Ziffernerkennung mithilfe eines künstlichen neuronalen Netzes

Alexander Vogel (791422)

29. März 2018

Inhaltsverzeichnis

1 Ziel

Ziel der, in Python geschriebenen, Software ist es handgeschriebene Ziffern mithilfe von künstlichen neuronalen Netzen zu erlernen und zu erkennen.

2 Theoretischer Hintergrund

2.0.1 Allgemeiner Aufbau Netzwerk

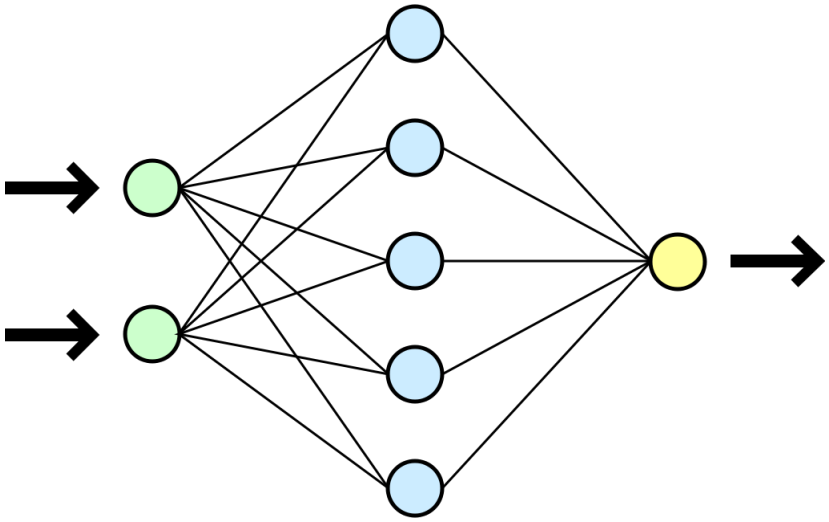

Abbildung 1: Allgemeiner Aufbau eines Feed Forward Netzes

Die vereinfachte Darstellung des künstlichen neuronalen Netzwerks zeigt den allemeinen Aufbau. Die zwei Grünen Eingangsneuronen bilden die Erste Schicht auch den Input-Layer genannt. Jedes Neuron aus dem Input-Layer ist mit jedem Neuron aus der zweiten Schicht(Hidden-Layer in Blau) verbunden. Jedes Neuron aus dem Hidden-Layer ist wiederum mit jedem Neuron aus der Ausgabeschicht (Output-Layer in Gelb) verbunden.

Es gibt weitaus dynamischere und komplexere Netzwerke. Für diesen zweck habe ich mich aber für ein einfaches Feed Forward Netz entschieden. Da es für die Aufgabe vollkommen ausreichende Ergebnisse liefert. Außerdem ist die Einstiegsschwelle angemessen um einen guten Einstieg in das Thema zu erlangen und um ein Grundverständnis aufzubauen.

2.0.2 Allgemeiner Aufbau Layer

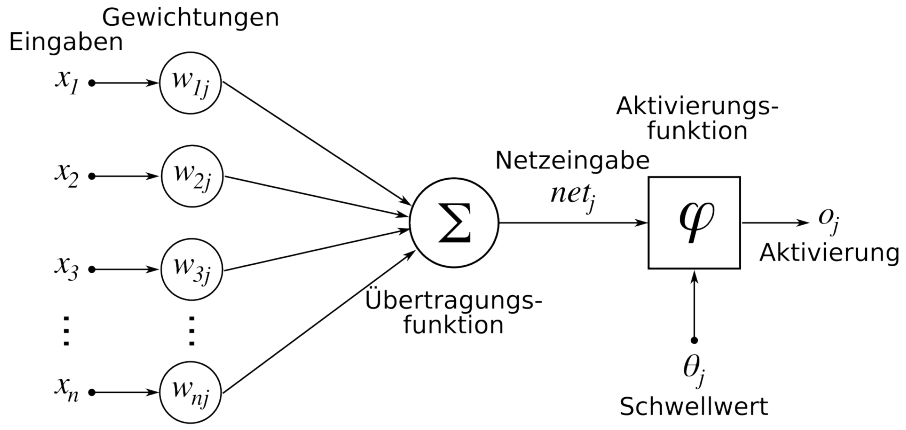

Abbildung 2: Allgemeiner Aufbau eines Layers

Jedes Neuron aus jedem Layer ist wie das Diagramm aus Abbildung 2 aufgebaut. Es summiert die Ausgabe, der davorliegenden Neuronenschicht, mit der jeweiligen Gewichtung der Verbindung. Die Netzeingabe wird dann in die Aktivierungsfunktion gegeben, in diesem Fall die Sigmoidfunktion. Der Ausgabewert dient dann wieder als Eingabe des nächsten Layers bzw. als Ausgabe des gesamten Netzwerks, falls die aktuelle Schicht die Ausgabeschicht ist.

