DeepLearning

Tim Lucas Halt

15. Dezember 2023

Sehr kurz

1 Methodik

Ziel

Mein Ziel war in einen der drei vorlesungsbegleitende Benchmarks zu führen und die Anstrengungen vorerst auf dieses Ziel zu legen. Dafür habe ich mich in der ersten Woche in alle drei Richtungen, maximale Genauigkeit, minimale Labels und minimale Parameter probiert. Zielerreicht wurde mit dem Netzt welches in Abschnitt 2 vorgestellt wurde. Eine kombination von verschiedensten quellen zusammengedrafen und mit etwas glück auf 99,47 % gebracht Die weitere Verbesserung wollte ich vor allem automatisieren. Dafür habe ich Kostenlose Grafikkarten von Kaggle (30 Stunden die Woche) und Colab (12 Stunden am Tag) genutzt. Im ersten Schritt wollte ich die Wahl der Hyperparemter meines optimieren. dafür habe ich verschiedene Kombinationen mit weights & biases sweepen und tracken lassen

2 Basis -

Das Ausgangsnetz, von dem die Betrachtungen ausgehen ist in Listing 1 dargestellt. Es hat etwa 62-tausend Parameter. Kompiliert mit dem Adam Optimizer, einer Learning-Rate von 0.003 und einem Training mit dem vollständigen Trainingsdatensatz (60000 Label) bei einer Batch-Size von 512 über 100 Epochen erreichte es 99,47% Genauigkeit.

Listing 1: Python-Code

```
1   InputLayer(input_shape=(28,28,1)),
2   Conv2D(filter=28, kernel_size=5, padding='same', activation='relu'),
3   MaxPooling2D(2,2),
4   Conv2D(filter=16, kernel_size=5, padding='same', activation='relu'),
5   MaxPooling2D(2,2),
6   layers.Dropout(0.2),
7   layers.Flatten(),
8   layers.Dropout(0.2),
9   layers.Dense(64, kernel_regularizer = tf.keras.regularizers.l2(0.07),
      activation = 'relu'),
10  GaussianNoise(0.1),
11  Dense(10, activation='softmax')
```

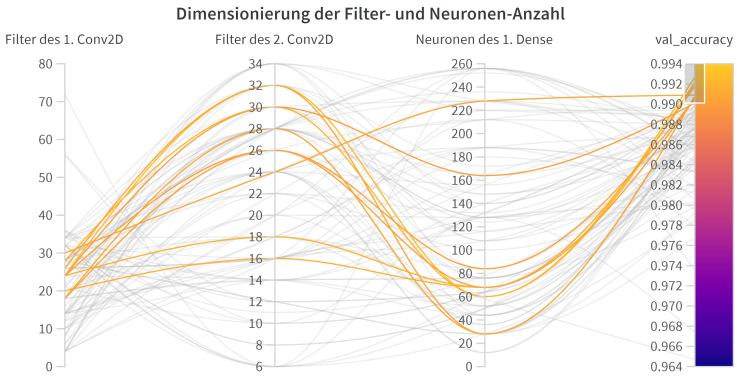

Abbildung 1: Etwa 100 Kombinationen aus Filter- und Neuronen-Anzahlen bewertet nach der val_accuracy. Dokumentiert mit Weight & Biases

3 Weights and Biases

Um den Einfluss der Hyperparameter teilautomatisiert zu testen, wird die Sweep-Funktion von Weights & Biases verwendet (https://docs.wandb.ai/guides/sweeps. Dabei werden die Hyperparameter aus einem vorgegebenen Raster gewählt, ein Training durchgeführt und die Ergebnisse dokumentiert. Als Methode ist *bayes*, die Bayes'sche Optimierung, zur Maximierung der val_acc gewählt.

3.1 Dimensionierung der Filter- und Neuronen-Anzahl

Zunächst erfolgte die Untersuchung der Filter-Anzahlen in den Convolution-Layern und der Neuronen-Anzahl des ersten Dense-Layer. Das Output-Dense-Layer wird bei 10 Neuronen und der *softmax*-Aktivierungsfunktion belassen, damit jeder Klasse eine Wahrscheinlichkeit zugeordnet wird.
Das Raster möglicher Hyperparameter wurde so gewählt, dass (1) eine Abstufung im Convolution-Teil erfolgt, (2) auch Zahlen nicht zur Basis zwei getestet werden und (3) die Parameteranzahl nicht exorbitant groß wird. Das erste Kriterium erwies sich als kontraproduktiv und wird später noch verworfen.
Die Ergebnisse von circa 100 Kombinationen sind in Abbildung 1 dargestellt. Hervorgehoben sind die besten Zehn. Die val_acc ist hoch, für eine Filteranzahl der Convolution-Layers um den Wert 28. Für die Neuronen des ersten Dense-Layers konzentrieren sich der erfolgreichen Kombinationen bei den kleineren Werten. Nach einem weiter eingegrenztem Sweep sind 28 Filter in beiden Convolution-Layern als gewählt und 54 Neuronen für das Dense-Layer. Das Symmetrie-Bedürfnis erkennt Ähnlichkei-

