Schriftzeichenerkennung mit Machine Learning in Python

Skriptsprachen Projekt FH SWF 2019/2020

Inhaltsverzeichnis

Einleitung **3**

Projektbeschreibung **4**

Problembeschreibung 4

Lösungsansatz 4

Ergebnisse **8**

Quellen **9**

Werkzeuge **9**

Kontaktinfo **9**

Erklärung zur Eigenständigen Ausarbeitung **10**

Einleitung

Das Projekt zielt darauf ab ein Programm zu schreiben, das handlich geschriebene Schriftzeichen auf Bildern (schwarz/weiß) erkennt und den Zeichencode auf der Konsole ausgibt. Der Zeichen-Standard variiert eventuell je nach trainierten Modell.
Das Endziel soll sein, japanische Schriftzeichen (Kanji, Katakana und Hiragana) erkennen können.

Da es so viele komplexere japanische Schriftzeichen - die Kanji - gibt, werden einfachheitshalber, um das Modell zu testen, erstmal Zeichen aus dem lateinischen Alphabet, arabische Ziffern, einige Satzzeichen und die Katakana trainiert.
Diese Auswahl kommt aus der ersten Datenbank (ETL1 aus ETLCDB, siehe [1.1]), welche uns bereitgestellt wurden.

In diesem Projekt müssen wir uns mit Neuronalen Netzwerken auseinander setzen und den passenden Algorithmus für unser Problem finden, beziehungsweise ein passendes Modell bauen. Spezifisch werden wir uns mit CNNs (Convolutional Neural Networks) befassen, da diese unseren Erkenntnissen nach für derartig komplexe Probleme bessere Ergebnisse liefern und eine bessere Laufzeit zu besitzen scheinen.
Als Sprache für das Projekt wurde entsprechend des Moduls Skriptsprachen Python gewählt. Als Bibliothek wurde Tensorflows Keras ausgewählt, es findet allerdings auch SciKit-Learn minimale Anwendung.

Projektbeschreibung

Problembeschreibung

Ein Computer ist nicht in der Lage mit einfachen Algorithmen aus Bildern Informationen zu extrahieren (mit guter Laufzeit). Das Problem ist, dass die Menschliche Schreibweise von Person zu Person unterschiedlich ist, obwohl dieselben Zeichen geschrieben werden. Außerdem kann in einem Bild, die gleiche Information an unterschiedlichen Positionen stecken.

Lösungsansatz

Deshalb verwenden wir ein Convolutional Neural Network (CNN), das Gehirnstrukturen aus lebenden Organismen simuliert.
Machine Learning kann generell in drei Bereiche eingeteilt werden: Supervised Learning, Unsupervised Learning und Reinforcement Learning. Für unsere Ausgabe macht Supervised Learning, also überwachtes Lernen am meisten Sinn, da wir bereits beschriftete (labeled) Daten haben. Desweiteren ist ein eifriger (eager) Algorithmus sinnvoll, da das Modell dazu dienen soll, direkt akkurate Ergebnisse zu liefern, auch bei wenig input von Nutzern.

Ein CNN betrachtet die Pixel eines Bildes und lernt über viele Iterationen und Lernschritte eine Gewichtung auf seine Auswertung anzuwenden, um relevante “Features” bzw. Merkmale herauszufiltern und anhand dieser auf ein Ergebnis zu kommen. In unseren Projekt verwenden wir nur Schwarz-Weiß Bilder, was das Unterscheiden zwischen Hintergrund und Schriftzeichen einfacher macht. In den meisten Fällen würden Farben vermutlich auch keine relevanten Informationen enthalten und unsere Datenbank enthält nunmal auch nur Bilder mit einem Farbkanal.

CNNs basieren auf den folgenden vier Komponenten:

- Convolution
- Activation Function (e.g. ReLU, Softmax)
- Pooling
- Full Connectedness

Convolution

Convolution passiert in vier Schritten:

1. Ein zweidimensionaler Filter wird Schritt für Schritt über das Bild geschoben
2. Der Filter enthält Gewichtungen, welche mit den derzeit angeschauten Pixeln multipliziert werden (jede Position des Filters hat eine eigene Gewichtung)
3. Die Produkte von jedem Pixel werden aufsummiert
4. Die Summe wird durch die Anzahl der betrachteten Pixel (entspricht Filtergröße) geteilt

Das Ergebnis wird danach in eine Ergebnismatrix geschrieben, dessen Größe von der Größe des Filters und der Eingabe (dem Bild) abhängt, jedoch durch Padding, also Auffüllen mit 0, des Bildes beeinflusst werden kann. Das ermöglicht dem Neuronalen Netzwerk zu überprüfen, ob das bekannte Muster im Bild ist.

