Automatische Bildklassifizierung mithilfe von Convolutional Neural Networks

Facharbeit Informatik

Tim D.

17.05.2019

Zusammenfassung

Diese Facharbeit beschäftigt sich mit der Bildklassifizierung mithilfe von *Convolutional Neural Networks*. Die Zielsetzung ist unter anderem die Entwicklung eines solchen Netzes zur Klassifizierung von Zeichnungen. Nach einer kurzen Einführung wird außerdem in der Theorie die allgemeine Funktionsweise von *Convolutional Neural Networks* erläutert. Dabei liegt der Fokus auf den Besonderheiten dieser Art von Netz im Vergleich zu gewöhnlichen *Multi Layer Perceptrons*. Grundlegendes Wissen über die Funktionsweise dieser ist daher Vorraussetzung für das Verständnis dieser Facharbeit.

Convolutional Neural Networks sind der Stand der Technik und Forschung im Bereich der Bildklassifizierung. Keine andere Technik erzielt so gute Ergebnisse wie *Convolutional Neural Networks*. Obwohl deren Erfindung bereits mehrere Jahrzehnte zurückliegt, sind sie immer noch Gegenstand der Forschung. Mit allgemein zunehmender Rechenleistung können außerdem immer größere Datenmengen verarbeitet werden, sodass sie wohl auch in Zukunft noch eine bedeutende Rolle spielen werden. Die zunehmende Relevanz von *Convolutional Neural Networks* war der Hauptgrund für die Wahl des Themas dieser Facharbeit.

Inhaltsverzeichnis

1 Einführung **4**
- 1.1 Problematik bei der Bilderkennung mit einem Multi Layer Perceptron . . . 4
- 1.2 Entstehung der Convolutional Neural Networks . . . 4
- 1.3 Verwendungszwecke . . . 5

2 Anwendungsbeispiel **6**
- 2.1 Datensatz Google QuickDraw . . . 6
- 2.2 Softwarebibliotheken TensorFlow/Keras . . . 7

3 Netzarchitektur und Training **8**
- 3.1 Convolutional Layer . . . 8
- 3.2 Pooling Layer . . . 9
- 3.3 Dense Layer . . . 10
- 3.4 Training . . . 11

4 Evaluierung **12**
- 4.1 Probleme . . . 13
- 4.2 Fazit . . . 14

5 Anhang **15**
- 5.1 Vollständiger Code . . . 15
- 5.2 Graphische Darstellung der Architektur . . . 17
- 5.3 Vergleiche von Architekturen . . . 18

Literatur **22**

Erklärung über selbständige Anfertigung **24**

1 | Einführung

1.1 Problematik bei der Bilderkennung mit einem Multi Layer Perceptron

Ein Neuronales Netzwerk für monochrome Bilder mit 128×128 Pixeln würde $128 \times 128 = 16384$ Eingabedaten benötigen. Für farbige Bilder mit drei Farbkanälen (RGB) würde sich diese Zahl verdreifachen ($784 \times 3 = 49152$). Bei einem *Multi Layer Perceptron* sind in der Regel alle Eingabedaten mit jeweils mehreren Neuronen verbunden, sodass die erste versteckte Schicht in diesem Beispiel bereits ein Vielfaches von 49152 Neuronen hätte. Die große Zahl an Neuronen in solch einem *Multi Layer Perceptron* hätte einen hohen Berechnungsaufwand zur Folge, der bei großen Datenmengen, insbesondere bei Bildern mit höherer Auflösung, nicht mehr von einem gewöhnlichen Computer gestemmt werden könnte.

1.2 Entstehung der Convolutional Neural Networks

Um dieses Problem (s. 1.1) zu lösen, wurden die sogenannten *Convolutional Neural Networks* als Erweiterung für *Multi Layer Perceptrons* entwickelt.

Die grundlegende Funktionsweise von *Convolutional Neural Networks* basiert auf den Forschungsergebnissen von *Hubel* und *Wiesel*, die 1959 in einem Experiment mit Katzen feststellten, dass die Neuronen im visuellen Kortex von Säugetieren hierarchisch strukturiert sind. Die ersten Neuronen reagieren auf einfache Formen und Muster, während die Neuronen danach auf komplexere Formen ansprechen, indem die Aktivitäten der ersten Neuronen kombiniert werden. [1]

Diese Arbeit war die Grundlage für *Neocognitron*, ein Neuronales Netzwerk, das 1980 von dem japanischen Forscher *Fukushima* entwickelt wurde und diese hierarchische Struktur modelliert. Der erstmalige Einsatz von soge-

nannten Kerneln (s. 3.1.1) ermöglichte es, die Anzahl der benötigen Neuronen stark zu reduzieren. [2]

Das erste moderne *Convolutional Neural Network* stammt von dem französischen Informatiker *LeCun*, der 1989 ein Neuronales Netzwerk vorstellte, das mithilfe von *Backpropagation* lernen konnte. Das Netzwerk war in der Lage handschriftliche Postleitzahlen mit einer Fehlerquote von weniger als 1% zu lesen und wurde zu diesem Zweck vom *Unites States Postal Service* (*USPS*) verwendet. [3]

1.3 Verwendungszwecke

Convolutional Neural Networks finden in erster Linie Verwendung bei der Verarbeitung von Bilddaten. Der primäre Einsatzzweck ist die Bildklassifizierung, die auch Gegenstand dieser Facharbeit ist. Des Weiteren kommen sie auch bei der Objekterkennung, also der Lokalisierung von Objekten in Bildern, zum Einsatz. So ist es beispielsweise mit *SSD* (*Single Shot MultiBox Detector*) möglich in Echtzeit Objekte wie Personen oder Straßenschilder zu erkennen. [4]

Darüber hinaus werden *Convolutional Neural Networks* unter anderem auch bei der Posenbestimmung (*Pose Estimation*), der Texterkennung (*OCR*), dem Erkennen von Aktivitäten (*Action Recognition*) sowie in der Computerlinguistik verwendet. [5]

2 | Anwendungsbeispiel

Ziel dieser Facharbeit war die Entwicklung eines *Convolutional Neural Networks*, das in der Lage ist Zeichnungen zu klassifizieren.