2.0.3 Aktivierungsfunktion

Abbildung 3: sig(t) Funktion in Rot und sig'(t) in Blau

Als Aktivierungsfunktion habe ich die Sigmoidfunktion gewählt, dies ist eine logistische Funktion. Sie muss differenzierbar sein um den Backpropagation Lernalgorithmus anzuwenden.

$$sig(t) = \frac{1}{1 + e^{-t}} \tag{1}$$

Die Ableitung dieser Funktion lautet:

$$sig'(t) = sig(t) \cdot (1 - sig(t)) \tag{2}$$

diese wird für das Gradientenabstiegsverfahren benötigt.

3 Umsetzung

3.1 Zurechtschneiden des Rohbildes

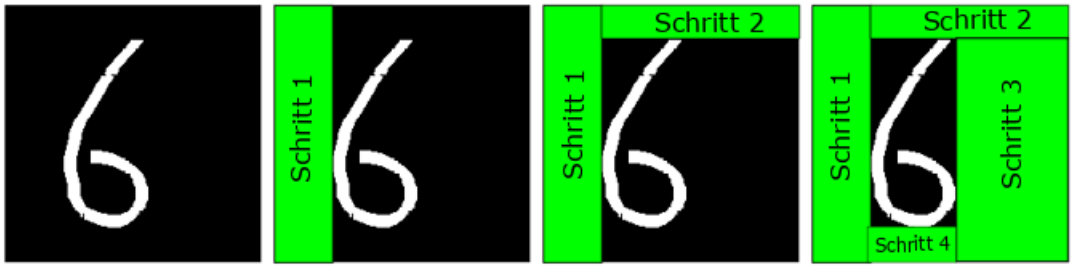

Abbildung 4: Ausschneiden der Ziffer auf die effektive Größe

Um das Rohbild zurechtzuschneiden gibt es eine Funktion find_bounding_box. Diese liefert das kleinste Rechteck zurück in den die Ziffer hineinpasst, damit lässt sich exakt die Ziffer aus dem Zeichenbereich rausschneiden. Somit spielt es keine Rolle in welcher Größe der User die Ziffer zeichnet oder an welchen Ort er diese auf dem Zeichenbereich zeichnet.

Die Funktion tastet sich von allen vier Seiten an die Ziffer heran bis sie den ersten weißen Pixel findet. Dadurch grenzt sie den Bereich um die Ziffer ein und gibt zum Schluss eine Box zurück mit den minimalen Abmaßen der Ziffer.

Diese Funktion war sehr Zeitkritisch, da sie in Echtzeit das gezeichnete Bild zurechtschneiden sollte.

Am Anfang hat die Funktion die komplette Breite und Höhe des Bildes abgesucht, nach der Optimierung braucht die Funktion nur soweit zu suchen bis sie an die Grenze der zuvor gefundenen Barriere stöst.

Abbildung 5: Ziffer nach dem Skalieren auf 20x20 Pixel

Nach dem Zurechtschneiden, musste die Ziffer auf die Größe der Eingabematrix (20x20 Pixel) skaliert werden, dies geschieht proportional damit die Ziffer dabei nicht verzerrt wird.

3.2 Layer

3.2.1 Input-layer

Die Rohdaten des Eingabebildes werden normalisiert und danach in diesem Zustand in die Eingabeschicht des neuronales Netzwerks gegeben. Die Eingabeschicht besteht, bei unserem Netzwerk aus 400 Neuronen, welches sich durch die Vektorisierung des 20 Pixel breiten und 20 Pixel hohen Eingabebildes ergibt.

$$Anzahl\ Eingabeneuronen = 20 \cdot 20 = 400$$

Da wir nur mit Schwarz/Weiß Werten bzw. Grauwerten arbeiten, besitzen die Farbkanäle des Bildes alle denselben Wert zwischen 0 und 255 (Rot, Grün, Blau) z.B. Schwarz (0,0,0) und (255,255,255) für Weiß. Beim normalisieren nehmen wir uns einen Kanal heraus und teilen den IST-Wert des Kanals durch den maximal möglichen Wert(255).

$$Normalisierter\ Wert = \frac{IST - Wert}{255}$$

Dadurch erhalten wir einen normalisierten Wert zwischen 0 und 1.

