# Praktikumsbericht

## Anfängerpraktikum
## Maschinelles Lernen im Kontext Trigonometrischer Funktionen

Betreuer: Prof. Dr. Guido Kanschat

Ahmet Efe, Sven Leschber, Mika Rother

21. Februar 2020

# Inhaltsverzeichnis

# Kapitel 1

# Einarbeiten in die Grundkonzepte von Maschinellem Lernen

## 1.1 Einführung in *TensorFlow*

Zu Beginn unseres Praktikums mussten wir uns erstmal damit auseinandersetzen, was *TensorFlow* überhaupt ist und wie genau *Machine Learning* funktioniert. Dazu haben wir uns als allererstes auf der offiziellen Webseite[1] von *TensorFlow* nach anfängerfreundlichen Übungen umgeschaut, um den Umgang mit den Grundkonzepten von *TensorFlow* zu lernen. Dazu haben wir ein paar praktische Tutorials[2] gefunden, bei denen man zum Beispiel eine ganz einfache lineare Funktion

$$f(x) = 3x + 1$$

bekommen hatte und mit wenigen Eingabe- und Ausgabewerten dem Programm verständlich machen sollte, dass die Eingabewerte $x$ auf die Funktion $f(x)$ abgebildet werden sollen. Das Programm dafür sieht dann wie folgt aus:

```
1  import tensorflow as tf
2  import numpy as np
3  from tensorflow import keras
4
5  model = tf.keras.Sequential([keras.layers.Dense(units=1, input_shape
       =[1])])
6  model.compile(optimizer='sgd', loss='mean_squared_error')
7
8  xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0], dtype=float
       )
9  ys = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0, 16.0, 19.0], dtype=
       float)
10
11 model.fit(xs,ys,epochs=500)
12 print(model.predict([10.0]))
```

Listing 1.1

[1] https://www.tensorflow.org
[2] https://www.tensorflow.org/tutorials/quickstart/beginner

Es erzeugte die folgende Ausgabe:

```
1  $ [[31.00079]]
```

Man kann also gut sehen, dass der Computer mit acht Eingabe-Ausgabe-Wert Paaren schon sehr gut (bis auf 4 Nachkommastellen) diese Funktion approximiert.
Nachdem wir also ein paar erste kleine Programme haben laufen lassen, hatten wir uns etwas genauer mit den Details des Programms auseinandergesetzt. Vor allem wichtig waren dabei das Kreieren eines Modells MODEL = TF.KERAS.SEQUENTIAL() (*Listing 1.1, codeline 5*), das Kompilieren MODEL.COMPILE() (*Listing 1.1, codeline 6*) und das Trainieren des Modells MODEL.FIT() (*Listing 1.1, codeline 11*). Wir haben uns nun etwas genauer diese drei Bestandteile des Programms angeschaut.

## 1.2 Austesten von Parametern

Nachdem wir unsere ersten Erfahrungen mit *TensorFlow* gemacht haben, wollten wir jetzt ein wenig die Parameter variieren, um herauszufinden welche Folgen es hat, wenn man zum Beispiel die Anzahl der Epochen (*Listing 1.1, codeline 11*) verändert. Oder, ob das Programm den Wert deutlich besser approximiert, wenn man ihm wesentlich mehr Werte zur Verfügung stellt. Bei so einem einfachen Neuronalen Netzwerk, das nur eine lineare Funktion approximieren soll, hatten wir allerdings kaum Unterschiede gesehen, wenn wir die Anzahl der Epochen oder die Menge der Werte verändert hatten. Wir werden später aber sehen, dass es bei komplexeren Programmen wesentlich schwieriger wird die Werte der Parameter so festzulegen, dass das Programm möglichst genau und schnell läuft.

# Kapitel 2

# Approximation des Sinus

## 2.1 Erstellen eines Modells

Nun haben wir mit der ersten Aufgabe angefangen, der Approximation der Sinus Funktion $\sin(x)$ auf einem Intervall $[0, \pi]$. Dafür haben wir das in *Listing 1.1* aufgeführte Programm genommen und es erweitert. Zuerst mussten wir mehr Layer hinzufügen, da die Approximation des Sinus eine größere Herausforderung für den Computer darstellt als $f(x) = 3x + 1$. Zudem konnten wir hier nicht nur mit acht Parametern arbeiten, sondern haben erstmal angefangen mit ca. 32 Werten unser Netzwerk zu trainieren. Zu diesem Zeitpunkt hatten wir folgendes Modell für den Sinus erstellt:

```
model = keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(1,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
    ])

model.compile(optimizer = tf.optimizers.Adam(), loss='
   mean_squared_error')
# xs_train are the x values and sin_train are the sin values of x
model.fit(xs_train, sin_train, epochs=500)
```

Listing 2.1

Mit diesem Modell hatten wir schon sehr gute Approximationen für den Sinus bekommen. Nun wollten wir das ganze aber etwas professioneller machen, sprich dem Modell nicht nur Arrays von Eingabe-Ausgabe-Werten geben, sondern eigene Datensets erstellen.

