DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

Dokumentation

Training eines neuronalen Netzes

Abgabe digitale Bildverarbeitung

Studiengang Elektrotechnik

Studienrichtung Fahrzeugelektronik

Duale Hochschule Baden-Württemberg Ravensburg, Campus Friedrichshafen

von

Martin Förstemann

Abgabedatum:	21. November 2023
Bearbeitungszeitraum:	12.11.2023 - 05.01.2024
Matrikelnummer:	8488669
Kurs:	TFE21-2

1 Konzept und Rahmenbedingungen

Im voraus sind zu erreichende Trainingsziele für das Netz und die Datenvorverarbeitung gesetzt worden. Die Daten sollen auf der lokalen Hardware in unter 30 s verarbeitet werden. Zudem soll der Datensatz nicht maßgeblich vergrößert werden (z.B. durch spiegeln der Bilder und somit Verdopplung der Bilderanzahl).

Das Modell selbst soll auf der lokalen Hardware in unter einer Minute trainiert werden können, die Anzahl der Parameter soll unter 1000 betragen und die Genauigkeit auf den Testdatensatz soll bei über 90% liegen.

Dazu wurde ein ursprüngliche Netz aufgestellt, das mit dem unbearbeiteten MNIST-Datensatz eine sehr hohe Accuracy aufweist und dann immer weiter an die eben genannten Parametervorgaben angenähert.

1.1 MNIST-Daten Vorbereitung

Um dem Netz das Lernen der Daten zu erleichtern und klarere Strukturen zu erkennen, wurden alle Graustufen aus dem Datensatz eliminiert. Mit einer definierten Schwelle wurde für jedes Pixel entschieden, ob es schwarz oder weiß ist (`Listing 1.1`). Diese Binarisierungstechnik trägt dazu bei, das Rauschen zu minimieren und die Daten zu vereinfachen.

Listing 1.1: Binarisierung der Bilddaten mit einem Schwellenwert

```
    threshold = 127
    x_train_binary = (x_train > threshold).astype('float32
'')
    x_test_binary = (x_test > threshold).astype('float32')

```

Es existieren nur noch 0 und 1 Werte die deutlich einfacher interpretiert werden können. Zudem reduziert es die Datenmenge deutlich und ein und hilft bei einer effektiveren Datenverarbeitung. Hier wurde die Datenmenge genau halbiert (siehe Code: vorher 376320000 Bytes, nachher 188160000 Bytes). Zusätzlich kann die klare Unterscheidung zwischen schwarz und weiß bei der Kantendetektierung helfen. Diese bilden den zweiten Bearbeitungsschritt.

Nach der Binärisierung wurde ein Sobel-Filter angewendet, der die Bilder nach Kanten untersucht. Er besteht aus zwei Kernels, die jeweils horizontal und vertikal nach Kanten suchen. Das macht insofern Sinn, da unser metrisches Zahlensystem viele dieser Kanten besitzt. Vereinfacht wird diese Suche durch die vorherige Binärisierung (`Listing 1.2`).

Listing 1.2: Anwendung des Sobel-Filters auf die binären Bilddaten

```
    x_train_edges = filters.sobel(x_train_binary)
    x_test_edges = filters.sobel(x_test_binary)

```

Schließlich werden die Daten mit einem Gauß-Filter bearbeitet, um die Kanten zu glätten und etwaige Treppenartefakte (die durch die Binärisierung entstehen können) an den Kanten zu minimieren (`Listing 1.3`).

Listing 1.3: Anwendung des Gauß-Filters auf die geglätteten Kanten

```
    sigma = 1.0  # Standardabweichung des Gauß-Filters
    x_train_smoothed = gaussian_filter(x_train_edges,
sigma=sigma)
```

```
3     x_test_smoothed = gaussian_filter(x_test_edges, sigma=
    sigma)
4
```

Letztlich wurde eine One-Hot-Codierung mit den Labeln durchgeführt, um dem Netz das Lernen zu erleichtern. Diese Methode wandelt die Labels in einen Vektor um, der nur Nullen enthält, außer für die repräsentative Klasse. Das ist besonders nützlich, da es sich um eine Klassifizierungsaufgabe handelt (`Listing 1.4`). Die Funktion `to_categorical` aus der `tensorflow.keras.utils`-Bibliothek wird verwendet, um Klassenlabels in eine sogenannte One-Hot-Kodierung umzuwandeln.