3.2.2 Hidden-Layer

Der Hidden-Layer, die zweite Schicht im Netzwerk besteht in unserem Fall aus 10 Neuronen. Diese Schicht ist mit unseren 400 Eingabeneuronen verbunden. Da jedes Neuron aus der Eingabeschicht mit jedem Neuron aus der zweiten Schicht verbunden ist, haben wir hier eine Gewichtungsmatrix von 400x10 mit insgesamt 4000 Gewichten.

3.2.3 Output-Layer

Der Output-Layer, die dritte und letzte Schicht in unserem neuronales Netzwerk besitzt 10 Ausgabeneuronen, von denen jeweils ein Neuron eine Ziffer repräsentiert an die das Netzwerk "denken"kann. Durch die Aktivierungsfunktion des Layers können wir hier nur werte zwischen 0 und 1 für jedes einzelne Neuron erwarten. Wobei Werte nahe an 0 für wenig Aktivivität stehen und Werte die nah an der 1 liegen für viel Aktivität.

3.3 Ausgabe im Userinterface

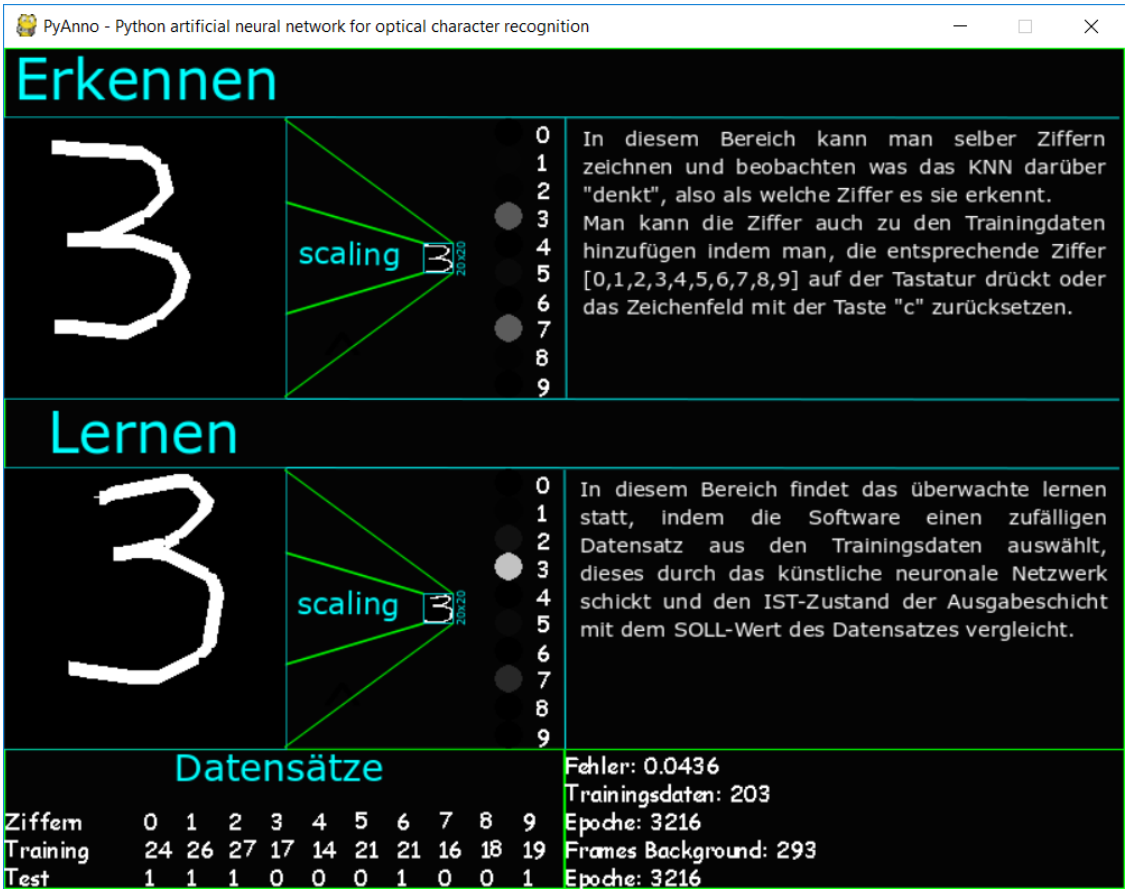

Abbildung 6: Ziffer nach dem Skalieren auf 20x20 Pixel