## 2.2 Erstellen von Datensets

In der Regel gibt man einem Modell drei Bestandteile von Daten: Trainings-, Validierungs- und Testdaten. Mit den Trainingsdaten trainiert das Modell sein Netzwerk, die Validierungsdaten sind dann dazu da, um während des Kompilierens die Genauigkeit zu verbessern, das heißt diese beiden Daten erhält das Modell bereits beim Training MODEL.FIT(). Jetzt bleiben noch die Testdaten; diese werden dazu benutzt um das Modell

zu evaluieren (MODEL.EVALUATE()). Um zu verhindern, dass die ersten 70% des Intervalls zu Trainingsdaten, die nächsten 15% zu Validierungsdaten und die letzten 15% zu Testdaten werden, sorgen wir für ein Durchmischen des Datensets. Dafür haben wir von *Scikit* ein Modul zum Mischen gefunden[1]. Unter der Verwendung dieses Tools hatten wir nun alles zusammen, um ein vernünftiges Datenset zu erstellen mit dem wir ab sofort arbeiten konnten. Wir haben gemerkt, dass es vor allem deshalb praktisch ist, ein gutes Datenset zu haben, weil man dieses sehr gut für die anderen Approximationen übernehmen kann. Unser fertiges Datenset für die Sinus Approximation sah dann wie folgt aus:

```
# xs are the x values and sin are the sine values of x
# xs_train are the training data, xs_val are the validation data,
    xs_test the test data
xs, sin = shuffle(xs, sin, random_state=0)
xs_train, xs_temp, sin_train, sin_temp = train_test_split(xs, sin,
    test_size = 0.3)
xs_val, xs_test, sin_val, sin_test = train_test_split(xs_temp,
    sin_temp, test_size = 0.5)
```

Listing 2.2

Logischerweise mussten wir nach dem Erstellen des Datensets auch das Modell bzw. das Programm überarbeiten, denn nun bekommt das Modell beim Trainieren auch noch Validierungsdaten mit denen es arbeiten muss. Zudem haben wir jetzt auch die Funktion MODEL.EVALUATE() benutzt, die dann wie folgt aussah:

```
results = model.evaluate(xs_test, sin_test)
print('Sin: test loss, test acc: ', results)
```

Listing 2.3

Das erzeugte die folgende Ausgabe:

```
  $ Sin: test loss, test acc:  0.00013300850696396083
```

Diese Zahl gibt einem ein ungefähres Gefühl dafür, wie gut die Approximation ist. Natürlich wollten wir uns aber nicht nur auf diese Zahl verlassen, sondern wollten auch wirklich ein Bild davon haben, wie gut unser Programm den Sinus nun wirklich nähert. Dazu haben wir einen Plot erstellt.

## 2.3 Ein Plot der Approximation

Damit man die Approximation auch qualitativ bewerten kann, hatten wir uns überlegt die Sinus Kurve zu plotten und dabei einmal die richtigen Werte zu verwenden und einmal unser Modell mehrere Werte vorhersagen zu lassen (MODEL.PREDICT()). Wenn wir das dann beides im selben Plot darstellen lassen, können wir sehen wie genau unser Modell die Kurve ausfüllt:

