Motivation

Die digitale Bildverarbeitung mit Hilfe neuronaler Netzwerke hat in der Automobilindustrie einen enormen Wandel eingeleitet und spielt eine entscheidende Rolle in der Entwicklung autonomer Fahrzeuge. Die Relevanz dieses Zusammenspiels erstreckt sich über verschiedene Aspekte, von der Sicherheit bis zur Effizienz und Komfort. Neuronale Netzwerke ermöglichen es, komplexe visuelle Informationen in Echtzeit zu verarbeiten, was besonders wichtig ist, um Fahrzeuge mit einer hohen Autonomiestufe sicher auf den Straßen zu integrieren. So können zum Beispiel Verkehrsschilder wie Geschwindigkeitsbegrenzungen oder vorfahrtsregelnde Verkehrszeichen erkannt werden und das Fahrverhalten angepasst werden – entweder durch den Fahrenden selbst oder das automatisierte System. Auch das Erkennen bzw. Unterscheiden von Personen und Fahrzeugen, sowie das Identifizieren des Fahrstreifens sind wichtige Elemente, um autonomes Fahren zu ermöglichen und sicher zu gestalten**.**

Projektdefinition

Ziel dieses Projekts ist der Entwurf, das Training und eine abschließende Evaluation eines neuronalen Netzwerkes zur Bilderkennung. Als Datenset wird hier MNIST-Dataset aus der keras-Bibliothek verwendet, welches verschiedene 28x28 Pixel große Bilder der Ziffern 0-9 enthält (Abbildung 1). Um die Bilder zu erkennen und den jeweiligen Ziffern zuzuordnen, wird die Plattform TensorFlow verwendet. Diese wurde von einem Forschungsteam von Google erstellt, um bei der Entwicklung von Machine-Learning-Anwendungen zu helfen. Der Code wurde in der Entwicklungsumgebung Jupyter verfasst, in welcher es möglich ist in sogenannten Notebooks Code zu schreiben und nacheinander auszuführen.