2.1 Datensatz Google QuickDraw

QuickDraw[1] ist ein Spiel des *Google Creative Labs*, bei dem ein Neuronales Netz versucht Zeichnungen (*"doodles"*) zu erkennen, die von Spielern in jeweils 20 Sekunden erstellt werden. So sind inzwischen insgesamt 50 Millionen Zeichnungen aus 345 Kategorien von über 15 Millionen Spielern entstanden, die *Google* frei und öffentlich zur Verfügung gestellt hat.

Abbildung 2.1: Beispielbilder der Klasse "Katze"

Das Neuronale Netz, das im Rahmen dieser Facharbeit entstanden ist, verwendet eine Auswahl von jeweils 12000 Zeichnungen aus vier Kategorien bzw. Klassen (Katze, Hund, Apfel und Banane).

[1] https://quickdraw.withgoogle.com/

2.1.1 Datenformat

Die Bilddaten liegen als *Numpy*[2] *Arrays* vor. Das *Array* mit Eingabedaten für das Neuronale Netz hat die Größe $48000 \times 28 \times 28 \times 1$ (Bildanzahl $\times$ Höhe $\times$ Breite $\times$ Farbkanäle). Die Pixelwerte sind Graustufen zwischen 0 (schwarz) und 1 (weiß). Jedem Bild in den Eingabedaten wird die zugehörige Klasse (s. 2.1) zugeordnet. Diese ist jeweils mithilfe eines *Arrays* mit vier Elementen, von denen jedes für eine Klasse steht, kodiert. Das Element mit der zugehörigen Klasse hat den Wert 1 während die anderen drei Elemente jeweils den Wert 0 haben. Diese Art der Kodierung von Klassen heißt *One Hot Encoding*. Damit hat das *Array* mit den Ausgabedaten die Größe 48000×4.

Die *Arrays* lassen sich in *Python* wie folgt laden:

```
x = numpy.load('x.npy') # Eingabedaten
y = numpy.load('y.npy') # Ausgabedaten
```

2.2 Softwarebibliotheken TensorFlow/Keras

TensorFlow[3] ist eine vom *Google Brain Team* (zuerst für interne Zwecke) entwickelte, quelloffene Softwarebibliothek für unter anderem Maschinelles Lernen in der Programmiersprache *Python*.

In dieser Facharbeit wird die höhere Schnittstelle *Keras*[4] benutzt, die insbesondere auf Nutzerfreundlichkeit ausgelegt ist. *TensorFlow* enthält eine eigene Implementierung von *Keras* um die Benutzung von *TensorFlow* zu vereinfachen.

2.2.1 Sequential Model

Keras bietet das sogenannte *Sequential model* für Neuronale Netzwerke, bei denen die Schichten einen linearen Stapel bilden, das heißt in einer Sequenz aufeinander folgen.

```
model = Sequential()
```

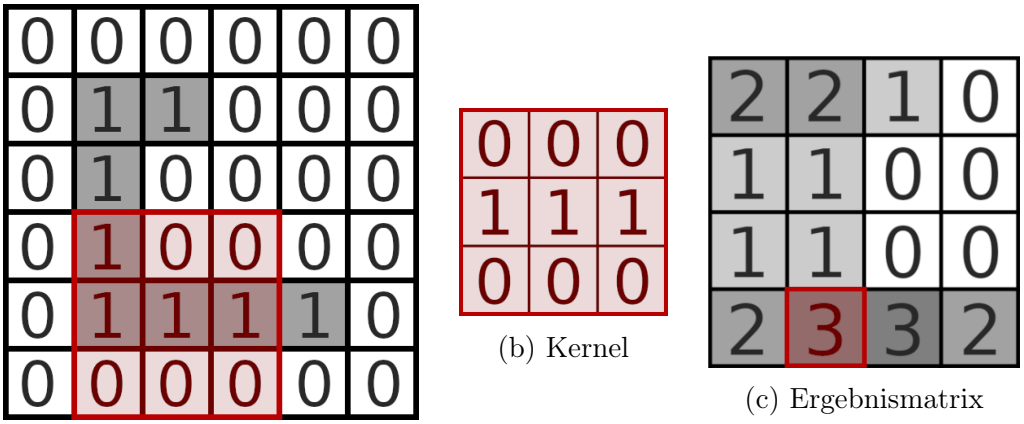
[2] https://www.numpy.org/
[3] https://www.tensorflow.org/
[4] https://keras.io/

3 | Netzarchitektur und Training

3.1 Convolutional Layer

3.1.1 Kernel

Ein *Kernel* (oft auch Filter oder Faltungsmatrix) ist eine kleine, meist quadratische, Matrix. Nimmt man aus einem Bild einen Teilbereich (repräsentiert als Matrix aus Pixelwerten), der genau so groß wie der *Kernel* ist, lässt sich aus diesen beiden Matrizen das Skalarprodukt bilden. Der Kernel wird dabei schrittweise über das gesamte Eingabebild bewegt, sodass sich für jeden Teilbereich des Bildes ein Wert in der Ergebnismatrix ergibt.



