Einstieg in Convolutional Neuronale Netze mit Keras

mt-itsolutions.com/ki-werkstatt/einstieg-in-convolutional-neuronale-netze-mit-keras/

Johannes Höhne 27. November 2018

#KI-Werkstatt

Worum es hier geht

In meinem letzten Blog Einstieg in neuronale Netze mit Keras habe ich recht umfassend beschrieben, wie man sich eine „Werkbank“ für die Arbeit mit einfachen neuronalen Netzen zusammen baut. Als Beispiel habe ich die MNIST Datenbank für handschriftliche Ziffern verwendet.
Die Genauigkeit der Vorhersage war mit ungefähr 96,5% zwar schon beeindruckend, für die Realität aber völlig unbrauchbar. Die Vorhersagegenauigkeit von Menschen liegt bei diesem Datensatz etwa bei 99,7%. Somit gibt es also noch viel Luft nach oben.
Ich will in diesem Blog versuchen mit sehr einfachen Mitteln auf 99,1% Genauigkeit zu kommen. Dies schafft man mit etwas Glück mit einer **CNN** Netzarchitektur.

Was sind CNN?

Convolutions, also Faltungen sind eine von zwei gängigen Architekturen die ein Netz tief machen (*Die andere Architektur sind die **RNN**, Recurrent Nuron Networks, über die ich in einem weitern Blog schreiben werde*). Die Tiefe bezeichnet ja die Anzahl der Schichten in einem Netz. Dabei fasst man aber mehrere Schichten zu funktionsbereichen zusammen. Hier ist der besondere Funktionsbereich das Falten von Bildern.
Die ganz einfachen, Fully Connected Neuronalen Netze sind zwar ein sehr mächtiges Tool, sie reagieren aber empfindlich auf zu große Hypothesenräume (*kleine Erinnerungsstütze: Ein Datensatz heißt **Sample**, die Attribute eines Samples heißen **Features** ==> Die Anzahl der Features bestimmt die Anzahl der Dimensionen im **NN** ==> Das bezeichnet man als **Hypothesenraum***).
Umgangssprachlich kann man sagen, dass zu viele beschreibende Attribute das Netz verwirren. Da funktionieren wir Menschen übrigens ganz ähnlich. Versuch doch mal aus folgendem Satz die Emotion abzuleiten: „Ach, ich weiß auch nicht. Es regnet, der Himmel ist grau und ich bin heute irgendwie mit dem linken Fuß aufgestanden.“ Es gelingt zwar, aber folgende zwei Zeichen zeigen die gleiche Emotion und sind um ein vielfaches schneller und unmissverständlicher zu erfassen :(.
Genau das machen die Convolution Layer vor den Fully Connected Layern. Sie erkennen in einer großen Menge von Features eindeutige Muster und heben diese hervor.
Nehmen wir zur Veranschaulichung das Bild einer Katze. Convolution Layer gehen dabei von den ganz feinen Strukturen also einem kleinen Strich, einem Punkt oder einer Farbe zu immer größeren Mustern, wie etwa einem Katzenohr, einer Katzennase, Katzenaugen. Die Fully Connected Layers bekommen danach also nicht mehr eine Menge von Bildpunkten zu sehen, sondern sie bekommen gesagt: „Auf dem Bild ist ein Katzenkopf“. Dadurch reduziert sich die Anzahl an Features, aus denen der Fully Connected Layers Teil des Netzes seine Antwort ableiten muss, was ihm die Arbeit leichter macht.
Es gibt ein wirklich phantastisches YouTube Video, das ganz anschaulich in die Mathematik hinter den CNN führt. Unbedingt anschauen. Es macht aus vielen ? ein ! im Kopf. A friendly introduction to Convolutional Neural Networks and Image Recognition

Vom Daten besorgen bis zum Feature Engineering

In meinem letzten Blog habe ich sehr detailliert beschrieben, wie man die Samples fürs Trainieren und Testen in seine „Werkbank einspannt“. Das kann ich hier also überspringen und nur noch die Codeblocks aufführen:

Boiler Plate

```
#tensorflow
import tensorflow as tf
from tensorflow.python.client import device_lib
#keras
import keras
from keras.backend.tensorflow_backend import set_session
from keras.datasets import mnist
from keras import models
from keras import layers
from keras import regularizers
from keras.utils import to_categorical
from keras import backend as K
#numpy
import numpy as np
#matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
#scipy
from scipy import ndimage
from scipy.ndimage.filters import gaussian_filter
#time
import time
#Force GPU support with growing memory
config = tf.ConfigProto()
config.gpu_options.allow_growth = True # dynamically grow the memory used on the GPU
sess = tf.Session(config=config)
set_session(sess)
```

Using TensorFlow backend.