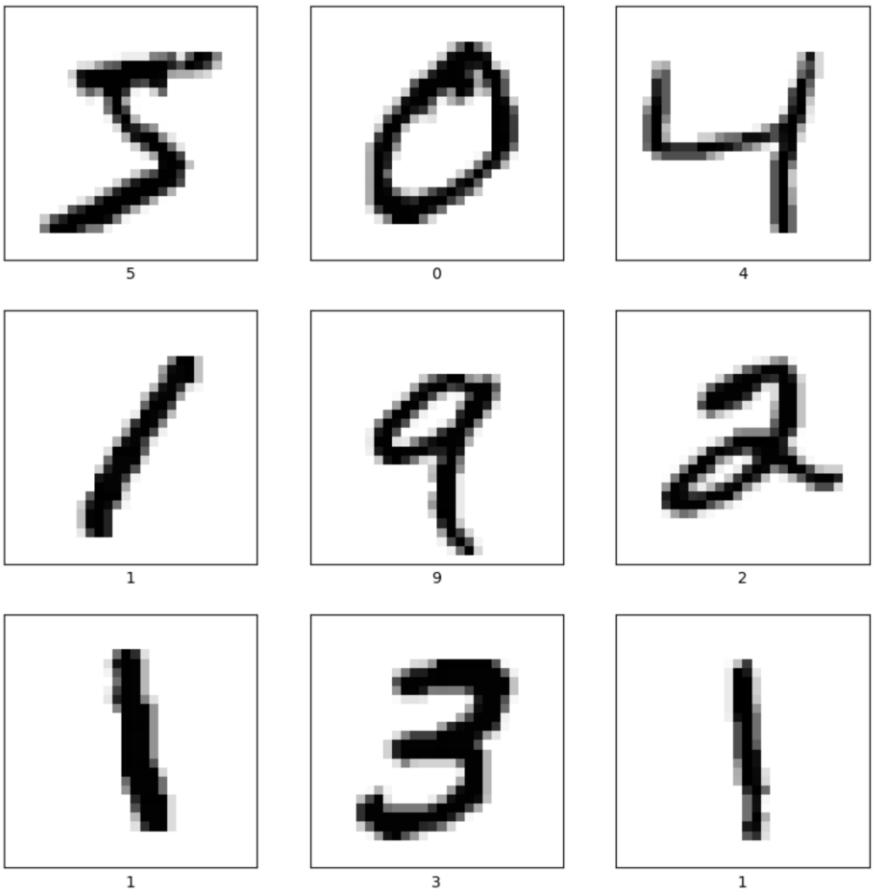

Abbildung 1: Auswahl an Daten aus dem MNIST-Datensatz

Grundlagen

- **Datensatz laden und speichern**
  - Der Datensatz wird in den Variablen x_train, y_train, x_test, y_test gespeichert und dessen Form (shape) ausgegeben.
  - x_train – 60.000 Daten/Bilder, mit jeweils 28x28 Pixeln → Bilddaten zum trainieren
  - y_train – 60.000 Daten/Ziffern → Ziffer, die auf dem Bild abgebildet ist
  - x_test – 10.000 Daten/Bilder, mit jeweils 28x28 Pixeln → Bilddaten zum testen
  - y_test – 10.000 Daten/Ziffern → Ziffer, die auf dem Bild abgebildet ist
  - Jeder Pixelwert besteht aus einem Graustufenwert von 0 (schwarz) bis 255 (weiß). Daraus folgt eine Größe von einem Byte pro Pixel und pro ganzes Bild 0,784 kByte.
  - Zusätzlich werden die ersten neun Daten des Datensatzes abgebildet
- **Datensatz anpassen**
  - Damit die Daten von TensorFlow verarbeitet werden können, müssen diese Normiert werden, d.h. die werde x-Daten werden durch 255 geteilt, wodurch sich Zahlenwerte zwischen 0 und 1 ergeben
  - Da das Modell später als Convolutional Neural Network (CNN) entworfen werden soll, muss zu den Daten eine zusätzliche Dimension hinzugefügt werden. Zu den vorhandenen drei Dimensionen (Anzahl der Bilder, Pixel Anzahl Breite, Pixel Anzahl Höhe) kommt eine vierte Dimension hinzu, welche den Kanal der Bilder repräsentiert. In diesem Fall ist dieser 1, was einem Graustufen-Kanal entspricht.
- **DEFINITION MODELL ARCHITEKTUR (PIXEL)**
  - Modell wird in Variable *pixel* gespeichert
  - Es wird die Klasse *sequential* verwendet, welche es ermöglicht das Modell mit Schichten (Layer) aufzubauen
  - Im Modell verwendete Layer:
    - **InputLayer:** Dieser Layer definiert den Eingabeplatzhalter für die Bilddaten. Die Eingabe hat eine Form von (28, 28, 1), was auf ein Graustufenbild mit einer Größe von 28x28 Pixeln hinweist.
    - **Conv2D:** Convolutional Layer mit 32 bzw. 64 Filtern und einer Filtergröße von (3, 3). Dieser Layer extrahiert Merkmale aus dem Bild.
    - **BatchNormalization:** Dieser Layer normalisiert die Aktivierungen des vorherigen Layers, um das Training zu stabilisieren und die Konvergenz zu beschleunigen.
    - **Activation (ReLU):** Aktivierungsfunktion Rectified Linear Unit (ReLU), die nicht-linearität in das Modell einführt, indem negative Werte auf Null gesetzt werden.
    - **MaxPooling2D:** Max-Pooling Layer mit einer Pooling-Größe von (2, 2), der die räumliche Dimension der Aktivierungen reduziert und die Berechnungskosten verringert.
    - **Dropout:** Dropout Layer mit einer Dropout-Rate von 25%. Dropout hilft beim Verhindern von Overfitting, indem zufällig eine bestimmte Anzahl von Neuronen während des Trainings ausgeschaltet wird.
    - **Flatten:** Flatten Layer, der die multidimensionalen Daten in einen flachen Vektor umwandelt, um sie für die Fully Connected-Schichten vorzubereiten.
    - **Dense (Fully Connected):** Fully Connected Layer mit 128 bzw. 64 Neuronen, L2-Regularisierung und ReLU-Aktivierung. Das Dense Layer kombiniert die verschiedenen komplexen Eingabeparameter.