[1] https://scikit-learn.org/stable/modules/generated/sklearn.utils.shuffle.html

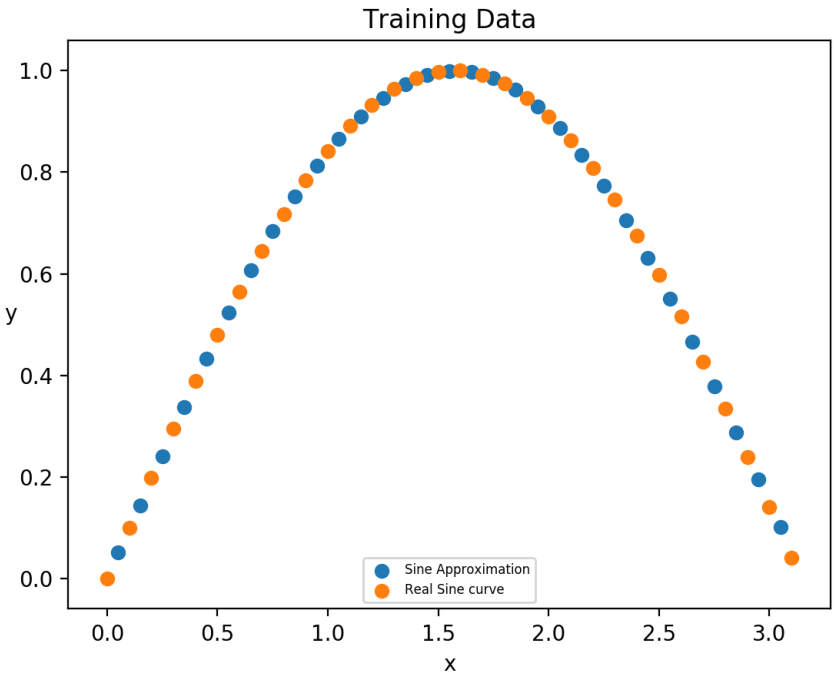


Abbildung 2.1

Wie man an dieser Grafik erkennen kann, sagt unser Modell den Sinus schon ziemlich genau hervor, was anhand der Evaluation schon erwartet werden konnte. Beim genaueren Hinsehen erkennt man, dass die Werte am Rand leicht verschoben sind. Wir haben im Allgemeinen bei allen Programmen, die wir insgesamt geschrieben haben, festgestellt, dass es an den Rändern immer ungenau wurde ($\rightarrow$ *Randwertproblematik*). Da wir aber auch den Fehler der Approximation abbilden wollten, erstellten wir einen zweiten Plot, der die Differenz unserer Approximation und den genauen Werten vom Sinus darstellt.

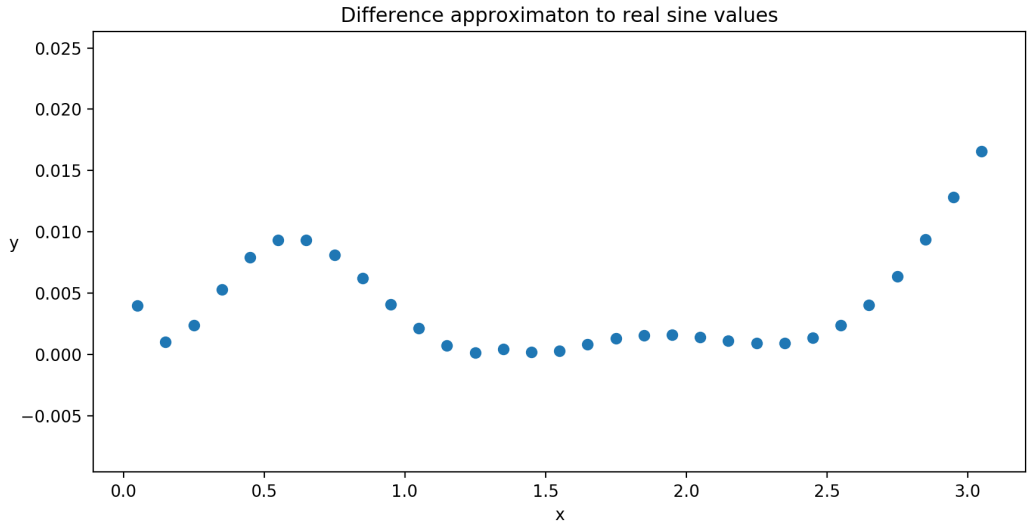


Abbildung 2.2

Hieran kann man sehr schön erkennen, dass es dort, wo die Werte des Sinus dichter sind, die Differenz niedriger ist und je näher man an die Grenzen des Intervalls gelangt (vor allem auf der rechten Seite), die Genauigkeit abnimmt.

Anmerkung: Die beiden Plots wurden erstellt, nachdem das Programm komplett optimiert wurde, d.h. nachdem auch einige weitere Optimierungen und Professionalisierungen vorgenommen wurden. Die Plots sahen aber auch nicht besonders anders aus, als wir diese Veränderungen noch nicht durchgeführt hatten.

Eine der ersten Sachen, die wir festgestellt hatten als wir die Kurven geplottet haben, war, dass es gelegentlich vorkommt, dass die Kurve total schief ist, was man auch daran gemerkt hat, dass es ab einem gewissen Zeitpunkt im Training zu keiner Verbesserung mehr kam. Auch im späteren Verlauf des Praktikums ist es immer wieder vorgekommen, dass in, grob geschätzt 5-10% der Fälle, die Approximation total daneben lag. Um dieses Problem zu umgehen, haben wir uns informiert, ob es eine Möglichkeit gibt, dies zu verhindern. Das Ergebnis unserer Recherche führte uns zu *Callbacks* und *Checkpoints*.

# Kapitel 3

# Approximation des Einheitskreises

## 3.1 Motivation zur Darstellung

Bevor wir über *Callbacks* und *Checkpoints* reden, die wir hier in diesem Kapitel behandeln werden, wollen wir erstmal erklären, wie wir den Einheitskreis darstellen möchten. Dazu haben wir uns zunächst verschiedene Darstellungsmöglichkeiten[1] angesehen. Letztendlich haben wir uns aufgrund der Erfahrungen der Sinus Approximation dazu entschieden den Einheitskreis per Sinus und Kosinus Funktionen auf einem Intervall $[0, 2\pi]$ zu parametrisieren.