ten zur Bildgröße von 28x28 Pixeln und wird gleichzeitig enttäuscht, weil 54 weder eine 2er-Potenz noch ein Vielfaches von 28 ist.

3.2 Wahl der Aktivierungsfunktionen.

Für die Wahl der Aktivierungsfunktionen wurden ebenfalls Sweeps durchgeführt. Die Ergebnisse sind in Abbildung 2 abgebildet und die besten fünf hervorgehoben. Sie haben die Sigmoid-Funktion im Dense-Layer gemeinsam. Bei den Convolution-Layern zeigte über alle Kombinationen hinweg die Relu-Funktion die stärkste Tendenz zu höher Genauigkeit. Entsprechend wurde gewählt: Relu, Relu, Sigmoid.

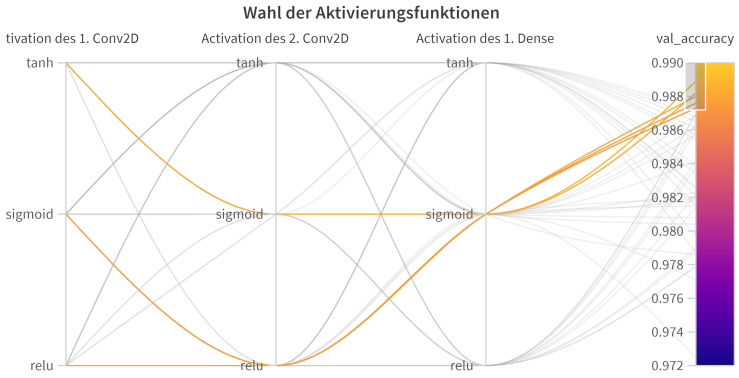

Abbildung 2: Kombinationen der Aktivierungsfunktionen bewertet nach der val_accuracy. Dokumentiert mit Weight & Biases

3.3 BatchNormalization, Dropout und GaussianNoise

Als drittes wird das Netz zum einen um BatchNormalization vor den MaxPool und Dropout erweitert, weil hiermit konsequent bessere Ergebnisse erzielt wurden.
Die Dropout- und GaussianNoise-Werte wurden ebenfalls mittels Weigts & Biasses getestet. Als erfolgreich Erwiesen sich Dropouts von 0,4 und GaussianNoise von 0.75. . Daraus ergibt sich das in **??** dargestellte Netz mit 95488 Parametern.

Listing 2: Python-Code

```
1   tf.keras.layers.InputLayer(input_shape=(28,28,1)),
2   tf.keras.layers.Conv2D(filters=28, kernel_size=5, padding='same',
      activation='relu'),
3   tf.keras.layers.BatchNormalization(),
4
```

```
5   tf.keras.layers.MaxPooling2D(2,2),
6   tf.keras.layers.Dropout(0.4),
7
8   tf.keras.layers.Conv2D(filters=28, kernel_size=5, padding='same',
      activation='relu'),
9   tf.keras.layers.BatchNormalization(),
10
11  tf.keras.layers.MaxPooling2D(2,2),
12  tf.keras.layers.Dropout(0.4),
13
14  tf.keras.layers.Flatten(),
15
16  tf.keras.layers.Dense(54, kernel_regularizer = tf.keras.regularizers.
      l2(0.07), activation = 'sigmoid'),
17  tf.keras.layers.BatchNormalization(),
18
19  tf.keras.layers.Dropout(0.4),
20  tf.keras.layers.GaussianNoise(0.75),
21
22  tf.keras.layers.Dense(10, activation='softmax')
```

3.4 Callbacks

Hinsichtlich des angestrebten Benchmarks sollte die Genauigkeit noch weiter gesteigert werden. Dafür wird der Trainingsprozess mit Callbacks angereichert.