        - **GaussianNoise:** Fügt dem Modell eine Gaußsche Rauschschicht hinzu, um die Robustheit gegenüber Rauschen zu erhöhen.
        - **Dense (Output):** Fully Connected Layer mit 10 Neuronen (entsprechend den Klassen des Problems) und einer Softmax-Aktivierung für die Klassifikation.
    - Insgesamt besteht das Modell aus 233.034 Parametern

Überlegungen zur Verbesserung des Modells (im Vergleich zum Grundmodell)

- Verwendung von mehreren Dropout Layer mit konstantem Dropout Wert, um overfitting zu vermeiden. Overfitting beschreibt das Verhalten eines Modells, wenn die accuracy bei den Trainingsdaten höher ist als bei den Testdaten. Somit kann das Modell zwar gut die Daten erkennen, mit denen es trainiert wurde, jedoch kann es schlecht mit unbekannten Daten umgehen. Durch Anpassung des Dropout Wertes kann das Netz weiter optimiert werden, in diesem Fall wurde ein optimaler Wert von 0,25 festgestellt. Das bedeutet, dass jedes Neuron eine Wahrscheinlichkeit von 25% hat, während eines Trainingsdurchlaufs deaktiviert zu werden. Dadurch wird das Netz weiter variiert und robuster gemacht https://aws.amazon.com/de/what-is/overfitting/
- Verwendung von einer Kernelgröße von 3x3 statt 5x5, da es bei der Ziffererkennung hauptsächlich auf die Erkennung von kleinen Regionen bzw. lokalen Merkmalen wie Kurven oder Kanten ankommt. Daraus ergibt sich eine Reduzierung der Parameter, was zu erhöhter Rechengeschwindigkeit führt. Falls die Rechnerressourcen ausreichend sind kann eine Kernelgröße von 5x5 verwendet werden, um die accuracy weiter zu erhöhen https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363
- Mehrere Convolutional-Layer mit unterschiedlichen Filter Größen, um unterschiedliche Merkmale zu erfassen. Umso höher die Filterzahl, desto komplexere Merkmale können erkannt werden - jedoch erhöht dies auch die Parameter zahl. Um einen Kompromiss aus accuracy und Parameteranzahl zu finden wurden im Pixel Modell zwei Convolutional-Layer mit einer Filtergröße von 32 (lokale Merkmale) und 64 (komplexere Merkmale) integriert. https://de.wikipedia.org/wiki/Convolutional_Neural_Network
- Integration von zusätzlicher Batch-Normalisation, Standardisiert die Trainingsdaten durch Berechnung von Mittelwert und Standartabweichung, wodurch das Modell stabiler wird. https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
- Hinzufügen einer GaussianNoise-Schicht, um zusätzliches Rauschen einzubauen, wodurch mehr Varianz generiert wird und folglich das Modell generell robuster gegenüber dem Rauschen aus den eigentlichen Daten machen soll. Dabei wurde ein Wert 0,1 durch experimentelles Ausprobieren ermittelt. https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting/
- Einfügen eines Regularisierung: in diesem Fall eine L2 Regularisierung mit der Gewichtung von 0,07. Diese Regularisierung führt Strafterme ein, welche dazu dienen, das Modell mehr zu generalisieren und somit overfitting zu reduzieren. https://keras.io/api/layers/regularizers/

Modell trainieren

- Training des Modells mithilfe der *fit* Methode
- Trainingsdaten sind x_train und y_train
- Modell wird 30 Zyklen lang trainiert (Epochen)
- Es werden 516 Trainingsbeispiele (Batch) geladen, bevor die Gewichtung aktualisiert wird
- Die Trainingsdaten werden für jede Epoche neu gemischt (shuffel = true)
- Die Validation des Modells erfolgt über die Testdaten x_test und y_test

Auswertung

- **Loss:** 0,0790 / 7.9 %
- **Accuracy**: 0,9893 / 98,9 %
- Die accuracy der Trainingsdaten beträgt 98,7 %, somit wurde overfitting erfolgreich verhindert
- Wie gut das Modell welche Ziffern erkennt, kann mithilfe einer Confusion Matrix, wie in Abbildung 2 zu sehen, ausgewertet werden
    - Vertikale Achse: echte / richtige Ziffern
    - Horizontale Achse: erkannte / vorhergesagte Ziffern
    - Richtige vorhersage der einzelnen Ziffern:
        - Ziffer 0: 0,9918
        - Ziffer 1: 0,9991
        - Ziffer 2: 0,9893
        - Ziffer 3: 0,9920
        - Ziffer 4: 0,9938
        - Ziffer 5: 0,9955
        - Ziffer 6: 0,9822
        - Ziffer 7: 0,9951
        - Ziffer 8: 0,9825
        - Ziffer 9: 0,9712
    - Ziffern 1, 5, 7 werden gut erkannt
        - Tendenziell harte Kanten, gerade Striche
    - Ziffern 6, 8, 9 werden eher schlechter erkannt
        - Viele Kurven, Kreise, Bögen
        - Ziffer 9 wird u.a. für eine 4 oder 7 gehalten, ähnlicher Aufbau, jedoch ist die 9 kurviger

Ausblick

Um das Modell im Bezug auf accuracy noch weiter zu verbessern kann versucht werden, Kurven und Bögen besser zu erkennen und zu interpretieren. Dafür könnten zum Beispiel noch weitere Convolutional-Layer mit einer höheren Filtergröße (z.B. 128) hinzugefügt werden. Dadurch würde sich aber die Parameterzahl weiter erhöhen und somit die Trainingszeit verlängern.

