{ Util::Shuffle(trainingData) };
114       NetworkAdjustments adjustments{};
```

[43] https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)

```
115         CreateAdjustmentsShape(adjustments);
116         for (
117             size_t sample = 0;
118             sample < shuffled.size();
119             sample += miniBatchSize)
120         {
121             const TrainingData miniBatch{
122                 CreateMiniBatch(shuffled, miniBatchSize, sample)
123             };
124             Backpropagation(miniBatch, adjustments);
125             ApplyAdjustments(adjustments, miniBatchSize, eta);
126         }
127     }
128 }
```

Wie man sehen kann, gibt es zwei For-Schleifen. In der Äußeren geht das Programm jede einzelne Epoche[44] durch. Für jede Epoche mischt es den Trainingsdatensatz, erstellt die Matrizen, in denen die Anpassungen kalkuliert werden (*CreateAdjustmentsShape*) und startet für jedes Mini-Batch die innere For-Schleife, in der zwei entscheidende Funktionen aufgerufen werden, nämlich *Backpropagation* und *ApplyAdjustments*.

9.3 Backpropagation

Zunächst möchte ich auf die Memberfunktion *Backpropagation* eingehen, die wie folgt definiert ist:

```
76 // Exzerpt 15, Commit: d637c72, Datei: src/AI/Network.cpp
77 void Network::Backpropagation(
78   const TrainingData& miniBatch,
79   NetworkAdjustments& adjustments)
80 {
81    for (const TrainingSample& sample : miniBatch)
82    {
83       Feedforward(sample.input);
84       Matrix y{ (sample.output - layers[layers.size() - 1].a).pow(2) };
85 
86       for (size_t l = layers.size(); l--> 1;)
87       {
88          const Matrix errorPropagation{
89             layers[l].PropagateError(y, adjustments[l]);
```

[44] https://www.baeldung.com/cs/epoch-neural-networks Chapter “3. Epoch in Neural Networks”

```
90          };
91          y = layers[l - 1].a - errorPropagation;
92      }
93  }
94 }
```

Diese Funktion nimmt als Parameter ein Mini-Batch und die Referenz der Anpassungen an. Jedes Sample des Mini-Batch-Arrays geht als Input in das Netzwerk ein, die Ausgabe wird kalkuliert. Anschließend wird die Prognose des Netzwerks mit der im Sample enthaltenen richtigen Lösung verglichen, der Fehler wird berechnet. Dann, in einer For-Schleife, für jede Schicht (angefangen mit der letzten) wird dieser Fehler propagiert. Basierend auf diesem Fehler werden für jede einzelne Schicht die Ableitungen der Verlustfunktion nach Gewichtungen, Bias und Aktivierungen bestimmt. Dann werden diese Ableitungen (außer der Ableitung nach Aktivierungen) von den Anpassungen subtrahiert. Die Ableitung der Aktivierungen wird dann von den aktuellen Aktivierungswerten subtrahiert. Diese Differenz wird dann weiter als Fehler propagiert, wie es im Kapitel 8.1. beschrieben ist.

9.4 PropagateError

Die Definition der womöglich allerwichtigsten Funktion des Programms – *Layer::PropagateError* – sieht folgendermaßen aus:

```
35 // Exzerpt 16, Commit: d637c72, Datei: src/AI/Layer.cpp
36 Matrix Layer::PropagateError(
37     const Matrix& y,
38     LayerAdjustments& adjustments)
39 {
40     const Matrix delCdelA{ (a - y) * 2.0f };
41     const Matrix delAdelZ{ z.Apply(g.derivative) };
42     const Matrix delCdelZ{ delCdelA.EntrywiseProduct(delAdelZ) };
43 
44     const Matrix& biasGradient{ delCdelZ };
45     const Matrix weightGradient{
46         delCdelZ *
47         previousLayerActivation->Transpose()
48     };
49     const Matrix activationGradient{ w.Transpose() * delCdelZ };
50 
51     adjustments.b -= biasGradient;
52     adjustments.w -= weightGradient;
```

