

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 Netz welches in Abschnitt 2 vorgestellt wurde. Eine kombination von verschiedensten quellen zusammengedrahten 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')
```

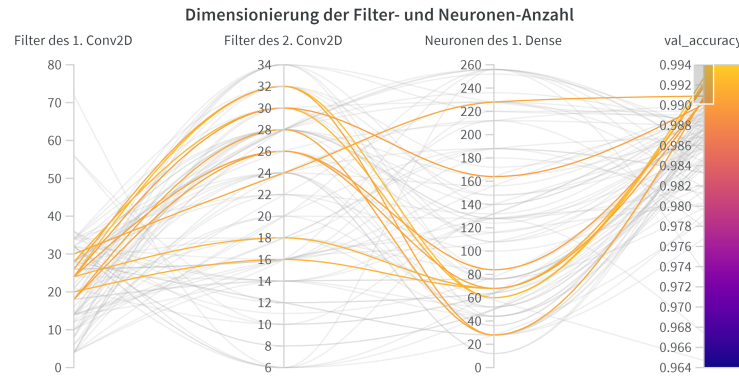


Abbildung 1: Etwa 100 Kombinationen aus Filter- und Neuronen-Anzahlen bewertet nach der `val_accuracy`. Dokumentiert mit Weight & Biases

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 Ähnlichkeit

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.

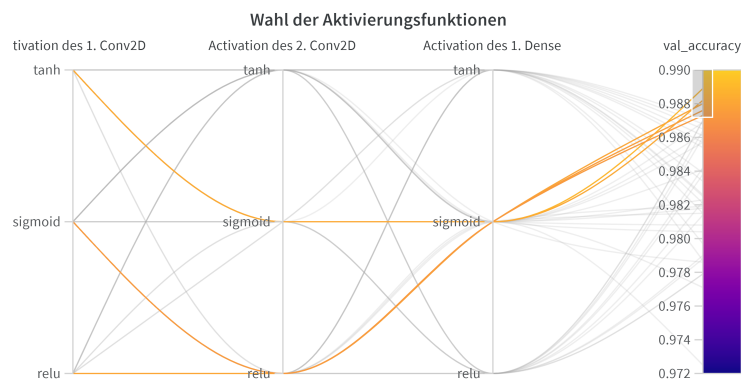


Abbildung 2: Kombinationen der Aktivierungsfunktionen bewertet nach der val_accuracy. Dokumentiert mit Weight & Biases

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. **papers (dropout gege noverfitting)**. Daraus ergibt sich das in Listing 2 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',
9         activation='relu'),
10    tf.keras.layers.BatchNormalization(),
11
12    tf.keras.layers.MaxPooling2D(2,2),
13    tf.keras.layers.Dropout(0.4),
14
15    tf.keras.layers.Flatten(),
16
17    tf.keras.layers.Dense(54, kernel_regularizer = tf.keras.regularizers.
18        l2(0.07), activation = 'sigmoid'),
19    tf.keras.layers.BatchNormalization(),
20
21    tf.keras.layers.Dropout(0.4),
22    tf.keras.layers.GaussianNoise(0.75),
23
24    tf.keras.layers.Dense(10, activation='softmax')

```

3.4 Callbacks - Early Stopping und Variable Learning Rate

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

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

Listing 3: Early Stopping

```

1     er = tf.keras.callbacks.EarlyStopping(
2         monitor="val_accuracy",
3         patience=10,
4         restore_best_weights=True
5     )

```

Um darüber hinaus schneller den Grenzwert zu erreichen wird eine Variable Learning Rate verwendet. In Kombination mit dem EarlyStopping zeigte eine gleichmäßige Reduktion mit jeder Epoche besser Ergebnisse, als die Methode ReduceLROnPlateau. In Kombination mit einer Batch Size von 32 ergaben sich die besten Ergebnisse

Listing 4: Variable Learning-Rate

```

1  vlr = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.995
    ** x)

```

3.5 Data Augmentation

Des Weiteren wird die Anzahl an Trainingsdaten mit einer Augmentation erweitert. Dazu kommt der ImageDataGenerator von Keras zum Einsatz. Für Rotation und Zoom werden 0,15 verwendet. Für Shear und Shift 0,1. Die Bilder werden nicht gespiegelt, weil die Zahlen nicht entsprechend symmetrisch sind. Die gewählten Werte haben sich in Weight & Biases Sweeps als geeignet herausgestellt