Activation Function

Ein Convolution Layer ruft normalerweise nach Berechnung der Ergebnismatrix eine Activation Function auf. Diese ändert nochmals die Gewichtung des Ergebnisses. Der Grund dafür ist, dass man nach starken, ausschlaggebenden Features sucht. So kann man z.B. per ReLU alle Werte, welche negativ und somit für uns irrelevant sind, auf 0 setzen und alle positiven Werte und 0 werden nicht geändert. Dies verhindert ungewollte Änderungen durch kommende Filter.

$$f(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$

Ein anderes Beispiel ist die Sigmoidfunktion, welche alle Werte in den Bereich $[0,1] \in \mathbb{R}$ bringt.

Pooling

Pooling funktioniert ähnlich wie Convolution mit Filtern. Wie ein Filter wird ein Fenster über die Matrix geschoben, die Prozedur ist jedoch anders. Aus allen Werten im Fenster werden neue Werte berechnet, jedoch werden keine Gewichte angewandt. Üblich ist es den maximalen Wert (Max Pooling) oder den Durchschnitt (Mean Pooling) aller Werte aus dem Fenster zu nehmen.

Full Connectedness

Als letzten Schritt bevor man ein Ergebnis erhalten kann, wird eine normale vollständig vernetzte Schicht eingefügt, welche anhand der Eingabewerte ein Ergebnis liefern kann. Diese

Keras bietet selbstverständlich alle notwendigen Werkzeuge, um ein CNN zu bauen.

Der erste Ansatz war ein Netzwerk zu erstellen, was ganz simpel ein zweidimensionales Bild (2D-Tensor) in ein eindimensionales Layer zu übertragen und auf dieses das ein bis zwei „Dense“-Layer, also vollständig verbundene Schichten, anzuwenden. Wie erwartet war das Ergebnis nicht besonders gut mit einer maximalen Präzision von rund 20% für das Training alleine.

Im nächsten Schritt wurde ein 2D Convolution-Layer hinzugefügt. Als Filter wurden einige Dimensionen ausprobiert, zuerst 5x5, dann 3x3, 10x10 und 8x8. Die besten Ergebnisse lieferte 5x5, danach der 8x8 Filter. Gewählt wurde 5x5, weil die Zeichen etwa immer die halbe Höhe und Breite des Bildes einnehmen und es schien, als würden die relevanten Informationen etwa diesen Bereich im Bild einnehmen. Danach wurde systematisch ausprobiert, jedoch tat sich zwischen 5x5 und 8x8 nicht viel. Als Dimensionalität der Ergebnismatrix wurden Werte zwischen 5 und 32 probiert, erreicht durch same-Padding.
Es wurden sowohl ReLU als auch softmax als Activation Functions probiert, wobei ReLU keinen besonderen Effekt haben sollte. Die Unterschiede waren minimal, jedoch hat softmax generell die besseren Ergebnisse geliefert.
Die Ergebnisse waren gut mit etwa 60%.

In der Hoffnung eine höhere Genauigkeit zu erzielen wurde Max Pooling and verschiedenen Stellen im Modell probiert und ein weiteres 2D Convolution Layer hinzugefügt.
Beispielsweise wurde Folgendes probiert:

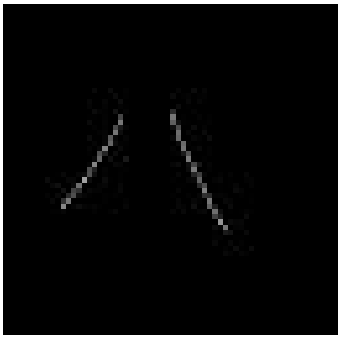
```
model = Sequential([
    Conv2D(32,(5,5), padding='same', activation='softmax',
        input_shape=(network._IMG_HEIGHT,network._IMG_WIDTH,network._IMG_CHANNELS)),
    MaxPooling2D((8, 8)),
    Conv2D(64, (5, 5), padding='same', activation='softmax'),
    MaxPooling2D((3, 3)),
    Flatten(),
    Dense(97, activation='softmax')
])
```

```
model = Sequential([
    Conv2D(32,(8,8), padding='same', activation='softmax',
        input_shape=(network._IMG_HEIGHT,network._IMG_WIDTH,network._IMG_CHANNELS)),
    MaxPooling2D((5,5)),
    Conv2D(64, (5, 5), padding='same', activation='softmax'),
    Flatten(),
    Dense(97, activation='softmax')
])
```