```
53     return activationGradient;
54 }
```

In der Zeile 40 wird $C_n(a^{(L)}, y_n) = (a^{(L)} - y_n)^2$ abgeleitet. Diese Ableitung in Form von $\frac{\partial C_n}{\partial a^{(L)}}$ ist der letzte Term aller drei Ableitungen, die in dieser Memberfunktion berechnet werden müssen.[45]

In der darauf folgenden Zeile wird $a^{(L)} = \sigma(z^{(L)})$ abgeleitet. $a^{(L)}$' ist nichts Anderes als $\sigma'(z^{(L)})$. Und σ' ist im Programm definiert, wie folgt[46]:

```
11 // Exzerpt 17, Commit: 13ee2d9, Datei: src/AI/Activation.cpp
12 Num SigmoidDerivative(const Num x)
13 {
14     return Sigmoid(x) * (1.0f - Sigmoid(x));
15 }
```

Zwecks Optimierung werden die zwei Terme $\frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_n}{\partial a^{(L)}}$ zu einem Term zusammengefasst und in der Zeile 42 berechnet. Da diese zwei Terme am Ende jeder Ableitung, die anschließend berechnet werden muss, stehen, speichere ich diesen Wert in die Variable *delCdelZ*, um diese dreimal wieder zu verwenden.[47]

In den Zeilen 44-49 werden die Ableitungen $\frac{\partial C_n}{\partial w^{(L)}}$, $\frac{\partial C_n}{\partial b^{(L)}}$ und $\frac{\partial C_n}{\partial a^{(L-1)}}$ berechnet (s. die Formeln 4-6 im Kapitel 8.1).

Schlussendlich werden in den Zeilen 51-52 all die oben kalkulierten Ableitungen (Gradientenvektoren) von den jeweiligen Anpassungsmatrizen subtrahiert. Genau diese zwei Zeilen lenken allmählich die Ausgabe des Netzwerks in die gewünschte Richtung. Der Wert *activationGradient* wird zurückgegeben und in der Funktion *Network::Backpropagation* von den Aktivierungen der vorigen Schicht abgezogen. Im nächsten Zyklus der For-Schleife der Funktion *Network::Backpropagation* wird dann diese Differenz als Fehler der aktuellen Schicht interpretiert und erneut der Funktion *Layer::PropagateError* übergeben, bis die erste Schicht erreicht ist.[48]

9.5 ApplyAdjustments

Die Funktion *ApplyAdjustments* ist recht einfach, sie ist folgendermaßen definiert:

[45] s. Formeln 1-3 im Kapitel 8.
[46] s. https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e
[47] vgl. Kapitel 8.2.
[48] s. Exzerpt 16.

```
91 // Exzerpt 18, Commit: d637c72, Datei: src/AI/Network.cpp
92 void Network::ApplyAdjustments(
93    NetworkAdjustments& adjustments,
94    const size_t miniBatchSize,
95    const Num eta)
96 {
97    for (size_t l = 1; l < layers.size(); ++l)
98    {
99       layers[l].w += adjustments[l].w * (eta / miniBatchSize);
100      layers[l].b += adjustments[l].b * (eta / miniBatchSize);
101      adjustments[l].w.setZero();
102      adjustments[l].b.setZero();
103   }
104 }
```

Es mag an dieser Stelle falsch erscheinen, dass die Anpassungen zu den aktuellen Parametern addiert werden, anstatt diese zu subtrahieren, jedoch möchte ich darauf hinweisen, dass der Gradient *adjustments* bereits in die Richtung des steilsten Abstiegs zeigt, d.h. er ist bereits negativ.[49] In den Zeilen 101-102 werden die Anpassungen gleich null gesetzt. Hier könnte man einerseits in jeder Iteration das ganze Objekt neu erstellen, andererseits erfordert dies neue Heap-Speicherzuweisungen, in der Regel ist diese Operation relativ langsam[50], weswegen ich darauf verzichte.

10 Test

Zu diesem Zeitpunkt sollte das Netzwerk in der Lage sein, trainiert zu werden, doch zuerst einmal möchte ich zeigen, wie sich ein nicht trainiertes Netzwerk verhält. Dafür habe ich den folgenden Test implementiert:

```
43 // Exzerpt 19, Commit: d637c72, Datei: src/Main.cpp
44 for (size_t i = 0; i < numTests; ++i)
45 {
46    const AI::TrainingSample& randomSample{
47       testData[rand() % testData.size()]
48    };
49    const AI::Matrix output{
50       network.Feedforward(randomSample.input)
51    };
```

[49] Siehe Exzerpt 17., Zeile 51-52.
[50] https://stackoverflow.com/a/80113, Teil “What makes one faster?”