(a) Eingabematrix

(b) Kernel

(c) Ergebnismatrix

Abbildung 3.1: Beispiel eines Kernels

Abbildung 3.1 zeigt ein Beispiel einer Eingabematrix (Abbildung 3.1a) und eines *Kernels* (Abbildung 3.1b). Aus dem Kernel wird mit dem rot makierten Teilbereich der Eingabematrix das Skalarprodukt gebildet.

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = 3$$

Dieser *Kernel* ist in der Lage, horizontale Linien zu erkennen. Jeder Teilbereich, der von solch einer Horizontalen durchzogen ist, ergibt in der Ergebnismatrix (Abbildung 3.1c) eine 3 an der entsprechenden Stelle.

Ein *Convolutional Layer* besteht aus mehreren dieser Kernel, sodass mehrere Ergebnismatrizen entstehen. Die Anzahl der *Kernel*, sowie deren Größe sind Teil der Architektur des Netzes. Die Werte in den *Kerneln* werden dagegen während des Trainingsprozesses bestimmt.

In *Keras* lassen sich *Convolutional Layer* folgendermaßen erstellen:

```
model.add(Conv2D(32, 5, activation="relu", input_shape=(28, 28, 1)))
model.add(Conv2D(16, 3, activation="relu"))
```

Dieser Code fügt dem Netz zwei *Convolutional Layer* mit 32 *Kerneln* der Größe 5×5 beziehungsweise mit 16 *Kerneln* der Größe 3×3 hinzu. Die Wahl der Aktivierungsfunktion fiel auf *ReLU* (*Rectified Linear Unit*). Diese ist heutzutage die am meisten genutzte Aktivierungsfunktion in tiefen Neuronalen Netzen. [6] Allerdings konnten auch mit anderen gängigen Aktivierungsfunktionen ähnlich gute Ergebnisse erzielt werden (s. 5.3.1). Bei der ersten Schicht muss außerdem angegeben werden, welche Dimensionen die Eingabedaten haben (s. 2.1.1). Für alle weiteren Schichten inferiert *Keras* dies automatisch.

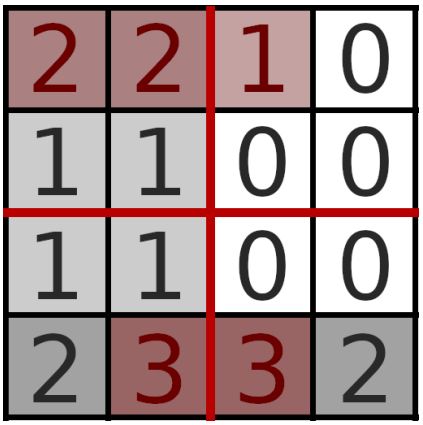
Shared Weights

Da ein *Kernel* für alle Teilbereiche der Eingabematrix gleich ist, werden deutlich weniger Parameter bzw. Gewichte benötigt als dies bei einem *Multi Layer Perceptron* der Fall wäre. Die einzelnen Bereiche "teilen" sich die Parameter, weshalb man diese als *Shared Weights* bezeichnet. Die reduzierte Anzahl an Parameter hat einen merklichen Effekt auf die Performance des Netzes beim Trainingsprozess.

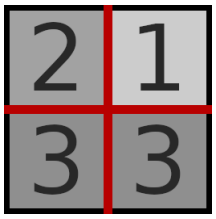
3.2 Pooling Layer

Auf einen oder mehrere *Convolutional Layer* folgt häufig ein *Pooling Layer*. Dieser dient dazu, die Größe der Ergebnismatrizen zu reduzieren (*Downsampling* / *Subsampling*) und so zum einen den Berechnungsaufwand zu verringern und zum anderen *Overfitting* zu verhindern.

Dazu wird ein quadratischer Bereich der Ergebnismatrix jeweils durch einen einzigen Wert repräsentiert. Meistens wird dafür der Maximalwert des entsprechenden Bereichs verwendet. Diese Art des *Poolings* heißt *Max Pooling*. Alternative Arten des *Poolings*, wie *Average Pooling*, haben sich in der Praxis als deutlich ungeeigneter erwiesen. In diesem Beispiel sorgte *Average Pooling* für nur minimal schlechtere Ergebnisse (s. 5.3.2).



(a) Ergebnismatrix des Convolutional Layers



(b) Ergebnismatrix des Pooling Layers

Abbildung 3.2: Max Pooling

Abbildung 3.2 zeigt die Anwendung von *Max Pooling* auf die Ergebnismatrix aus Abbildung 3.1. Die Matrix wird in vier Bereiche der Größe 2×2 aufgeteilt. Die rot hinterlegten Pixel in Abbildung 3.2a sind jeweils die Maximalwerte in diesen Bereichen, aus denen die Ergebnismatrix des *Pooling Layers* (Abbildung 3.2b) gebildet wird.

Dem Netz werden wie folgt zwei *Max Pooling Layer* hinzugefügt:

```
model.add(Conv2D(32, 5, activation="relu", input_shape=(28, 28, 1)))
model.add(MaxPooling2D())
model.add(Conv2D(16, 3, activation="relu"))
model.add(MaxPooling2D())
```

3.3 Dense Layer

Während die *Convolutional Layer* dazu dienen, bestimmte *Features* wie Kurven, Kanten oder Muster zu erkennen, werden am Ende des Netzes vollvermaschte Schichten (*Dense Layer*) benötigt, die für die eigentliche Klassifizierung zuständig sind:

```
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(4, activation="softmax"))
```

Ein vorangesteller *Flatten Layer* formt die dreidimensionalen Daten der vorhergenden Schichten in einen eindimensionalen Vektor um, der dann als Eingabe für die *Dense Layer* verwendet werden kann.

