# GAN - Teil 2 - Keras kann auch Ziffern schreiben

## Worum geht es in diesem Teil

Nachdem ich im ersten Teil den grundlegenden Trick der GAN beschrieben habe, möchte ich hier ein einfaches GAN bauen, dass in der Lage ist, die handschriftlichen Ziffern aus der MNIST Datenbank täuschend echt nachzumachen. Ich verwende für diesen „Zaubertrank“ nur die elementaren Zutaten aus der Speisekammer der neuronalen Netze um mich auf die GAN Spezifika konzentrieren zu können. Nichts desto Trotz ist das Ergebnis schon recht beeindruckend. Wenn ich hier von Zaubertrank und Speisekammer rede, so geschieht das nicht ganz ohne Grund. In der Literatur wird gerne der Ausdruck "Alchemie der GAN" verwendet, der darauf hinweisen soll, dass man viel Erfahrung und Gespür beim Bau von GAN's benötigt. Das liegt daran, dass wir mit unseren neuronalen Netzen mittels Gradientenverfahren nicht einem festen Minima entgegenschreiten, sondern wir suchen ein Gleichgewicht zwischen G(z) und D(x). Dadurch verschieben sich die Minima in den Netzten nach jedem Iterationsschritt. Die Aufgabe beim Bau von GAN's ist es also seine Netzte so zu bauen, dass das Gleichgewicht leicht zu finden ist. Man braucht ein Netz wie eine tiefe Schüssel, in der eine kleine Kugel automatisch dem tiefsten Punkt und damit seinem Gleichgewicht zustrebt. Das andere Extrem ist ein Netz wie ein Bleistift, den man auf der Spitze ausbalancieren möchte, was bekanntlich schnell mal frustrieren kann. Hängen wir uns also unsere Druidenmäntel um und greifen zu unseren goldenen Sicheln 😊

## Die lästigen Vorbereitungen

Wie immer sammle ich alle nötigen Importstatements in einem Block

#tensorflow  
import tensorflow as tf  
from tensorflow.python.client import device\_lib  
from keras.backend.tensorflow\_backend import set\_session  
  
import keras  
from keras.layers import Input  
from keras.models import Model, Sequential  
from keras.layers.core import Dense, Dropout  
from keras.layers.advanced\_activations import LeakyReLU  
from keras.datasets import mnist  
from keras.optimizers import Adam, RMSprop  
from keras import initializers  
  
config = tf.ConfigProto()  
config.gpu\_options.allow\_growth = True # dynamically grow the memory used on the GPU  
config.log\_device\_placement = True # to log device placement (on which device the operation ran)  
 # (nothing gets printed in Jupyter, only if you run it standalone)  
sess = tf.Session(config=config)  
set\_session(sess) # set this TensorFlow session as the default session for Keras  
print("TensorFlow Version:\t{}".format(tf.\_\_version\_\_))  
print("Keras Version:\t\t{}".format(keras.\_\_version\_\_))  
print("Found GPU & CPU devices:\n{}".format(device\_lib.list\_local\_devices()))  
  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
# ein hübsches Tool für Progerss Bars  
from tqdm import tqdm  
  
# Für das Rauschen brauchen wir Zufallszahlen  
np.random.seed(911972)

## Daten vorbereiten

* Zuerst besorge ich mir die Testdaten auf dem üblichen Weg.
* Da ich hier wirklich die einfachsten Netze verwenden möchte (also simple Dense Netze), sammle ich alle Pixelinformationen nicht in einer zwei dimensionalen 28 x 28 Matrix, sondern in einem 28 x 28 = 784 Elemente großen Vektor.
* Ein erster Erfahrungswert ist es, den Tangens Hyperbolicus als finale Aktivierungsfunktion zu verwenden. Deshalb normiere ich meine Daten auf das Intervall von [-1, 1].
* Die Größe des Input Vektors für den Generator setzte ich Zufällig auf 64
* Die Optimierung der Hyperparameter ist die wesentliche Fleißarbeit. Somit sammle ich sie hier in einem Block zusammen.