3.4.1 Variable Learning Rate

Zum einen wird EarlyStopping eingesetzt, um den automatisierten Trainingsprozess zu verkürzen und wenig erfolgreiche Druchläufe abzubrechen. Gleichzeitig werden die besten Ergebnisse gespeichert.

er = tf.keras.callbacks.EarlyStopping(monitor="val$_accuracy$", $patience = 15, restore_best_weights =$ $True$)

3.4.2 Variable Learning Rate

Die be

Listing 3: Variable Learning-Rate

```
vlr = tf.keras.callbacks.ReduceLROnPlateau(monitor='loss', factor =
  0.95, patience=10, min_lr=0.0001)
vlr2 = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.995
  ** x)
```

er = tf.keras.callbacks.EarlyStopping(monitor="val$_accuracy$", $patience = 15, restore_best_weights = True$)

datagen = tf.keras.preprocessing.image.ImageDataGenerator(rotation$_range = 15$, $zoom_range = 0.15$, $shear_range = 0.1$, $width_shift_range = 0.1$, $height_shift_range = 0.1$, $rescale = 0$, $fill_mode =' nearest'$, $horizontal_flip = False$, $vertical_flip = False$)$datagen.fit(x_train)$

3.5 Netzanpassung

24C5 means a convolution layer with 24 feature maps using a 5x5 filter and stride 1 24C5S2 means a convolution layer with 24 feature maps using a 5x5 filter and stride 2 P2 means max pooling using 2x2 filter and stride 2 256 means fully connected dense layer with 256 units

784 - [32C3-32C3-32C5S2] - [64C3-64C3-64C5S2] - 128 - 10

4 batchsize

während des testens 256 sinnvoll gleicher erfolg auch mit 32 aber mehr streuung und hinsichtlich banchmark peaks gewünscht

5 callbacks

5.1

sonstige versuche: adam mit sgd ersetzen erfolglos

6 Variable Learning Rate

Listing 4: Variable Learning-Rate

```
vlr = tf.keras.callbacks.ReduceLROnPlateau(monitor='loss', factor =
  0.95, patience=10, min_lr=0.0001)
vlr2 = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.995
  ** x)
```

7 Optimierung

Chat GPT: Wenn du zwei aufeinanderfolgende Convolutional Layers mit einer Filtergröße von 3x3 verwendest, wird das Netzwerk in der Lage sein, komplexere und nichtlineare Muster zu erlernen. Hier sind ein paar Gründe, warum das der Fall ist:

Receptive-Field-Erweiterung: Die Verwendung von zwei Convolutional Layers mit einer Filtergröße von 3x3 führt zu einem größeren Receptive Field im Vergleich zu einer einzelnen Schicht mit 5x5 Filtergröße. Dadurch kann das Netzwerk über einen größeren Bereich des Eingabebildes Informationen erfassen.

Mehrere Nichtlinearitäten: Jede Convolutional Layer wird durch eine nichtlineare Aktivierungsfunktion wie ReLU aktiviert. Wenn zwei 3x3 Convolutional Layers hintereinander geschaltet werden, werden zwischen den Schichten zwei Aktivierungsfunktionen angewendet, was zu einer stärkeren nichtlinearen Transformation der Eingabedaten führt.

Parameterreduktion: Zwei 3x3 Convolutional Layers haben insgesamt weniger Parameter als eine 5x5 Convolutional Layer. Weniger Parameter bedeuten weniger Berechnungen und weniger Anfälligkeit für Overfitting, da weniger spezifische Features gelernt werden müssen.

Lernen von tieferen Merkmalen: Das Stapeln von mehreren Convolutional Layers hintereinander ermöglicht es dem Netzwerk, hierarchische und abstrakte Merkmale zu lernen. Durch die schrittweise Verfeinerung der Merkmale von einer Schicht zur nächsten können komplexere Repräsentationen des Eingaberaums erzeugt werden.

Dieses Konzept der Verwendung von kleineren Filtern nacheinander anstelle eines einzelnen großen Filters wird oft in modernen Convolutional Neural Networks verwendet, um die Lernfähigkeit und Effizienz zu verbessern. Es trägt dazu bei, dass das Netzwerk tiefergehende und komplexere Muster im Eingaberaum erfassen kann.

https://doi.org/10.48550/arXiv.2202.01653

Chat GPT: Strided Convolution für Downsampling: Bei Verwendung einer Convolutional Layer mit Strides von 2 werden die Ausgabefeature-Maps aufgrund des größeren Schrittweitenwerts um den Faktor 2 in jeder Dimension reduziert. Das führt zu einer räumlichen Reduzierung der Feature-Maps und erzeugt somit einen Downsampling-Effekt ähnlich dem einer Max-Pooling-Schicht.