Jedoch ließ sich durch die zweite Schicht mit Convolution nur 80% erreichen. Max Pooling hat auch wenig bis gar nichts bewirkt.
Der Grund für die Auswahl des Max Pooling über Mean Pooling war, dass Störfaktoren im Bild leichter ignoriert werden können. Die trainierten bilder enthielten nun auch nur Weiß auf Schwarz mit ein wenig „noise“ im Bild. Dieser Störfaktor wurde von uns amplifiziert, um die Varianz zwischen den Bildern ein wenig zu erhöhen und ein realistisches Input darzustellen, was auch Overfitting reduzieren sollte. Die Bilder waren recht dunkel, wodurch die Erhöhung der Helligkeit sinnvoll schien.

Overfitting schien nie ein Problem zu sein, da die Validierungspräzision nur anstieg.

Nach einigen weiteren Tests fiel auf, dass die derzeitige Genauigkeit über viele Wege zu erreichen ist. Unter anderem ganz simpel nur ein einziges Convolution-Layer mit einer 64x64 großen Ergebnismatrix zu verwenden und Max Pooling auf diese anzuwenden. Daraufhin wurden kurze Tests mit eigens erstellten Bildern durchgeführt. Bei diesen Tests kam raus, dass die Genauigkeit fürchterlich war. Die Ergebnisse waren falsch und Treffergenauigkeiten des Modells lagen unter 50%, größtenteils um 10%.
Beispiel für eigene Testdaten im Vergleich zu den Bildern aus ETL1:




Daraus wurde klar, dass Overfitting eventuell doch ein großes Problem ist. Aufgefallen ist dies dadurch, dass das Overfitting ein Problem zu werden schien, selbst bei erhöhter Anzahl Epochen beim Training. Schließlich wird eine große Menge an sehr ähnliches Bildern mit relativ wenigen Merkmalen trainiert und mit sehr ähnlichen Bildern validiert. Ein nächster Schritt wäre zu versuchen noch mehr Varianz in die Daten reinzubringen. Man könnte Bilder drehen, eine große Menge eigene hinzufügen oder unterschiedliche Mengen an Störfaktoren einführen. Eine andere Methode, Overfitting zu vermeiden ist es, einen gewissen Teil der Trainingsdaten zufällig in der letzten Schicht ausscheiden zu lassen, was hier jedoch vermutlich keinen größeren Effekt haben würde. Leider sind diese Änderungen innerhalb der Zeitbegrenzung nicht möglich gewesen.

Es besteht jedoch die Möglichkeit, dass das derzeitige oder ein ähnliches Modell auf den deutlich komplexeren Kanji besser funktionieren würde. Der Grund dafür ist, dass Kanji meistens deutlich mehr lernbare Merkmale aufweisen.
Ein Beispiel: 練 besteht aus zwei Teilen, 糸 und 東, welche wiederum aus Einzelteilen bestehen. Allein durch die Vielzahl der Striche sollten handgeschriebene Zeichen größere Unterschiede aufweisen.

Ergebnisse

Das derzeitige Modell sieht folgendermaßen aus:

```
self.model = Sequential([
    Conv2D(64,(5,5), padding='same', activation='relu',
        input_shape=(self._IMG_HEIGHT, self._IMG_WIDTH, self._IMG_CHANNELS)),
    MaxPooling2D((8, 8)),
    Flatten(),
    Dense(97, activation='softmax')
])

self.model.compile(loss='categorical_crossentropy',
                   optimizer='sgd',
                   metrics=['accuracy'])
```

```
Epoch 15/15
1546/1546 [==============================] - 42s 27ms/step - loss: 0.6180 - accuracy: 0.8488 - val_loss: 0.6281 - val_accuracy: 0.8437
Training took 1e+01min 13seconds
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 63, 64, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 7, 8, 32)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 8, 64)          51264
_________________________________________________________________
flatten (Flatten)            (None, 3584)              0
_________________________________________________________________
dense (Dense)                (None, 97)                347745
=================================================================
Total params: 399,841
Trainable params: 399,841
Non-trainable params: 0
```