(real\_images\_all, \_\_), (\_\_, \_\_) = mnist.load\_data()  
print('1. Rohdaten:')  
print(real\_images\_all[0, 14, 10:20])  
  
real\_images\_all = real\_images\_all.reshape(60000, 784)  
print('2. Zum Vektor aufgebogen:')  
print(real\_images\_all[0, 14\*28 +10 : 14\*28 + 20])  
  
real\_images\_all = real\_images\_all.astype('float32') / 255 \* 2 - 1  
print('3. in das Interval [-1, 1] normiert:')  
print(real\_images\_all[0, 14\*28 +10 : 14\*28 + 20])  
  
z\_size = 64  
  
# Da ich für alle Netzte einen Optimizer brauche, der ein wenig sanfter als der Default eingestellt ist, monfiguriere ich ihn einmal  
#optimizer= Adam(lr=0.0002, beta\_1=0.5)  
optimizer= RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-8)  
  
# dropout ist wichtig bei GAN  
dropout\_rate = 0.2  
  
# die steigung der LeakyReLu im negativen Bereich  
leaky\_faktor = 0.2

## Der Generator

Im Prinzip ist der Generator das einfachste Netz, das man bauen kann. Eine kleine Besonderheit hat es aber dennoch. Ich verwende hier ‚LeakyReLU‘ als Ansatzfunktion. Seit einiger Zeit setzt sich ReLU als Allzweckwaffe unter den Ansatzfunktionen der Hidden - Layer durch. Allerdings hat sie eine kleine Schwäche. Dadurch, dass sie den negativen Wertebereich auf Null abbildet, lässt sie Knoten schnell absterben. Ein einmal abgestorbener Knoten wird nicht mehr aktiviert, da auf ihn kein Fehler mehr aufgeteilt wird. Das wirkt sich negativ auf das „tiefe“ propagieren von Fehlern aus. Mit der LeakyReLu wird dieses Problem repariert, indem man den negativen Wertebereich nicht ganz auf 0 abbildet.

Als Aktivierungsfunktion des letzten Layers wähle ich wie bereits erwähnt den tanh als „magische“ Zutat.

generator = Sequential()  
generator.add(Dense(256, input\_dim=z\_size))  
generator.add(LeakyReLU(leaky\_faktor))  
  
generator.add(Dense(512))  
generator.add(LeakyReLU(leaky\_faktor))  
  
generator.add(Dense(1024))  
generator.add(LeakyReLU(leaky\_faktor))  
  
generator.add(Dense(784, activation='tanh'))  
generator.compile(loss='binary\_crossentropy', optimizer=optimizer)  
generator.summary()

## Der Diskriminator

Der Diskriminator ist ähnlich schlicht aufgebaut, wie der Generator. Die Input Dimension ist 28 x 28, also jeweils ein Bild. Die Output Dimension ist ein Skalar zwischen Null und Eins. Der Diskriminator verfügt anders als der Generator über Dropout Layer, um auch hier den Zufall mit einzubinden. Der Zufall verhindert im GAN, dass sich die Netze zu schnell einschnüren und in einen dünn besiedelten Zustand gleiten. Das ist bei GAN’s sehr schlecht, da die „Kreativität“ in der Vielfalt der Möglichkeiten liegt. Somit müssen möglichst viele Gewichte aktiviert sein um eine möglichst breite Varianz zu bekommen. Der Diskriminator bekommt seine Vielfalt übrigens nicht durch Dropout Layer, sondern durch das Rauschen des Input Vektors z.

discriminator = Sequential()  
discriminator.add(Dense(1024, input\_dim=784))  
discriminator.add(LeakyReLU(leaky\_faktor))  
discriminator.add(Dropout(dropout\_rate))  
  
discriminator.add(Dense(512))  
discriminator.add(LeakyReLU(leaky\_faktor))  
discriminator.add(Dropout(dropout\_rate))  
  
discriminator.add(Dense(256))  
discriminator.add(LeakyReLU(leaky\_faktor))  
discriminator.add(Dropout(dropout\_rate))  
  
discriminator.add(Dense(1, activation='sigmoid'))  
discriminator.compile(loss='binary\_crossentropy', optimizer=optimizer)  
discriminator.summary()

## Das GAN

Um den Generator trainieren zu können, muss man das GAN zusammensetzen. Dazu nutzt man bei Keras das funktionale API. Damit kann man einem Modell weitere Layer hinzufügen. So wird hier der Input Layer auf den Generator und dieser auf den Diskriminator geschichtet. Danach ruft man den Modell Konstruktor auf, um das GAN zu bekommen. Übrigens spiegelt sich in der funktionalen Schreibweise von Python die theoretische Herleitung der GAN’s wieder D(G(z)).