Listing 1.4: Durchführung der One-Hot-Codierung auf die Labels

```
1     # one-hot encoding of labels
2     y_train_one_hot = to_categorical(y_train)
3     y_test_one_hot = to_categorical(y_test)
4
```

Die Anzahl an Trainingsdaten wurden nicht reduziert, da es für das einhalten der oben genannten Vorgaben nicht nötig ist.

1.2 Training des Netzes

Zu Beginn wurde ein Netz aufgestellt, das vornehmlich eine hohe Genauigkeit aufweist, ohne die Menge an Parametern zu beachten. Die Architektur ist in `Listing 1.5` dargestellt.

Listing 1.5: Definition des ersten neuronalen Netzwerks (marvin1)

```
1     marvin1 = tf.keras.models.Sequential([
2     tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
    input_shape=(28, 28, 1)),
3     tf.keras.layers.MaxPooling2D((2, 2)),
```

```
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
    ])

```

Die Anzahl der Parameter, Labels oder die Berechnungszeiten sind hier nicht berücksichtigt worden. Dies soll folglich in der zweiten Version des Netzes optimiert werden.

Das optimierte Netz startet mit dem Layer

Listing 1.6: Optimierter erster Layer des Netzes

```
tf.keras.layers.Conv2D(16, (3, 3), activation=tf.keras.
 layers.LeakyReLU(alpha=0.03), input_shape=(28, 28, 1))
```

und hat nur noch 16 Merkmalerkennungen. Die Leaky ReLU-Funktion wird verwendet, um früh vermeintlich falsch abgestrafte Merkmale nicht 'sterben' zu lassen. Das wird in der zweiten Schicht (vgl. 1.7) ebenso verwendet, aber bereits mit einem reduzierten Faktor, da die relevanten Merkmale hier schon deutlich mehr ausgeprägt sein sollten. Ab dem dritten Convolutional-Layer wird die normale ReLU als Aktivierungsfunktion genutzt.

Um die Gesamtanzahl der Parameter zu verringern, wird die Filteranzahl der Convolutional-Layer reduziert. Zudem wird die Kernel-Dimension angepasst, um genauere Ergebnisse zu erzielen. Es wird ein Feature Detector von (6, 1) verwendet, um horizontale Strukturen zu erkennen, und im späteren Layer ein (1, 6) Feature Detector, um vertikale Merkmale zu erkennen. Das deckt sich mit der Idee, die Kanten zu finden, die im Voraus im Datensatz durch den Sobel-Filter markiert wurden (`Listing 1.7`). Die `MaxPooling2D`-Layer werden verwendet um die räumliche Dimension zu reduzieren

und damit die Berechnungskosten zu senken. Die `Flatten`-Schicht bereitet die Daten für den `Dense`-Layer vor. Die Dense-Schicht mit 16 Neuronen hat die Aufgabe, abstrakte Merkmale aus den vorherigen Schichten zu gewinnen. Durch die Anwendung der ReLU-Aktivierungsfunktion werden nicht-lineare Aspekte betont, wobei negative Werte unterdrückt werden. Ziel ist es, eine repräsentative Daten-Darstellung zu erzeugen, während die Parameteranzahl moderat gehalten wird. Die Dropout-Schicht wird hinzugefügt, um Overfitting zu verhindern. Sie deaktiviert zufällig 2,5% der Neuronen während des Trainings, was dazu beiträgt, dass das Modell nicht zu sehr auf spezifische Muster der Trainingsdaten angewiesen ist. Dies verbessert die Generalisierungsfähigkeit des Modells. Die letzte Dense-Schicht mit 10 Neuronen entspricht den 10 Klassen des MNIST-Datensatzes (0 bis 9). Durch die Anwendung der Softmax-Aktivierungsfunktion wird die Ausgabe in Wahrscheinlichkeiten für jede Klasse umgewandelt. Das Neuron mit der höchsten Wahrscheinlichkeit wird dann als die vom Modell vorhergesagte Klasse interpretiert. Diese entscheidende Schicht führt die endgültige Klassifikation durch und bildet den Output des neuronalen Netzwerks.