Der letzte *Dense Layer* stellt die Ausgabeschicht dar, weshalb er aus vier Neuronen (jeweils ein Neuron pro Klasse) besteht. Die Aktivierungsfunktion *Softmax* erzeugt einen Vektor mit vier Werten zwischen 0 und 1, sodass die Summe dieser Werte exakt 1 ergibt. Diese Werte lassen sich dann als Wahrscheinlichkeiten der Zugehörigkeit des Bildes zur jeweiligen Klasse interpretieren. Ein Bild, das als Ausgabe den unten stehenden Vektor erzeugt, in dem die Werte für die Klassen "Katze", "Hund", "Apfel" und "Banane" stehen, zeigt beispielsweise mit einer Wahrscheinlichkeit von 80% eine Katze.

$$\begin{pmatrix} 0.8 \\ 0.2 \\ 0.0 \\ 0.0 \end{pmatrix}$$

3.4 Training

Bevor der Trainingsprozess gestartet werden kann, muss das Netz kompiliert werden. Dazu muss angegeben werden, mit welchem Algorithmus das Netz optimiert werden soll. Ein häufiger genutzter Optimierungsalgorithmus ist *Adam*, der auch in diesem Beispiel für gute Ergebnisse gesorgt hat. [7] Als Fehlerfunktion wird hier *Categorical Cross-Entropy* verwendet, die üblicherweise zusammen mit *Softmax* als Aktivierungsfunktion (s. 3.3) bei Klassifizierungsproblemen eingesetzt wird.

```
model.compile(optimizer="adam", loss="categorical_crossentropy")
```

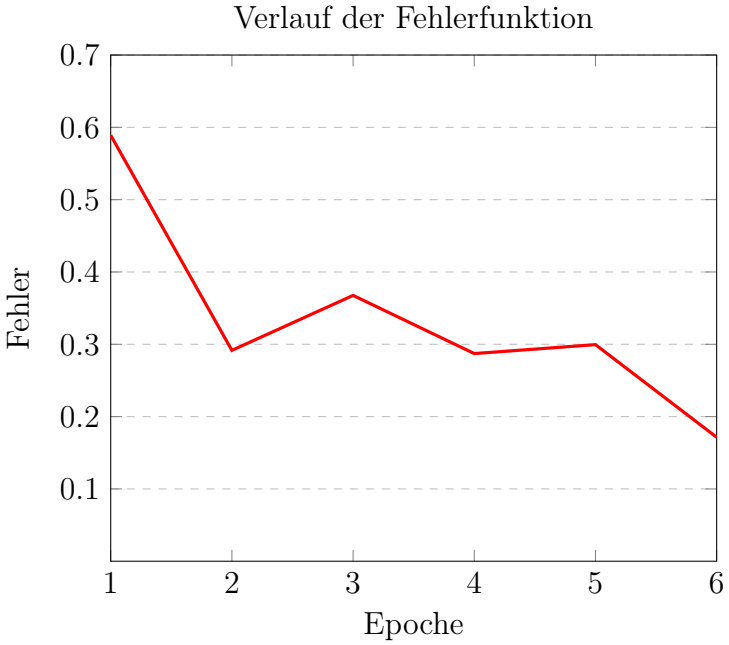
Daraufhin kann mit dem Training (*Fitting*) begonnen werden. Dazu muss neben den Ein- und Ausgabedaten die *Batch Size* und die Anzahl der Epochen angegeben werden. Außerdem bietet *Keras* eine einfache Möglichkeit, einen Teil (hier 20%) der Trainingsbeispiele als Testdaten zu verwenden.

```
model.fit(x, y, batch_size=100, epochs=6, validation_split=0.2)
```

4 | Evaluierung

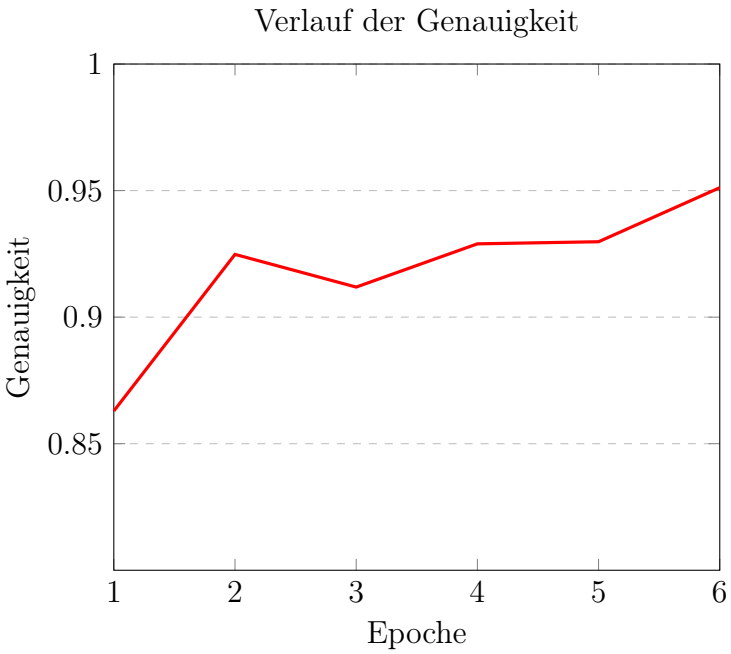
TensorFlow stellt mit *TensorBoard*[1] einige Tools zur Verfügung, mit denen man den Trainingsprozess verfolgen und visualisieren und so verschiedene Netzarchitekturen vergleichen kann. So lassen sich zum Beispiel auch die Verläufe der Fehlerfunktion und der Genauigkeit darstellen.