Um die Ausgabeschicht für den Benutzer zu visualisieren, habe ich mich dafür entschieden jedes Neuron der Ausgabeschicht, als Kreis darzustellen, dessen Grauwert die Aktivität des jeweiligen Neurons wiederspiegelt. Dadurch ist gut zu erkennen, welche Ziffer das KNN dem Eingabebild zuordnet.

3.4 Benchmarking

3.4.1 Ablauf

Das Benchmarking läuft parallel zum Lernen des neuronalen Netzes, dabei wird der Fehler und die Anzahl der Lerniterationen gespeichert und beim Schließen des Programms für jede der Ziffern von 0 bis 9 als CSV Datei abgespeichert. Somit kann man den Lernerfolg nach jeder Iteration nachvollziehen. Die Folgenden 3 Grafiken zeigen den Lernerfolg für drei ausgewählte Ziffern nach jeweils 60.000 Iterationen.

3.4.2 Ziffer 0

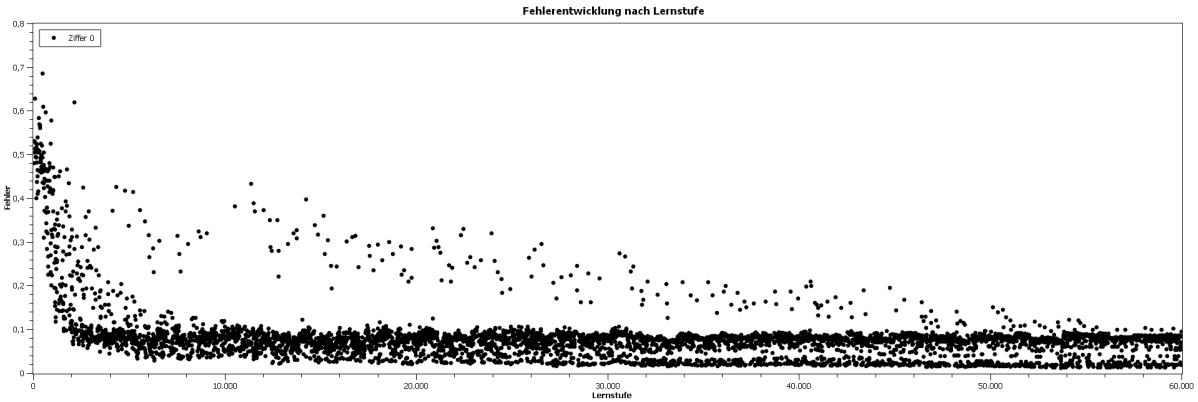

Abbildung 7: Ziffer 0 nach 60.000 Lerniterationen

Man kann erkennen, dass sich manche Ziffern aus den Trainingsdaten der 0 nur sehr langsam einem geringen Fehler annähern, nach 60.000 Iterationen ist aber keine Abweichung mehr zu den restlichen Trainingsziffern zu erkennen und alle Fehler bleiben konstant unter 0,1.