Daten besorgen

```
#keras / tensorflow has already the full MNIST dataset
(train_images_raw, train_labels_raw), (test_images_raw, test_labels_raw) = mnist.load_data()
```

Feature Engineering

Hier ist dann doch noch eine Kleinigkeit anders als beim letzten Mal.
Ich konvertiere den 3D Bildtensor in einen 4D Tensor, da Keras das so für seine Convolutional Layers als Input braucht:
* 1. Dimension: Die Nummer des Samples
* 2. Dimension: Die Bildzeile
* 3. Dimension: Die Bildspalte
* 4. Dimension: Die Farbkanäle (hier haben wir nur einen, da es sich ja um Graustufenbilder handelt. Sonst sind es drei für Rot, Grün und Blau)
Ich teile alle Grauwerte durch ihren Maximalwert (255), um sie in den Wertebereich zwischen Null und Eins zu trimmen. Dieser Wertebereich ist für die Aktivierungsfunktionen bekömmlicher.

```
#convert samples into (60.000, 28,28, 1) norm - tesnsors
train_images = train_images_raw.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255 #255 different gray scales
test_images = test_images_raw.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255 #255 different gray scales
#convert labels into one hot representation
train_labels = to_categorical(train_labels_raw)
test_labels = to_categorical(test_labels_raw)
```

Training

Das Modell Konfigurieren

Ich habe mich hier für eine extrem schlanke Konfiguration meines Netzes entschieden. Man findet zum Beispiel auf Kaggle wesentlich leistungsstärkere, aber eben auch größere Konfigurationen.
Allerdings kommt mir die reduzierte Komplexität später wieder zugute, wenn ich ein paar interessante Details des Netzes herausheben möchte. Zudem verfügt das Netz über alle wichtigen Bestandteile, die man für seine Experimente braucht.

Die Convolutions

Über `models.Sequential()` hole ich mir von Keras ein frisches Modell. Dann füge ich den ersten Convolutional Layer hinzu. Dabei sagt die `32` und die `(3, 3)`, dass ich 32 verschiedene Filter der Größe 3 auf 3 Pixel trainieren möchte. In Aktion fährt dann ein 3 x 3 großes Fenster über das Bild und wendet alle 32 Filter pro Ausschnitt an. Jeder dieses Filter konzentriert sich auf ein bestimmtes kleines Detail wie zum Beispiel einen waagrechten schwarzen Strich. Findet der Filter diesen Strich meldet er das Ergebnis weiter. Findet er ihn nicht, bleibt der Filter dunkel. Das Attribute **padding** gibt an, ob die Ränder mit beachtet werden sollen oder nicht. Dabei bedeutet ‚same', dass sie beachtet werden sollen. Als Aktivierungsfunktion habe ich hier die **relu** verwendet. Die ist ein wenig performanter als die **sigmoid** und skaliert über den Wertebereich linear, was **sigmoid** ja genau nicht macht. Du solltest beim tuning verschiedene Funktionen versuchen.

Des Pooling

Danach kommt der erste Pooling Layer, der das Ergebnis des Filters um den Faktor 2 in jede Richtung reduziert. Bei den traditionellen **CNN** verwendet man Pooling Layer, man geht aber mehr und mehr dazu über, das Ergebnis zu normalisieren, statt zu konzentrieren. Das liefert oft bessere Vorhersagen, ist aber um einiges langsamer beim Training. Du kannst ja die `layers.MaxPooling2D(2, 2)` durch `layers. BatchNormalization()` austauschen und die Ergebnisse miteinander vergleichen.