# der Diskriminator wird hier gleich eingefroren  
discriminator.trainable = False  
  
# der verrauschte Input Vektor  
z = Input(shape=(z\_size,))  
  
# Das GAN D(G(z))  
gan\_core = discriminator(generator(z))  
  
# Um ein eigenständiges Netz zu erhalten, baut man sich ein neues Modell  
gan = Model(inputs = z, outputs = gan\_core)  
gan.compile(loss = 'binary\_crossentropy', optimizer = optimizer)  
gan.summary()

## Das Training

Wie im Teil eins beschrieben werden hier folgende Schritte immer wieder abgearbeitet: 1. Zuerst generiert man mit dem Generator einen Stapel Fake – Bilder, wobei wir ein nach Gauß normalverteiltes Rauschen erzeugen. (Es verändert übrigens die Qualität der Ergebnisse, wenn man die Werte σ und µ feinabstimmt.) 2. Diese Bilder nimmt man zusammen mit gleich vielen echten Bildern, um den Diskriminator zu trainieren. Dadurch wird der Diskriminator ein wenig besser. 3. Das GAN trainiert man mit einem Stapel Rauschens und fordert, dass das Ergebnis immer *real* sein sollen, da wir ja die Realität fälschen wollen. D(G(z)) = 1.

epochs = 5000  
batch\_size = 128  
batch\_count = real\_images\_all.shape[0] / batch\_size  
  
σ = 1  
µ = 0  
  
history\_g = []  
history\_d = []  
  
for e in range(1, epochs+1):  
 print('-'\*15, 'Epoch %d' % e, '-'\*15)  
 for count in tqdm(range(int(batch\_count))):  
 # 1.1 wir bauen 'bach\_size' viele z vektoren mit normalverteiltem Rauschen der Länge 'z\_size'  
 z = np.random.normal(µ, σ, size=[batch\_size, z\_size])  
 # 1.2 daraus generieren wir Fake Bilder  
 fake\_images = generator.predict(z)  
  
 # 2.1 dazu entnehmen wir eine gleich große Anzahl an echten Bildern  
 real\_images = real\_images\_all[np.random.randint(0, real\_images\_all.shape[0], size=batch\_size)]  
 # 2.2 beide Stapel zusammen ergeben die Trainingsmenge für den Diskriminator  
 x\_dis = np.concatenate([real\_images, fake\_images])  
 # 2.3 der Ergebnisvektor wird mit null also fake vorbelegt  
 y\_dis = np.zeros(2\*batch\_size)  
 # 2.4 die richtigen Ergebnisse werden nicht mit 1 sondern mit 0.9 belegt.   
 # Das nennt sich "one-sided label smoothing" und dient dazu die "Früchte"   
 # nicht allzu hoch zu hängen   
 y\_dis[batch\_size:] = 0.9  
 # 2.5 den Diskriminator auffrieren  
 discriminator.trainable = True  
 # 2.6 und trainieren  
 history\_d.append(discriminator.train\_on\_batch(x\_dis, y\_dis))  
  
 # 3.1 Wieder eine Stapel Rauschen bauen  
 x\_gan = np.random.normal(µ, σ, size=[batch\_size, z\_size])  
 # 3.2 Um echte Bilder vorzutäuschen belegen wir die Antwort mit Null  
 y\_gan = np.zeros(batch\_size)  
 # 3.3 Den diskriminator einfrieren, um seine Gewichte nicht zu ändern  
 discriminator.trainable = False  
 # 3.4 und so den generator mittelbar als Teil des GANs trainieren  
 history\_g.append(gan.train\_on\_batch(x\_gan, y\_gan))  
   
 # alle 20 Schritte werfe ich eine kleine Auswertung aus   
 if e == 1 or e % 20 == 0:  
 # 1. Die entwicklung der Verlusraten beider Trainings  
 plt.plot(history\_g, 'r', label='generator loss')  
 plt.plot(history\_d, 'b', label='discriminator loss')  
 plt.title('Training of D(x) and D((g(z)))')  
 plt.legend()  
 plt.show()  
 plt.close()  
  
 examples = 100  
 dim=(10, 10)  
 figsize=(10, 10)  
 noise = np.random.normal(µ, σ, size=[examples, z\_size])  
 generated\_images = generator.predict(noise)  
 generated\_images = generated\_images.reshape(examples, 28, 28)  
  