Listing 5: Data Augmentation

```

1  datagen = tf.keras.preprocessing.image.ImageDataGenerator(
2      rotation_range = 15,
3      zoom_range = 0.15,
4      shear_range = 0.1,
5      width_shift_range = 0.1,
6      height_shift_range = 0.1,
7      rescale = 0,
8      fill_mode = 'nearest',
9      horizontal_flip=False,
10     vertical_flip=False)
11     datagen.fit(x_train)

```

3.6 Netzanpassung

Nachdem die vorausgegangenen Methoden wieder stagnierten. Erfolgte eine Anpassung des Netzes.

Zum einen werden zwei aufeinanderfolgende Convolutional Layers mit einer Filtergröße von 3x3 verwendet, anstelle eines mit 5x5. Dadurch ist das Netzwerk in der Lage sein, komplexere und nichtlineare Muster zu erlernen. <https://doi.org/10.48550/arXiv.2202.01653>

Strided Convolution für Downsampling: Bei Verwendung einer Convolutional Layer mit Strides von 2 werden die Ausgabefeature-Maps aufgrund des größeren Schrittwertenwerts um den Faktor 2 in jeder Dimension reduziert. 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.

die filter anzahl im 2. conv wurde wie angekündigt erhöht auf 56

Zusätzlich wurde auch noch ein Dense layer hinzugefügt mit 82 Neuronen (also 28 mehr als das 54 Neuronen Netz, für das Symmetriebedürfnis)

und nach jedem dropout ein Gaussian Noise gegen Overfitting eingefügt

4 Zusammenfassungen

Mit diesen Änderungen wurde bei 50 Netzen... maximal 99.77 und damit der Benchmark

5 Sonstige Versuche

Neben dieser ausführlich dokumentierten Optimierung hinsichtlich des Accuracy-Benchmarks wurden an den folgenden Punkten ausprobiert..

5.1 Reduktion der Labels

Listing 6: Python-Code

```
1  from scipy.spatial import distance
2
3  x_train_best = [] # Array für die ähnlichsten Bilder
4  y_train_best = [] # Array für die zugehörigen Labels
5  similar_indices = [47647] # Array für die Indizes der ähnlichsten
    Bilder
6
7  # Berechnung der ähnlichsten Ziffern für jede Klasse von 0 bis 9
8  for digit in range(10):
9
```



```

10  print('Digit:', digit)
11  # Filtern der Ziffern nach ihrer Klasse
12  class_images = x_train[y_train == digit]
13
14  # Berechnung der durchschnittlichen Cosinus-Ähnlichkeit für jede
    Ziffer zu anderen Ziffern derselben Klasse
15  similarities = []
16  for i, image in enumerate(class_images):
17      avg_similarity = 0
18      for other_image in class_images:
19          if not np.array_equal(image, other_image):
20              # Umwandlung von 28x28 Bildern in Vektoren für Cosinus-Ähnlichkeit
21              image_vector = image.flatten()
22              other_image_vector = other_image.flatten()
23              # Berechnung der Cosinus-Ähnlichkeit
24              cosine_similarity = 1 - distance.cosine(image_vector,
                other_image_vector)
25              avg_similarity += cosine_similarity
26              avg_similarity /= len(class_images) - 1 # Durchschnittliche Ä
                hnlichkeit zu allen anderen Ziffern der Klasse außer sich selbst
27              similarities.append((i, avg_similarity))
28
29  # Sortieren nach der durchschnittlichen Ähnlichkeit und Auswahl der ä
                hnlichsten Ziffer
30  similarities.sort(key=lambda x: x[1], reverse=True)
31  most_similar_index = similarities[0][0]
32
33  most_similar_index_train_images = np.where((y_train == digit))[0][
    most_similar_index]
34  most_similar_digit = x_train[most_similar_index_train_images]
35
36  print('Index:', most_similar_index_train_images)
37
38  # Hinzufügen des ähnlichsten Bildes, seines Labels und seines Index im
    train_images Array in den Arrays
39  x_train_best.append(most_similar_digit)
40  y_train_best.append(digit)
41  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)

```

5.2 NLP - Tiny Schiller

tiny schiller referenz

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

Abschließend ein Zitat des *drunkenSchiller* Models