Trainierbare Downsampling-Operation: Im Gegensatz zur Max-Pooling-Schicht, die eine festgelegte Operation (Maximum aus einem Fenster) ohne trainierbare Parameter ist, ermöglicht die Verwendung einer Convolutional Layer mit Strides von 2 das Lernen von Downsampling-Operationen. Die Gewichte in der Convolutional Layer werden während des Trainings angepasst, um die beste Darstellung der Daten zu finden, während gleichzeitig eine Reduzierung der räumlichen Dimensionen erfolgt.

End-to-End-Lernen: Durch die Verwendung einer Convolutional Layer für Downsampling wird das Lernen von Merkmalen und Downsampling in einem End-to-End-Modell ermöglicht. Das Netzwerk lernt die geeigneten Filter, um Merkmale zu extrahieren, während es gleichzeitig die räumliche Dimension reduziert, ohne auf separate Schichten wie Max-Pooling zurückgreifen zu müssen.

Flexibilität und Anpassungsfähigkeit: Die Convolutional Layer mit Strides von 2 bietet Flexibilität in Bezug auf die Art des gelernten Downsampling. Das Netzwerk kann verschiedene Arten von Downsampling lernen, die möglicherweise besser zur spezifischen Datenverarbeitung passen als die festgelegte Max-Pooling-Operation.

Die Verwendung einer Strided Convolution für Downsampling bietet somit die Möglichkeit, eine lernbare Downsampling-Operation zu integrieren, die besser in das Gesamtkonzept des neuronalen Netzes eingebunden werden kann und es dem Netzwerk ermöglicht, sowohl Merkmale zu lernen als auch die Dimensionen der Feature-Maps zu reduzieren.

8 Finale - Mit Sinn und Verständnis

9 Optimierung -

Mal schauen was so kommt

Listing 5: Python-Code

```
from scipy.spatial import distance

x_train_best = []  # Array für die ähnlichsten Bilder
y_train_best = []  # Array für die zugehörigen Labels
similar_indices = [47647]  # Array für die Indizes der ähnlichsten Bilder

# Berechnung der ähnlichsten Ziffern für jede Klasse von 0 bis 9
for digit in range(10):

print('Digit:', digit)
# Filtern der Ziffern nach ihrer Klasse
class_images = x_train[y_train == digit]

# Berechnung der durchschnittlichen Cosinus-Ähnlichkeit für jede Ziffer zu anderen Ziffern derselben Klasse
similarities = []
for i, image in enumerate(class_images):
avg_similarity = 0
for other_image in class_images:
if not np.array_equal(image, other_image):
# Umwandlung von 28x28 Bildern in Vektoren für Cosinus-Ähnlichkeit
image_vector = image.flatten()
other_image_vector = other_image.flatten()
# Berechnung der Cosinus-Ähnlichkeit
cosine_similarity = 1 - distance.cosine(image_vector, other_image_vector)
avg_similarity += cosine_similarity
avg_similarity /= len(class_images) - 1  # Durchschnittliche Ähnlichkeit zu allen anderen Ziffern der Klasse außer sich selbst
similarities.append((i, avg_similarity))

# Sortieren nach der durchschnittlichen Ähnlichkeit und Auswahl der ähnlichsten Ziffer
similarities.sort(key=lambda x: x[1], reverse=True)
most_similar_index = similarities[0][0]

most_similar_index_train_images = np.where((y_train == digit))[0][most_similar_index]
most_similar_digit = x_train[most_similar_index_train_images]

print('Index:', most_similar_index_train_images)

# Hinzufügen des ähnlichsten Bildes, seines Labels und seines Index im train_images Array in den Arrays
x_train_best.append(most_similar_digit)
y_train_best.append(digit)
similar_indices.append(most_similar_index_train_images)
```

```
42
43  # Umwandeln der Listen in numpy arrays
44  x_train_best = np.array(x_train_best)
45  y_train_best = np.array(y_train_best)
46  similar_indices = np.array(similar_indices)
```

10 Dropout

Paper: Dropout: A Simple Way to Prevent Neural Networks from Overfitting Nitish Srivastava nitish@cs.toronto.edu Geoffrey Hinton hinton@cs.toronto.edu Alex Krizhevsky kriz@cs.toronto.edu Ilya Sutskever ilya@cs.toronto.edu Ruslan Salakhutdinov

11 Sonst noch

cosinus nähste

Autoencoder

12 Teil 2 - NLP

Gandalf in Moria: „Im Zweifelsfall sollte man immer seiner Nase folgen . . .“