Beim Training des *Convolutional Neural Networks* ist die Fehlerfunktion folgendermaßen verlaufen:



Bereits nach der ersten Epoche befindet sich der Fehler deutlich unter 1.0 und wird im Laufe des Trainings auf etwa 0.17 optimiert. Dies ist der beste Wert, der mit dieser Netzarchitektur nach mehreren Trainingsversuchen erreicht werden konnte.

[1] https://github.com/tensorflow/tensorboard



Die Genauigkeit konnte auf über 0.95 optimiert werden, das heißt das Netz erkennt in 95% der Fälle die richtige Klasse.

Wichtig ist hierbei, dass diese Messwerte anhand der Trainingsdaten, also Bilder, mit denen das Netz nicht trainiert wurde, erhoben werden. Nur so kann sichergestellt werden, dass das Netz generalisiert hat anstatt lediglich die Trainingsdaten "auswendig" zu lernen (*Overfitting*).

Zu Testzwecken wurde auch eine graphische Zeichenoberfläche entwickelt, auf der der Nutzer eines der vier Motive zeichnen kann. Das Netz versucht dabei in Echtzeit die Eingabe zu klassifizieren. Auch hier ist das Ergebnis sehr zufriedenstellend.

4.1 Probleme

Die vorliegende Architektur weist zwei nennenswerte Probleme auf.

Das erste Problem hat sich beim Testen mit der o. g. graphischen Oberfläche gezeigt sobald der Nutzer ein Bild zeichnet, was zu keiner der vier Klassen gehört. Da das Netz ausschließlich die vier Klassen, mit denen es trainiert wurde, kennt, ist in solch einem Fall das Ergebnis unvorhersehbar. Auch die intuitive Erwartung, dass sich dann für alle Klassen jeweils eine Wahrscheinlichkeit von etwa 25% ergibt, wird nicht erfüllt. Das liegt in erster Linie an der Aktivierungsfunktion *Softmax*, die immer versucht einen Wert hervorzuheben, während die restlichen möglichst klein gehalten werden.

Daher erkennt das Netz in diesen Fällen scheinbar eindeutig eine Klasse. Um dieses Problem zu lösen, könnte man sich die Werte vor der Anwendung von *Softmax* anschauen.

Das zweite Problem bestand in der Erweiterbarkeit der Architektur. Beim Hinzufügen einer weiteren beliebigen Klasse, erkannte das Netz keine Bilder mehr richtig. Letztendlich stellte sich dies als Problem der Daten dar. Diese waren ursprünglich nicht zufällig angeordnet, sodass das letzte Fünftel der Daten ausschließlich aus Bildern der fünften Klasse bestand. Beim Trainieren wurde genau dieses letzte Fünftel als Testdaten verwendet (s. 3.4), wodurch das Netz lediglich mit Bildern der anderen Klassen trainiert und nur mit Bildern der fünften Klasse getestet wurde, was eine Genauigkeit von 0% zur Folge hatte. Eine Randomisierung der Daten konnte dieses Problem lösen.

4.2 Fazit

Convolutional Neural Networks sind eine effektive Technik zur Erkennung von Bildern. Mit wenig Code konnte ein sehr zufriedenstellendes Ergebnis erreicht werden. Allerdings ist das Trainieren mit hohem Rechenaufwand verbunden, der mangels guter Hardware und trotz der Beschränkung auf vier Klassen und vergleichsweise kleine Bilder viel Zeit in Anspruch genommen hat. Komplexere Architekturen mit mehr Klassen und hochauflösenden Bildern lassen sich heute noch nicht effizient mit gewöhnlicher Hardware trainieren. Dies erschwert das Finden einer geeigneten Architektur enorm. Ständige Verbesserungen von Software und Hardware sowie Fortschritte in der Forschung lassen jedoch eine positive Entwicklung diesbezüglich erwarten, sodass *Convolutional Neural Networks* in Zukunft wahrscheinlich immer häufiger zum Einsatz kommen und noch bessere Ergebnisse liefern werden.

5 | Anhang

5.1 Vollständiger Code

5.1.1 Convolutional Neural Network

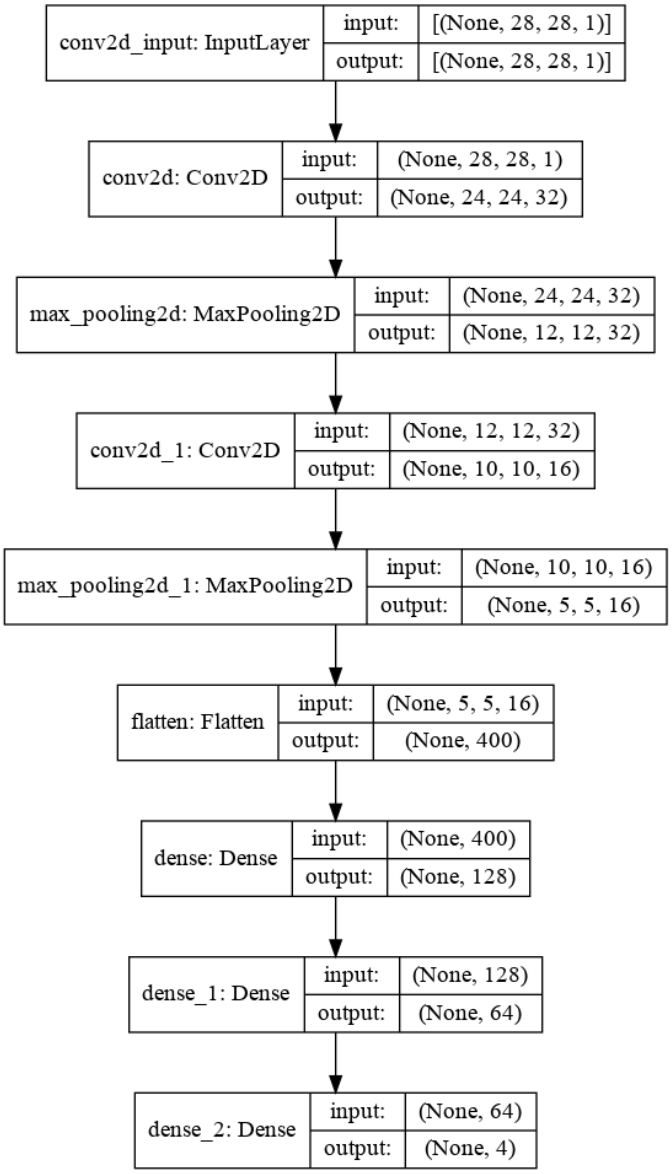
```
1  import numpy
2  from tensorflow.keras.layers import Conv2D, Dense, Flatten,
3    MaxPooling2D
4  from tensorflow.keras.models import Sequential
5
6  x = numpy.load('x.npy')
7  y = numpy.load('y.npy')
8
9  model = Sequential()
10
11 model.add(Conv2D(32, 5, activation="relu", input_shape=(28, 28, 1)))
12 model.add(MaxPooling2D())
13 model.add(Conv2D(16, 3, activation="relu"))
14 model.add(MaxPooling2D())
15
16 model.add(Flatten())
17 model.add(Dense(128, activation="relu"))
18 model.add(Dense(64, activation="relu"))
19 model.add(Dense(4, activation="softmax"))
20
21 model.compile(optimizer="adam", loss="categorical_crossentropy")
22 model.fit(x, y, batch_size=100, epochs=6, validation_split=0.2)
```