Ein Paar ist nicht genug

Ich staple insgesamt vier Convolution Layer übereinander und reduziere die Anzahl der Filter zum Schluss von **32** auf **16**. Das ist eher unüblich. Eigentlich bläht man die Anzahl der Filter nach hinten auf. Man fängt zum Beispiel mit 32 Filtern an, und erweitert dann auf 64. Ich mache hier aber aus zwei Gründen genau das Gegenteil. Zum einen sollen nachher in den Filtern eindeutige Merkmale für zehn verschiedene Ziffern freigestellt werden. Dazu sollte ein Raum von 16 drei auf drei großen Matrizen mehr als ausreichend sein. Zum anderen möchte ich nachher visualisieren, wie die Convolutions die einzelnen Ziffern codieren. Das zeigt sich an einer übersichtlichen Menge leichter.

An der Grenze wird gerade gebogen

Am Übergang vom **CNN** zum Fully Connected Nuron Network, biegt der `Flatten()` Layer den Output 3D Tensor aus dem Layer der Convolutions in einen 1D Tensor auf. 16 Filter x 3 auf 3 Pixel macht also einen Vektor der Länge 144.

Dropout

Ich habe im Modell an einigen Stellen **Dropout Layer** hinzugefügt. Das ist ein sehr simples Mittel um **overfitting** zu vermeiden. Die 0,4 sagt dabei, dass 40% der Ergebnisse pro Iteration zufällig ausgewählt und verworfen werden. Da pro Epoche immer wieder die selben Samples zum Trainieren verwendet werden, vermeidet man so, dass die Entscheidung (aka Gewichtung) des Netzes an einigen wenigen prominenten Features festgemacht wird. Neuronale Netze verhalten sich da ähnlich wie wir Menschen auch. Sie wählen immer den einfachsten Weg. Mit den Dropout Layern verstellt man die einfachen Wege und zwingt Das Netz auch in anderen Dimensionen nach Minima zu suchen.

Regularizers

Beim ersten **Dense Layer** habe ich noch den Parameter `kernel_regularizer = regularizers.l2(0.007)` hinzugefügt. Regularizer nähern sich dem Problem **Overfitting** von der anderen Seite. Sie verhindern, dass das Netz einzelne Gewichte überinterpretiert. Zu laute Töne werden damit also heruntergeregelt. Auch hier gibt es eine Analogie: In der Klasse von Harry Potter meldet sich immer Hermine. Da sie zudem auch meist die richtige Antwort kennt, wird sie nicht mehr so oft aufgerufen. Man regelt sie also herunter, damit die anderen auch drankommen. Der Faktor `0.007` gibt an, wie stark man herunterregeln möchte.

Was man noch machen könnte

Ein Mittel für das Verbessern des Modells habe ich hier weggelassen: Das **Augmentieren der Daten.** Dabei werden die einzelnen Samples verzerrt, um mehr Testdaten zu erhalten. Eigentlich ist das nicht sonderlich schwer hinzuzufügen und es bringt die Lösung weit nach vorne. Um genau zu sein augmentieren die top MNIST Modelle in Kaggle alle. Allerdings bläht mir das den Python Code um einige *Generatoren* auf. Diese zusätzlich Komplexität möchte ich hier weglassen, und mich auf das Wesentliche konzentrieren.

```
model = models.Sequential()
# Convolution Layers
model.add(layers.Conv2D(32, (3, 3), padding = 'same', activation = 'relu', input_shape = (28, 28, 1)))
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Conv2D(32, (3, 3), padding = 'same', activation = 'relu'))
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Conv2D(32, (3, 3), padding = 'same', activation = 'relu'))
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Conv2D(16, (3, 3), padding = 'same', activation = 'relu'))
model.add(layers.Dropout(0.4))
# Fully connected Layers
model.add(layers.Flatten())
model.add(layers.Dropout(0.4))
model.add(layers.Dense(128, kernel_regularizer = regularizers.l2(0.007), activation='relu'))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(10 , activation='softmax'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 32)	0
conv2d_3 (Conv2D)	(None, 7, 7, 32)	9248
max_pooling2d_3 (MaxPooling2	(None, 3, 3, 32)	0
conv2d_4 (Conv2D)	(None, 3, 3, 16)	4624
dropout_1 (Dropout)	(None, 3, 3, 16)	0
flatten_1 (Flatten)	(None, 144)	0
dropout_2 (Dropout)	(None, 144)	0
dense_1 (Dense)	(None, 128)	18560
dropout_3 (Dropout)	(None, 128)	0