Und wie man auf dem folgenden Bild erkennen kann, ist die Präzision der Validierung immer sehr nah an der Trainingspräzision dran.

```
Train for 1546 steps, validate for 663 steps
Epoch 1/15
1546/1546 [==============================] - 40s 26ms/step - loss: 0.8146 - accuracy: 0.8126 - val_loss: 0.8034 - val_accuracy: 0.8138
Epoch 2/15
1546/1546 [==============================] - 41s 26ms/step - loss: 0.7918 - accuracy: 0.8176 - val_loss: 0.7794 - val_accuracy: 0.8184
Epoch 3/15
1546/1546 [==============================] - 42s 27ms/step - loss: 0.7719 - accuracy: 0.8208 - val_loss: 0.7681 - val_accuracy: 0.8177
Epoch 4/15
1546/1546 [==============================] - 40s 26ms/step - loss: 0.7513 - accuracy: 0.8240 - val_loss: 0.7501 - val_accuracy: 0.8228
Epoch 5/15
1546/1546 [==============================] - 41s 27ms/step - loss: 0.7336 - accuracy: 0.8276 - val_loss: 0.7311 - val_accuracy: 0.8252
Epoch 6/15
1546/1546 [==============================] - 40s 26ms/step - loss: 0.7184 - accuracy: 0.8309 - val_loss: 0.7224 - val_accuracy: 0.8249
Epoch 7/15
1546/1546 [==============================] - 40s 26ms/step - loss: 0.7049 - accuracy: 0.8331 - val_loss: 0.7260 - val_accuracy: 0.8221
Epoch 8/15
1546/1546 [==============================] - 41s 27ms/step - loss: 0.6938 - accuracy: 0.8341 - val_loss: 0.7135 - val_accuracy: 0.8275
Epoch 9/15
1546/1546 [==============================] - 41s 27ms/step - loss: 0.6797 - accuracy: 0.8374 - val_loss: 0.6863 - val_accuracy: 0.8354
Epoch 10/15
1546/1546 [==============================] - 42s 27ms/step - loss: 0.6685 - accuracy: 0.8388 - val_loss: 0.7007 - val_accuracy: 0.8306
Epoch 11/15
1546/1546 [==============================] - 41s 27ms/step - loss: 0.6573 - accuracy: 0.8421 - val_loss: 0.6735 - val_accuracy: 0.8338
Epoch 12/15
1546/1546 [==============================] - 41s 26ms/step - loss: 0.6448 - accuracy: 0.8446 - val_loss: 0.6579 - val_accuracy: 0.8361
Epoch 13/15
1546/1546 [==============================] - 41s 26ms/step - loss: 0.6351 - accuracy: 0.8467 - val_loss: 0.6658 - val_accuracy: 0.8358
Epoch 14/15
1546/1546 [==============================] - 42s 27ms/step - loss: 0.6293 - accuracy: 0.8473 - val_loss: 0.6695 - val_accuracy: 0.8335
Epoch 15/15
1546/1546 [==============================] - 42s 27ms/step - loss: 0.6180 - accuracy: 0.8488 - val_loss: 0.6281 - val_accuracy: 0.8437
```

Dies ist einer der Gründe für den Verdacht auf ein Problem mit den verwendeten Daten.

Quellen

1. Daten:
   1.1. http://etlcdb.db.aist.go.jp/
2. Links zu Texten und Lernmaterialien:
   1. https://www.edureka.co/blog/convolutional-neural-network/
   2. Google Image Classification
   3. Keras Klassifizierungsbeispiel
   4. Charlie Tsai Handwritten Japanese Characters
3. Bücher:
   3.1. Python Machine Learning, Second und Third Edition: Sebastian Raschka & Vahid Mirjalili, Packt Publishing

Werkzeuge

1. Github, siehe: https://github.com/erikahub/KanjiOCR
2. Google Docs
3. Visual Studio Code
4. Tensorflow
5. scikit learn
6. Trello

Kontaktinfo

Gruppenmitglied	E-Mail - Adresse
Robin Hüttebräucker	robin.huettebraeucker@gmail.com
Björn Lewe	bjoern.lewe@gmx.de

Erklärung zur Eigenständigen Ausarbeitung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. (Erklärung entnommen von https://www.uni-kassel.de/fb10/fileadmin/datas/fb10/Eigenstaendigkeitserklaerung.pdf)

Ort, Datum	Unterschrift Robin Hüttebräucker
05.03.2020	_____

Ort, Datum	Unterschrift Björn Lewe
05.03.2020	_____