5.1.2 Programm zur Vorverarbeitung der Daten

```
1  import numpy
2  from tensorflow.keras.utils import to_categorical
3
4  files = ['cat.npy', 'dog.npy', 'apple.npy', 'banana.npy']
5
6  x = numpy.concatenate([numpy.load(f)[:12000] for f in files]) / 255.0
7  y = numpy.concatenate([numpy.full(120000, i) for i in range(4)])
8
9  perm = numpy.random.permutation(len(x))
10 x = x[perm]
11 y = y[perm]
12
13 x = x.reshape(x.shape[0], 28, 28, 1).astype('float32')
14 y = to_categorical(y)
15
16 numpy.save('./x.npy', x)
17 numpy.save('./y.npy', y)
```

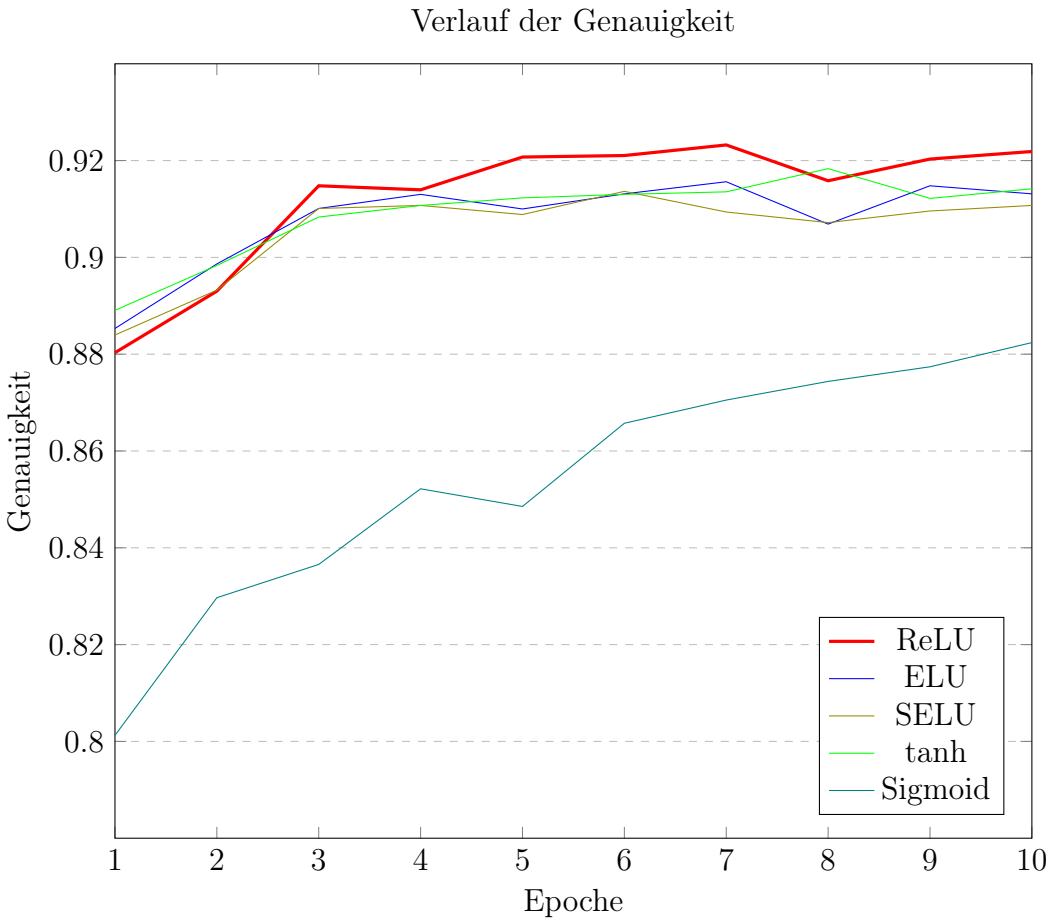
5.2 Graphische Darstellung der Architektur

```
1 from tensorflow.keras.utils import plot_model
2 plot_model(model, show_shapes=True)
```