Listing 1.7: Optimiertes Netz marvin2

```
1    marvin2 = tf.keras.models.Sequential([
2    tf.keras.layers.Conv2D(16, (3, 3), activation=tf.keras
   .layers.LeakyReLU(alpha=0.03), input_shape=(28, 28, 1)),
3    tf.keras.layers.MaxPooling2D((2, 2)),
4    tf.keras.layers.Conv2D(2, (6, 1), activation=tf.keras.
   layers.LeakyReLU(alpha=0.01)),
5    tf.keras.layers.MaxPooling2D((2, 2)),
6    tf.keras.layers.Conv2D(4, (1, 6), activation='relu'),
7    tf.keras.layers.Flatten(),
8    tf.keras.layers.Dense(16, activation='relu'),  #
   Anpassung der Dense-Schicht
9    tf.keras.layers.Dropout(0.025),  # Dropout-Schicht
   hinzugefügt
10   tf.keras.layers.Dense(10, activation='softmax')
11   ])
12
```

Die Entscheidung für 6 Epochen und einen Batch-Size von 256 wurde getroffen, um eine angemessene Balance zwischen Trainingseffizienz und Modellleistung zu gewährleisten (vgl. 1.8). Der Einsatz von TensorBoard bietet darüber hinaus eine effektive Möglichkeit, den Trainingsverlauf zu analysieren und etwaige Muster oder Probleme zu identifizieren.

Listing 1.8: Training des Modells mit TensorBoard-Rückruf

```
marvin2.fit(
x_train,
y_train_one_hot,
epochs=6,
batch_size=256,
validation_data=(x_test, y_test_one_hot),
callbacks=[tensorboard_callback]
)
```

1.3 Loss und Metriken

Um die performance des Modells zu bewerten wurden mehrere Metriken verwendet.

Auf den Testdatensatz liefert das Modell eine Accuracy von 92,36%. Damit ist die zu Beginn gesetzte Vorgabe erreicht. Um des weiteren die Echtheit dieser Aussage zu überprüfen werden die Übereinstimmungen mit der Kappa-Statistik überprüft. Sie filtert zufällige Übereinstimmungen heraus und liefert einen Wert von -1 bis 1 als Rückgabewert. 1 steht dabei für die perfekte Übereinstimmung. Der vom Modell erreichte Kappa-Wert liegt bei 0.915 und kann als sehr gute bis perfekte Übereinstimmung interpretiert werden. Auch die Präzision, der Recall und der F1 Score des Modells sind sehr gut.

Zudem wird eine Confusion Matrix ausgegeben, die die wahren Label mit den vorhergesagten Labeln gegenüberstellt. Zu erkennen ist, dass Zahlen die bei undeutlicher

Schrift sehr ähnliche Formen aufweisen am häufigsten verwechselt werden. So wird die tatsächliche Zahl 7 oft als 2 oder 9 interpretiert. Auch die im Datensatz mit 9 gelabelte Zahl wird verhältnismäßig oft als 4 oder 7 klassifiziert.

1.4 Ergebnisse und Bewertung

Aufgrund der Metriken bewertet sich das Modell angesichts der minimalen Trainingszeit und Parameter als äußerst erfolgreich. Verbesserungsmöglichkeiten ergeben sich jedoch bei der genaueren Unterscheidung von Zahlverwechslungen, insbesondere bei ähnlichen Formen wie 7, 2 und 9. Eine potenzielle Optimierung, beispielsweise durch die Integration eines Filters zur Erkennung runder Strukturen, wurde getestet (`Zoom`-Funktion die die bilder auf doppelte breite streckt um "Löcher"besser zu erkennen), jedoch stellte sich die damit verbundene längere Vorverarbeitungszeit als unpraktikabel heraus. Hiermit könnte die Genauigkeit noch verbessert werden, allerdings ist fraglich inwiefern die zu Beginn gesetzten Anforderungen dann noch eingehaten werden können.

Genauere Analysedaten können über den Tensorflow Codeblock im Quellcode eingesehen werden.