```
52      const size_t prediction{
53          AI::Util::FindGreatestIndex(output)
54      };
55      const size_t rightAnswer{
56          AI::Util::FindGreatestIndex(randomSample.output)
57      };
58
59      if (prediction == rightAnswer)
60      {
61          rightPredictions++;
62      }
63  }
```

Diesen Test möchte ich folgendermaßen intuitiv erklären: wenn wir einem trainierten Netzwerk die Ziffer *9* präsentieren, erwarten wir im Idealfall (wie es im MNIST-Datensatz der Fall ist) folgende Ausgabe: *[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]* (es „feuert" das Neuron, dem die Ziffer 9 zugewiesen ist), doch es ist ziemlich unrealistisch, dies von einem Netzwerk zu erwarten, denn die Ausgabe eines Netzwerks ist eine Komposition von reellen Zahlen im Intervall [-1; 1), daher sähe das beste Ergebnis, mit dem man rechnen kann, in etwa so aus: *[0, 0.001, 0.001, 0.002, 0.001, 0.002, 0, 0.001, 0, 0.992]*.[51]

Schlussendlich beinhaltet auch der MNIST-Datensatz solche Ziffern, bei denen auch unsere biologischen neuronalen Netze nicht ganz eindeutig entscheiden können, um welche Ziffern es sich dabei handelt:

Abbildung 5. Quelle: https://people.cs.umass.edu/~smaji/projects/digits/index.html

[51] Das kann man so interpretieren, dass keine Prognosen mit hundertprozentiger Sicherheit gemacht werden können.

Der im Exzerpt 20. dargestellte Test macht folgendes: Er vergleicht den Index des größten Wertes in der gewünschten Ausgabe mit dem Index des größten Wertes in der eigentlichen Ausgabe des Netzwerks, also er vergleicht das Neuron, das aktuell feuert, mit dem, das bei der gegebenen Eingabe feuern muss. Wenn diese gleich sind, stuft der Test den Versuch als richtig ein.

Dann wird der prozentuale Anteil der als richtig eingestuften Versuche angezeigt:

```
69 // Exzerpt 20, Commit: d637c72, Datei: src/Main.cpp
70 LINE_OUT("Genauigkeit: %.1f%%", rightPredictions / numTests * 100.f);
```

Die Ausgabe für ein nicht trainiertes Netzwerk (ohne den Funktionsaufruf *Network::SGD*) ist, wie folgt:

```
>> Genauigkeit: 8.9%
```

Oder wenn ich den Test noch ein paar mal durchführe:

```
>> Genauigkeit: 8.0%
>> Genauigkeit: 10.1%
>> Genauigkeit: 8.8%
```

Kein Wunder, wenn man bedenkt, dass die Gewichtungen und Bias an dieser Stelle lediglich zufällig initialisiert sind. Die Wahrscheinlichkeit, richtig zu raten, liegt bei 10%. Doch was passiert, wenn wir unmittelbar vor der Durchführung des Tests die Funktion *Network::SGD* aufrufen?[52]

Nach dem ersten Durchgang hat das Programm folgende Ausgabe angezeigt:

```
>> Genauigkeit: 92.1%
```

Die weiteren Versuche haben ähnliche Werte angezeigt:

```
>> Genauigkeit: 91.6%
>> Genauigkeit: 93.2%
>> Genauigkeit: 92.7%
```

Aus diesen Resultaten und der Überlegung, dass die Wahrscheinlichkeit, richtig zu raten, lediglich bei 10% liegt, lässt sich schließen, dass der Trainingsalgorithmus funktioniert.

11 Vergleich mit dem Netzwerk von Michael Nielsen

Die früher erwähnte Implementierung von Michael Nielsen ist unter folgendem Link zu finden: https://github.com/mnielsen/neural-networks-and-deep-learning. Diese enthält die Datei src/network.py. Für diesen Vergleich habe ich diese Quellcodedatei allerdings ein wenig verändert, sodass das Programm nach jeder Epoche die Zeit anzeigt, die es für diese Epoche

[52] Siehe Exzerpt 14.

gebraucht hat. Dasselbe habe ich meiner Implementierung hinzugefügt. Für diesen Vergleich werden mit Hilfe der beiden Projekte die Netzwerke folgendermaßen erstellt:

```
// Nielsens Implementierung in Python.
// Exzerpt 21, Datei: src/network.py
net = network.Network([784, 16, 16, 10])
```

beziehungsweise:

```
// Meine Implementierung in C++.
// Exzerpt 22, Commit: d637c72, Datei: src/Main.cpp
AI::Network network({
    { 784 },
    { 16 },
    { 16 },
    { 10 }
});
```

und mit diesen Funktionsaufrufen trainiert:

```
// Nielsens Implementierung in Python.
// Exzerpt 23, Datei: src/network.py
net.SGD(training_data, 30, 20, 3.0, test_data=test_data)
```

und

```
// Meine Implementierung in C++.
// Exzerpt 24, Datei: src/Main.cpp
network.SGD(trainingData, 20, 1, 3.0f);
```

Wenn ich die beiden Programme nacheinander ausführe, zeigen sie folgende Ausgaben:

Das Netzwerk in Python:

```
>> SGD-Dauer: 219s.
>> Durchschnittliche Genauigkeit: 92.39%
```

Das Netzwerk in C++:

```
>> SGD-Dauer: 46s.
>> Durchschnittliche Genauigkeit: 93.42%
```

Unten sind die Ergebnisse von fünf weiteren Tests (links C++, rechts Python):

```
>> SGD-Dauer: 47s.
>> Durchschnitt. Genauigkeit: 93.51%
```

```
>> SGD-Dauer: 205s.
>> Durchschnitt. Genauigkeit: 92.69%
```

```
>> SGD-Dauer: 45s
>> Durchschnitt. Genauigkeit: 93.65%
```

```
>> SGD-Dauer: 208s.
>> Durchschnitt. Genauigkeit: 92.45%
```

```
>> SGD-Dauer: 46s.
>> Durchschnitt. Genauigkeit: 93.61%
```

```
>> SGD-Dauer: 203s.
>> Durchschnitt. Genauigkeit: 92.54%
```

```
>> SGD-Dauer: 46s.
>> Durchschnitt. Genauigkeit: 93.44%
```

```
>> SGD-Dauer: 218s.
>> Durchschnitt. Genauigkeit: 93.13%
```

```
>> SGD-Dauer: 46s.
>> Durchschnitt. Genauigkeit: 93.59%
```

```
>> SGD-Dauer: 203s.
>> Durchschnitt. Genauigkeit: 92.66%
```

Anhand der Tabelle wird deutlich, dass das in C++ implementierte Netzwerk ungefähr 4.5-mal weniger Zeit braucht, um ungefähr dieselbe Genauigkeit zu erreichen, was wiederum diejenige Hypothese bestätigt, die unter anderem der Gegenstand dieser Arbeit ist und im Kapitel 4.2. „Eine mögliche Lösung“ aufgestellt wird.

12 Diese Erkenntnis in den führenden Projekten

In diesem Kapitel wird die Rolle dieser Erkenntnis in den führenden KI-Bibliotheken, vor allem in TensorFlow, detaillierter untersucht. Die Ergebnisse der bisherigen in dieser Arbeit behandelten Hypothesen und Tests schaffen eine gewisse Diskrepanz: einerseits zeigen die kompilierten Programmiersprachen eindeutig bessere Ergebnisse, andererseits ist Python trotzdem die am weitesten verbreitete Programmiersprache in der KI-Industrie.

12.1 Wahl der Programmiersprachen

Wenn man allerdings die Repositories der berühmten KI-Bibliotheken betrachtet, fällt sofort ein sehr interessantes Merkmal auf:

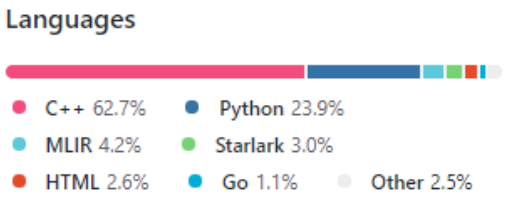


Abbildung 6. Programmiersprachen in TensorFlow

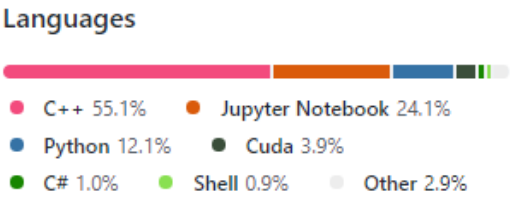


Abbildung 7. Programmiersprachen in Microsoft CNTK

Die zwei sehr berühmte Projekte (links TensorFlow, rechts Microsoft CNTK) bestehen zu 63% bzw. 55% aus C++ Code[53], jedoch werden sie fast ausschließlich mit Python benutzt.