 # 2. Ein paar Demobilder  
 plt.figure(figsize=figsize)  
 for i in range(generated\_images.shape[0]):  
 plt.subplot(dim[0], dim[1], i+1)  
 plt.imshow(generated\_images[i], interpolation='nearest', cmap='gray\_r')  
 plt.axis('off')  
 plt.tight\_layout()  
 plt.savefig('output/gan\_generated\_image\_epoch\_%d.png' % e)  
 plt.close()  
   
 # alle 40 Schritte speicher ich das Modell des generators  
 if e % 40 == 0:  
 # Gewichte speichern  
 generator.save('output\_model/generator\_model\_%d.h5' % e)  
 # Konfiguration speichern  
 model\_json = generator.to\_json()  
 with open("output\_model/generator\_model\_{}.json".format(e), "w") as json\_file:  
 json\_file.write(model\_json)

## Schluss und wie geht es von hier weiter

### Ergebnisse

Nach der ersten Epoche sehen die Ziffern noch so aus:

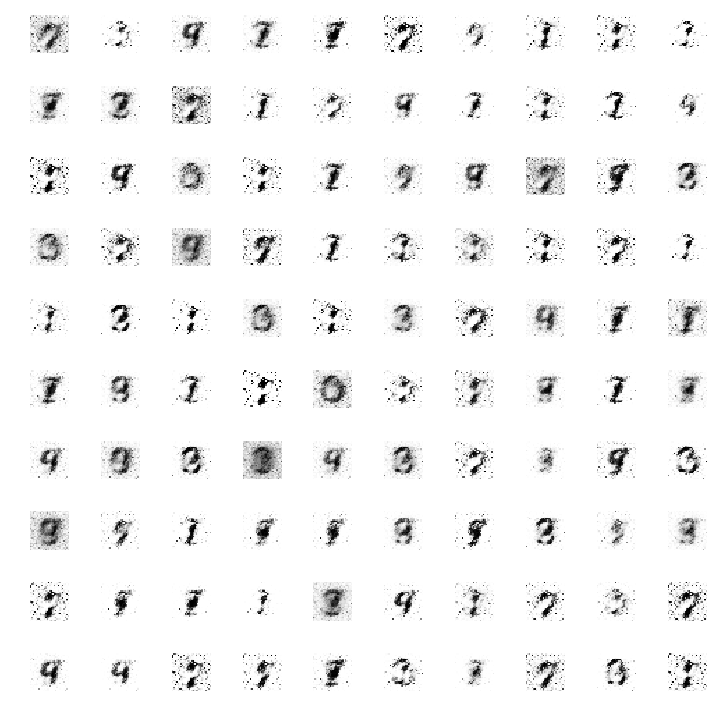


Abb. 1: Ziffern nach der ersten Epoche

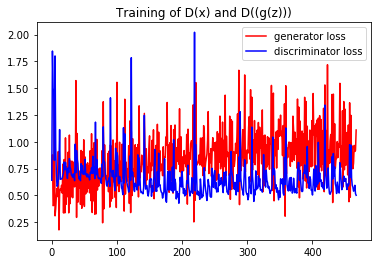


Abb. 2: Fehlerrate nach der ersten Epoche

Wenn man dem Netz nun ein wenig Zeit gibt, wird man schon nach wenigen Iterationen feststellen, dass Die Ergebnisse durchaus beeindruckend sind.