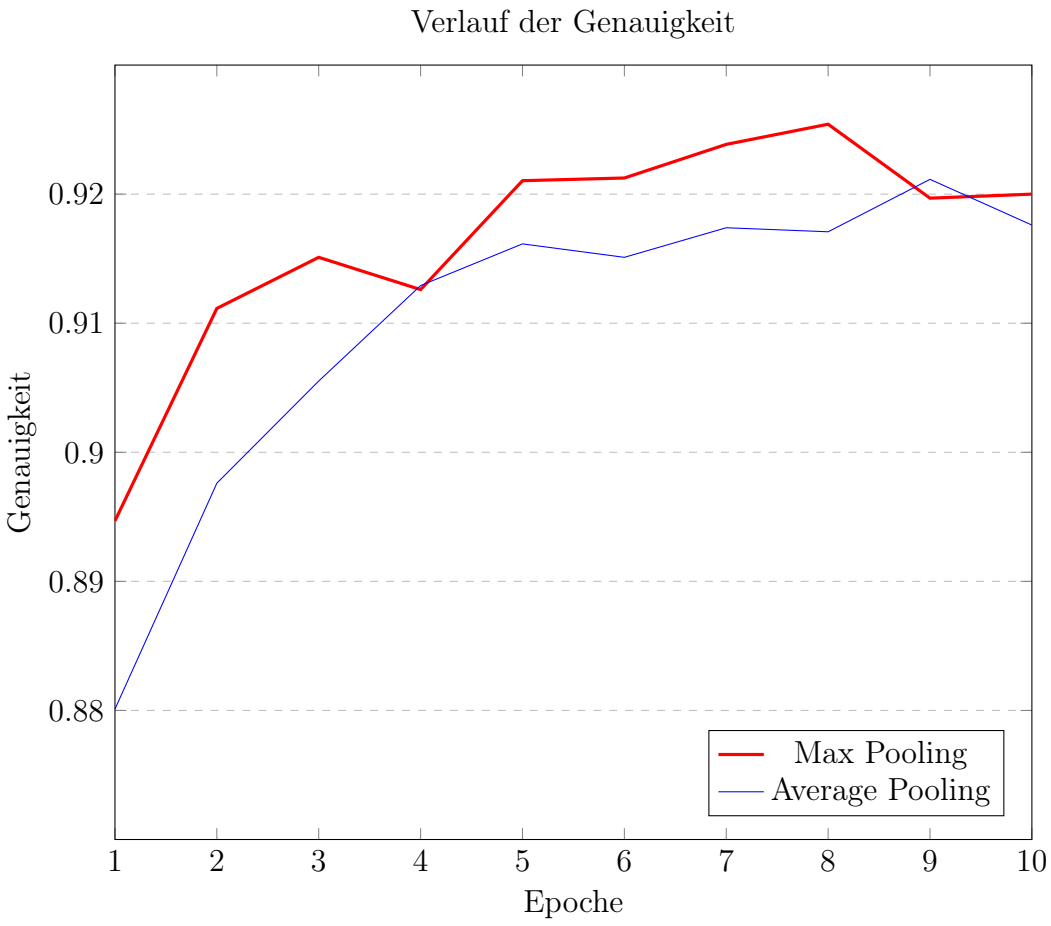


5.3 Vergleiche von Architekturen

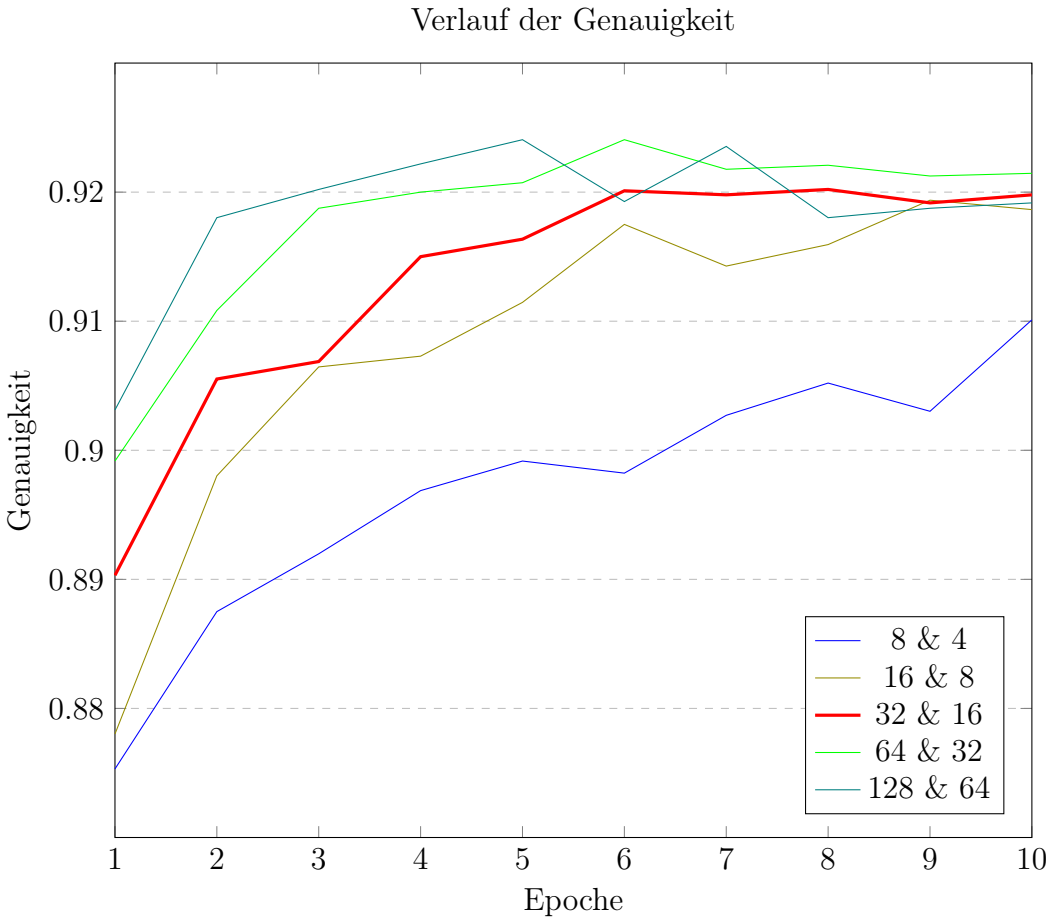
5.3.1 Aktivierungsfunktion



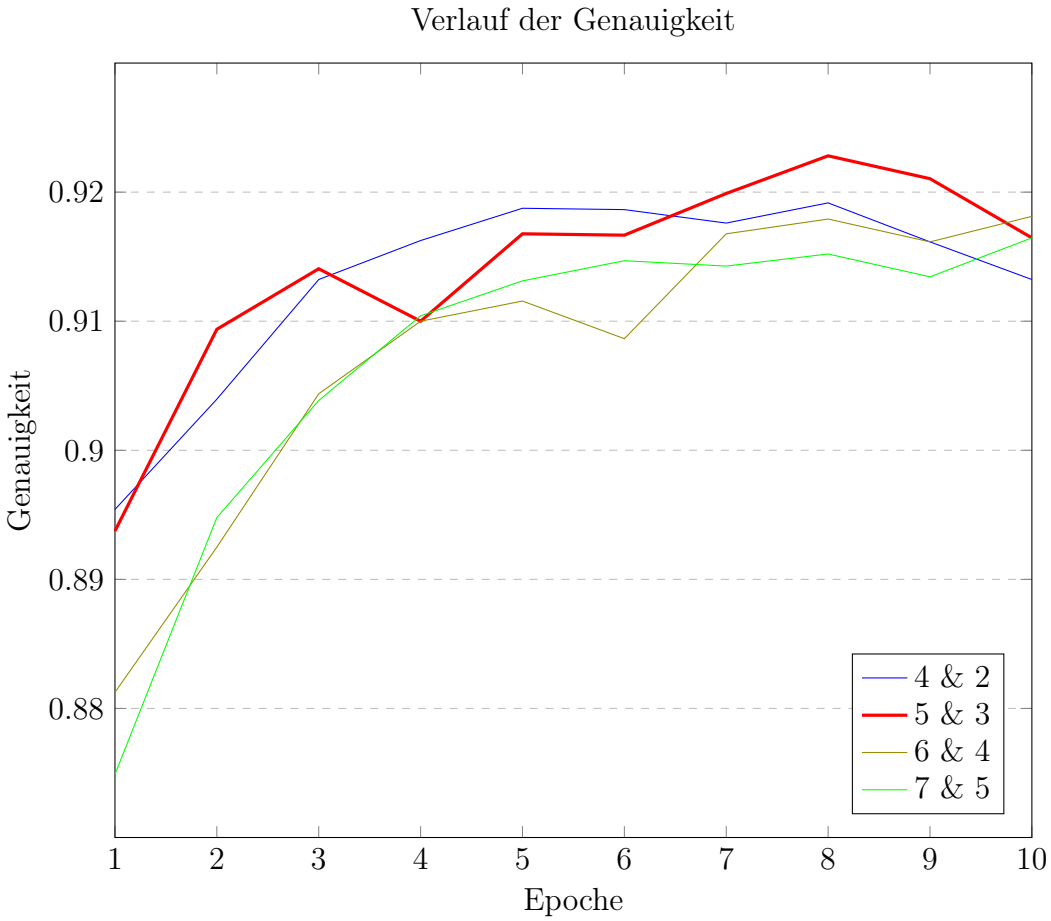
5.3.2 Max Pooling und Average Pooling



5.3.3 Kernelanzahl



5.3.4 Kernelgröße



Literatur

[1] D. H. Hubel und T. N. Wiesel. „Receptive fields of single neurones in the cat's striate cortex“. In: *The Journal of Physiology* (1959). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/ (besucht am 17. 05. 2019).

[2] K. Fukushima. „Neocognitron. A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position“. In: *Biological Cybernetics* (1980). URL: https://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf (besucht am 17. 05. 2019).

[3] Y. LeCun u. a. „Backpropagation Applied to Handwritten Zip Code Recognition“. In: *Neural Computation* (1989). URL: http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf (besucht am 17. 05. 2019).

[4] W. Liu u. a. „SSD: Single Shot MultiBox Detector“. In: *CoRR* (2016). URL: http://arxiv.org/abs/1512.02325 (besucht am 17. 05. 2019).

[5] J. Gu u. a. „Recent Advances in Convolutional Neural Networks“. In: *CoRR* ([2015] 2017). URL: http://arxiv.org/abs/1512.07108 (besucht am 17. 05. 2019).

[6] P. Ramachandran, B. Zoph und Q. V. Le. „Searching for Activation Functions“. In: *CoRR* (2017). URL: http://arxiv.org/abs/1710.05941 (besucht am 17. 05. 2019).

[7] D. P. Kingma und J. Ba. „Adam: A Method for Stochastic Optimization“. In: *CoRR* (2015). URL: https://arxiv.org/abs/1412.6980 (besucht am 17. 05. 2019).

Erklärung über selbständige Anfertigung

Ich erkläre hiermit, dass ich diese Facharbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

17.05.2019

Tim D.