12.2 Dynamische Bibliotheken

C++ spielt dabei die folgende Rolle: die beiden Bibliotheken sind eigentlich in C++ implementiert, weil deren Entwickler jener Probleme und Lösungen, die auch in dieser Arbeit

[53] vgl. https://github.com/tensorflow/tensorflow bzw. https://github.com/microsoft/CNTK

beschrieben werden, durchaus bewusst sind[54]. Das Endziel der Kompilierung dieser C++ Projekte ist allerdings nicht eine Anwendung, sondern eine dynamische Bibliothek, wie z. B. eine dll-Datei. Diese Dateien sind wie Programme, aber es gibt keinen Einsprungpunkt, sie beinhalten einfach alle Klassen und Funktionen, welche vorher in C++ implementiert und kompiliert worden sind. Anschließend können diese dll-Dateien in einer anderen Programmiersprache (z. B. Python) geladen und die in ihnen enthaltenen Funktionen aufgerufen werden.

12.3 Vorteile

Die Vorteile dieser Vorgehensweise sind vor allem folgende:

- Der potentielle Benutzer der Bibliothek ist nicht an eine bestimmte Programmiersprache gebunden. Das ermöglicht eine große Flexibilität bei der Wahl der Programmiersprache.
- Das Endprodukt (die Bibliothek) muss nicht in jeder gewünschten Programmiersprache implementiert werden, sondern es wird einmal implementiert und kompiliert, und dann von allen möglichen Interfaces benutzt.

13 Fazit

Wie die Praxis zeigt, ist es in Informatik äußerst wichtig, passende Instrumente für konkrete Aufgaben zu wählen. In Nielsens Implementierung ist die Wahl von Python dadurch gerechtfertigt, dass er es gut für Bildungszwecke gestalten wollte. Wenn man allerdings die Leistung des Programms als Hauptziel setzt (was man in denjenigen KI-Projekten, deren Ziel nicht Bildung ist, wahrscheinlich tun sollte), bringen die kompilierten Programmiersprachen viele Vorteile, die in diesem Teilgebiet entscheidend sind.

[54] https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html Chapter “How TensorFlow works”

14 Quellen

Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015

http://yann.lecun.com/exdb/mnist

https://de.wikipedia.org/wiki/C_(Programmiersprache)

https://de.wikipedia.org/wiki/C%2B%2B

https://de.wikipedia.org/wiki/Künstliche_Intelligenz

https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz

https://de.wikipedia.org/wiki/Leibniz-Reihe

https://de.wikipedia.org/wiki/Neuronales_Netz

https://de.wikipedia.org/wiki/Python_(Programmiersprache)

https://de.wikipedia.org/wiki/Überwachtes_Lernen

https://en.wikipedia.org/wiki/Artificial_neural_network

https://en.wikipedia.org/wiki/Artificial_neuron

https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)

https://en.wikipedia.org/wiki/Learning_rate

https://en.wikipedia.org/wiki/Loss_function

https://en.wikipedia.org/wiki/Python_(programming_language)

https://en.wikipedia.org/wiki/Stochastic_gradient_descent