3.4.3 Ziffer 3

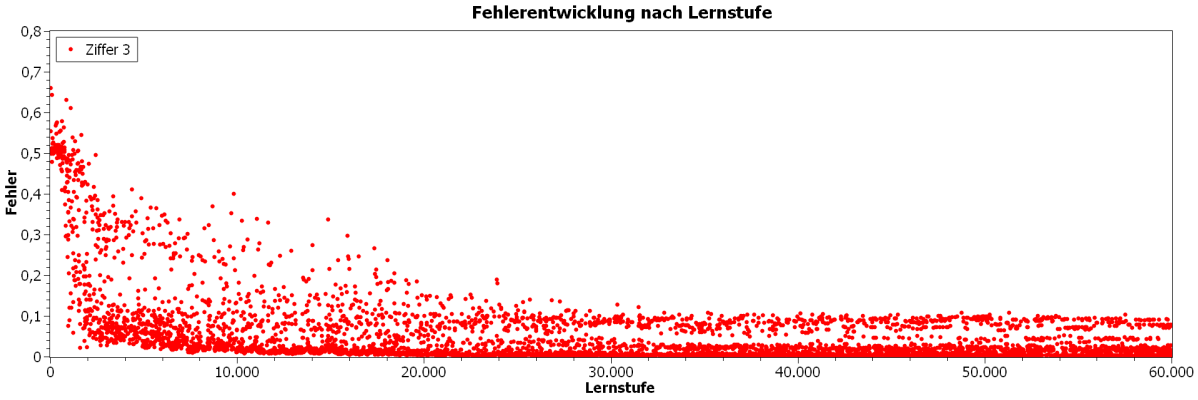

Abbildung 8: Ziffer 3 nach 60.000 Lerniterationen

Die Trainingsdaten der Ziffer 3 nähern sich nach ca. 30.000 Iterationen einem maximalen Fehler von 0,1 an.

3.4.4 Ziffer 7

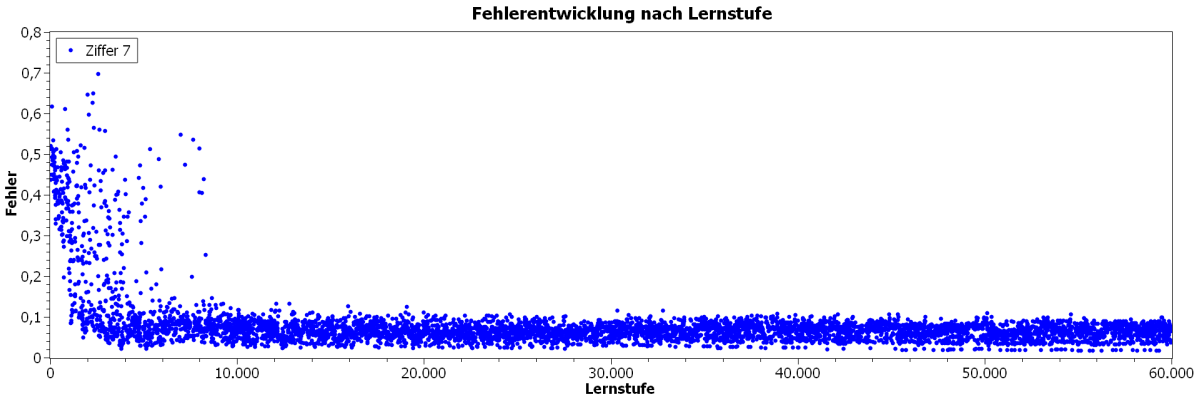

Abbildung 9: Ziffer 7 nach 60.000 Lerniterationen

Alle Trainingsdaten der Ziffer 7 nähern sich schon nach ca. 10.000 Iterationen einem maximalen Fehler von 0,1 an.

3.5 Verzeichnisstruktur

3.5.1 data

Im data Ordner befindet sich der Unterordner weights, dort werden die Gewichts-Matrizen der einzelnen Layer abgespeichert. Um sie beim nächsten Programmstart wieder zu öffnen.

3.5.2 gfx

Im Ordner gfx befinden sich alle statischen Grafiken, die zur Programmoberfläche gehören.

3.5.3 training

Im Ordner training werden die Grafiken der Ziffern als png Grafik gespeichert. Die Datensätze werden zum trainieren des Netzwerks benutzt. Jeder Datensatz enthält im Dateinamen eine eindeutige Identifikationsnummer und die Ziffer, die in der Grafik steckt.

3.5.4 testing

Der Ordner testing soll nur Ziffern enthalten, die ausschließlich dazu bestimmt sind das Netzwerk zu testen, und dessen Datensätze das Netzwerk aber nie gelernt hat.
(Die Struktur dazu ist soweit angelegt. Siehe ToDo.)

4 Versionen

Als Entwicklungsumgebung nutze ich PyCharm 2018.1
Python nutze ich in der Version 3.6
Es kamen folgende externe Bibliotheken zum Einsatz:
sqlite 3
numpy 1.14.2
pygame 1.9.3

5 To Do

Ich habe eine Klasse CStatistics erstellt, diese ist aber noch nicht fertig gestellt. Diese soll später helfen Statistiken über den Lernerfolg des künstlichen neuronalen Netzwerks zu erstellen.

6 Quellen

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