Nach 300 Epochen sieht das Ergebnis dann so aus

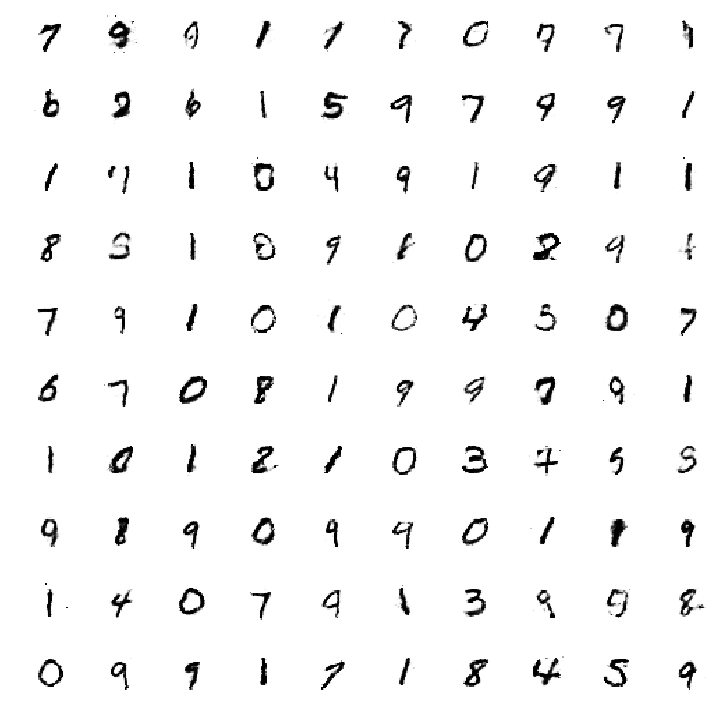


Abb. 3: Ziffern nach 300 Epochen

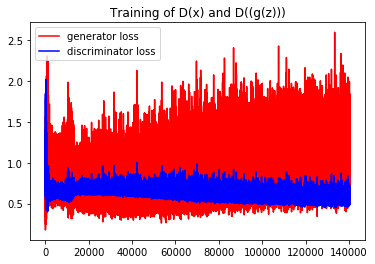


Abb. 4: Fehlerrate nach 300 Epochen

Lässt man das Netz weiter trainieren, stellt man fest, dass sich das Ergebnis nicht mehr wesentlich verbessert. Hier das Ergebnis nach 700 Epochen:



Abb. 5: Ziffern nach 700 Epochen

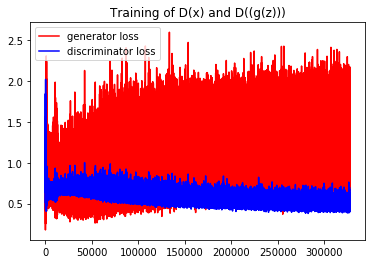


Abb. 6: Fehlerrate nach 700 Epochen

Anscheinend ist das Netz an seiner Leistungsgrenze und konvergiert wirklich in ein stabiles Gleichgewicht. Natürlich fällt auf, dass viele der Ziffern sehr geschmiert erscheinen. Dazu darf man aber nicht vergessen, dass auch die originalen Ziffern, die der Diskriminator zum Trainieren bekommen hat nicht gerade einen Schönschreibwettbewerb gewinnen würden: 

Im Endeffekt erfindet das Netz nicht wirklich neue Ziffern, sondern bildet ein statistisches Mittel aus allen Bildern, mit denen es trainiert wurde und hebt Variationen aus dem Latenzraum, also den Raum der möglichen richtigen Ziffern. Was man nun als folgenden Schritt noch machen sollte, sind zwei Prüfungen:

### Verteilung der Ziffern

Sind die erfundenen Ziffern gleich verteilt? Wenn ich also zum Beispiel 100.000 Ziffern erfinden lasse, so sollten etwa 10.000 Stück pro einzelne Ziffer vorhanden sein.

### Trennschärfe der Ziffern

Wie deutlich lassen sich die erfundenen Ziffern erkennen? In einem meiner letzten Blogs ([So entwirft man ein Top CNN](https://www.mt-ag.com/so-entwirft-man-ein-top-cnn/)) habe ich ein recht passables Netz implementiert, das Ziffern mit hoher Genauigkeit erkennt. Wenn die erfundenen Ziffern nun durch dieses Netz geschickt werden, kann man prüfen, wie genau eine Ziffer zwischen Null und Neun erkannt wird. Ein mögliches Maß für diese Genauigkeit ist die von mir definierte Trennschärfe [Trennschärfe bei der Klassifizierung](https://www.mt-ag.com/trennschaerfe-bei-der-klassifizierung/)

### Und weiter

Die Architektur um die GAN’s hat sich schnell und tief entwickelt. Allerdings liegt der Fokus stark auf Bildern. Ich denke, man sollte auch Texten, Musik und Daten einer genaueren Untersuchung unterziehen, um auch hier Artefakte täuschend echt nachzuahmen. Alles in allem geht mit den GAN’s für mich die Reise in Richtung Turing Test…

Ich wünsche Dir viel Spaß beim Ausprobieren.