https://github.com/microsoft/CNTK

https://github.com/mnielsen/neural-networks-and-deep-learning

https://github.com/tensorflow/tensorflow

https://link.springer.com/article/10.1007/s12065-020-00540-3

https://michaelnielsen.org

https://scikit-learn.org/stable/install.html

https://stackoverflow.com/a/80113

https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

https://www.baeldung.com/cs/epoch-neural-networks

https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185

https://www.computerwoche.de/a/die-besten-coding-sprachen-fuer-ki

https://www.facebook.com/business/news/good-questions-real-answers-how-does-facebook-use-machine-learning-to-deliver-ads

https://www.golem.de/news/deep-learning-maschinen-die-wie-menschen-lernen-1510-116468-3.html

https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html
https://www.tensorflow.org/install
https://www.westhouse-group.com/ki-programmiersprachen-python-c-java-lisp-prolog
https://youtu.be/aircAruvnKk
https://youtu.be/IHZwWFHWa-w
https://youtu.be/tIeHLnjs5U8

Auf alle oben stehenden Quellen wurde zuletzt am 11.02.2022 zugegriffen.

15 Angewandte Programmbibliotheken

GLEW (https://github.com/nigels-com/glew)	Letzter Zugriff: 29.10.2021
GLFW (https://github.com/glfw/glfw)	Letzter Zugriff: 29.10.2021
ImGui (https://github.com/ocornut/imgui)	Letzter Zugriff: 29.10.2021
MNSIT-Reader (https://github.com/wichtounet/mnist)	Letzter Zugriff: 03.11.2021
eigen (https://gitlab.com/libeigen/eigen)	Letzter Zugriff: 07.02.2022

16 Anhang

Material 1.

Es wäre sinnlos, jede einzelne Zeile des Quellcodes in dieser Facharbeit zu beschreiben, dennoch erfordern einige Code-Auszüge eine ausführliche Erklärung, weswegen ich den Quellcode folgendermaßen gestaltet habe:

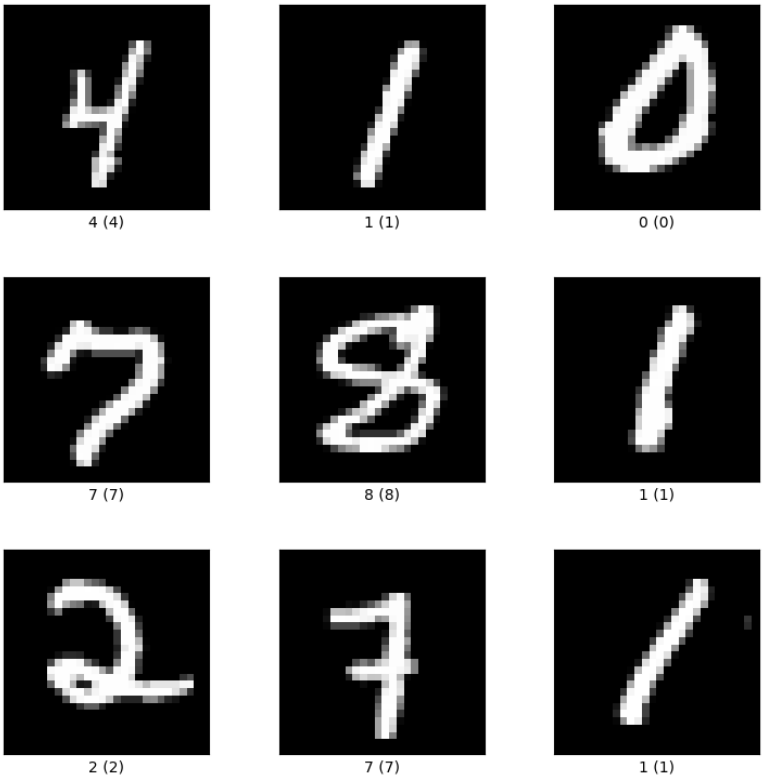
- Der gesamte Quellcode ist auf GitHub gespeichert und unter folgendem Link verfügbar: https://github.com/eugen-bondarev/BLL.
- Es werden in dieser Arbeit nur für das aktuelle Thema und Zusammenhang relevante Codezeilen gezeigt und erläutert.
- Falls sich der Leser den Code jedoch genauer ansehen möchte, kann er dies sehr bequem tun, indem er diesem Link folgt: https://github.com/eugen-bondarev/BLL/commit/ und nach "/BLL/commit/" die Identifikationsnummer des interessierenden Commits eingibt, die ich sehr gerne in der Kopfzeile jedes Codefensters hinterlasse.

Wenn sich der Leser beispielsweise für den folgenden Code interessiert,

```
 9 // Commit: 31797e3, Datei: src/AI/LayerDescriptor.h
10 struct LayerDescriptor
11 {
12     size_t numNeurons;
13     // Mathematische funktion f(x).
14     Activation g;
15 };
```

so kann er dem Link https://github.com/eugen-bondarev/BLL/commit/31797e3 folgen, auf "Browse files" klicken, die Datei src/AI/LayerDescriptor.h finden und sich diese ansehen.

Material 2. MNIST-Datensatz, Beispiel



Quelle: https://www.tensorflow.org/datasets/catalog/mnist