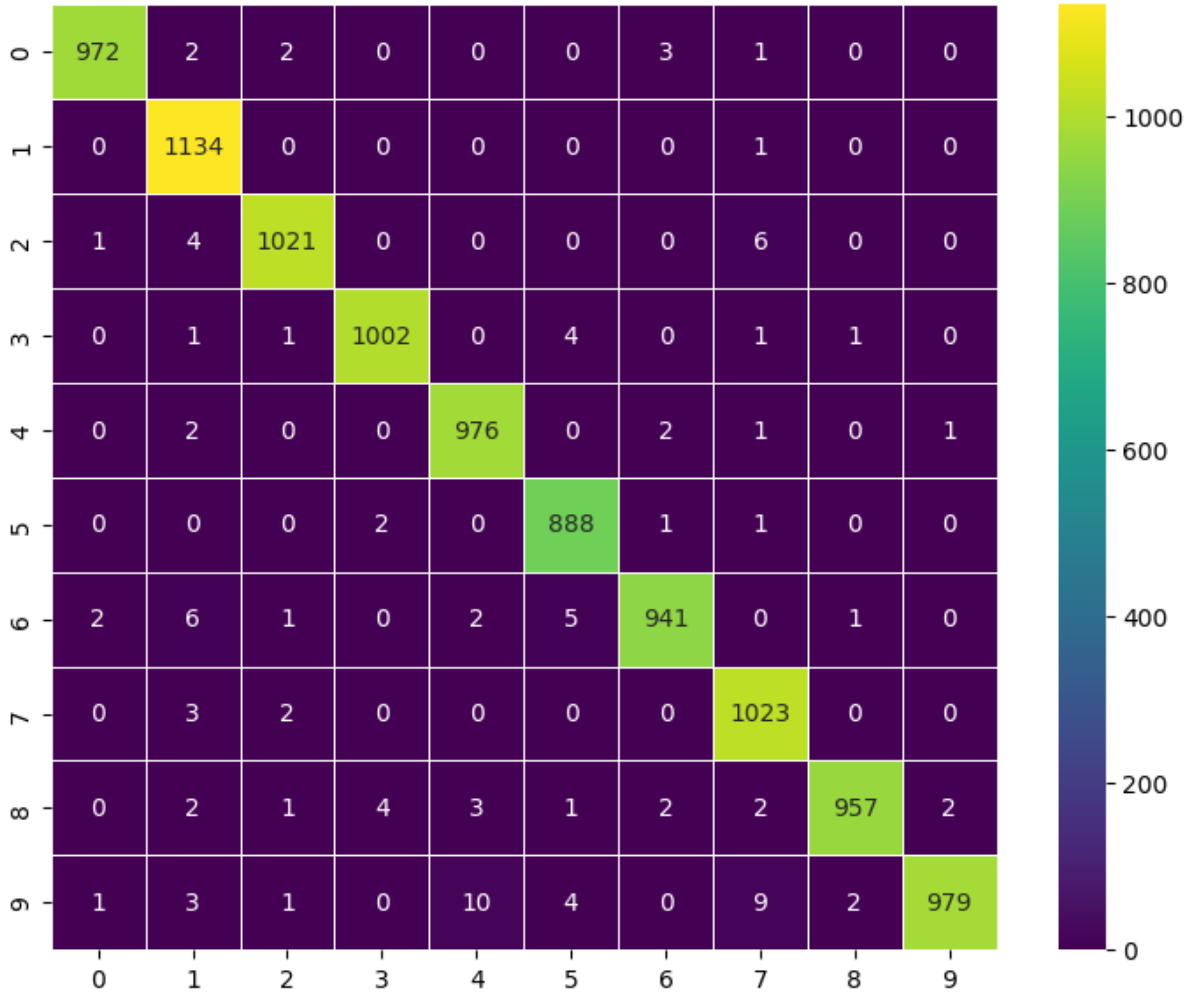

Abbildung 2: Confusion Matrix - Auf der vertikalen Achse sind die echten Werte und auf der horizontalen Achse die erkannten Werte abgebildet