## 3.2 Erstellen der Modelle und Datensets

Genau wie beim Sinus war es auch hier nötig, die Parameter der Modelle zu variieren, um ein Gefühl dafür zu bekommen, wie das Neuronale Netzwerk auf die Veränderungen reagiert. Die wichtigsten Parameter waren hierbei die Anzahl der Epochen und die Größe des Datensets. Da die Datensets und Modelle denen vom Sinus relativ ähnlich waren, wollen wir uns hier nur das Modell des Kosinus ansehen:

```
# dataset for the cosine model
xc, cos = shuffle(xc, cos, random_state=0)
xc_train, xc_temp, cos_train, cos_temp = train_test_split(xc, cos,
    test_size = 0.3)
xc_val, xc_test, cos_val, cos_test = train_test_split(xc_temp,
    cos_temp, test_size = 0.5)

# model of the cosine
modelCos = keras.Sequential([
    tf.keras.layers.Dense(512, activation='tanh', input_shape=(1,)),
    tf.keras.layers.Dense(256, activation='tanh'),
    tf.keras.layers.Dense(64, activation='tanh'),
    tf.keras.layers.Dense(1)
    ])
```

Listing 3.1

[1] https://de.wikipedia.org/wiki/Einheitskreis

Abgesehen davon, dass dies das Datenset vom Kosinus ist, gibt es hier keinen Unterschied zu dem Datenset des Sinus. Was wir aber verändern mussten, war die Größe der Layer: es gibt nun 512 Knoten im ersten Layer und 256 Knoten im zweiten Layer (*Listing 3.1, codeline 8f.*). Der Grund dafür ist, dass wir nun ein größeres Netzwerk brauchten, um die Werte genauer bestimmen zu können, da wir nun auch Werte im negativen Bereich betrachten mussten und die Funktionen jetzt mehrere Wendepunkte hatten. Was wir ebenfalls änderten war die Aktivierungsfunktion, dazu aber später mehr.

## 3.3 Callbacks und Checkpoints

Nun zurück zu *Callbacks* und *Checkpoints*. Der Gedanke, den wir hatten war, dass wir mit *Callbacks* verhindern, dass ein gerade trainierendes Modell im Laufe seines Trainings sich wieder verschlechtert. Mittels *Checkpoints* hatten wir die Möglichkeit die besten Gewichte des Netzwerks während des Trainings abzuspeichern und mit diesen Gewichten später weiter zu arbeiten. Diese beiden Parameter ins Programm zu integrieren war leider nicht ganz einfach, jedoch hatten wir auch hierfür ein sehr gutes Tutorial[2] gefunden, dass uns deutlich half, unsere Programme zu schreiben, da leider auch die Keras Dokumentation[3] dafür etwas schwierig nachzuvollziehen war. Unsere Implementierung von den *Callbacks* und den *Checkpoints* sah dann wie folgt aus:

```
1 # es is the callback function, mcSin and mcCos are the checkpoints
2 es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,
     patience=500)
3 filepathSin = 'weightsSin.best.hdf5'
4 mcSin = ModelCheckpoint(filepathSin, monitor='val_loss', mode='min',
      verbose=1, save_best_only=True)
5 filepathCos = 'weightsCos.best.hdf5'
6 mcCos = ModelCheckpoint(filepathCos, monitor='val_loss', mode='min',
      verbose=1, save_best_only=True)
```

Listing 3.2

FILEPATHSIN und FILEPATHCOS sind hier die Dateipfade, in denen die besten Gewichte gespeichert wurden.

Anmerkung: Es kann durchaus passieren, dass wir am Ende nicht wirklich mit den besten Gewichten arbeiten. Ursache dafür ist, dass die *Callback*-Funktion mit PATIENCE=500 das Training unterbricht, wenn sich das Programm nicht verbessert. Da aber das Training auch wieder schlechter werden kann, wenn es zu viele Epochen trainiert ($\rightarrow$ *Overfitting*), kann es passieren, dass wenn das Training wieder besser wird, neue Gewichte gespeichert werden, die nicht unbedingt besser sind als die vor der Verschlechterung.

[2] https://machinelearningmastery.com/check-point-deep-learning-models-keras/
[3] https://keras.io

Was wir auch merkten, war, dass es sich kaum lohnt *Callbacks* einzubauen, wenn es nur wenige Epochen gibt, da wir meistens erst dann sichtbare Erfolge mit den *Callbacks* und *Checkpoints* festmachen konnten, wenn wir 500 Epochen PATIENCE (*Listing 3.2, codeline 2*) eingestellt hatten. Das heißt wir gaben dem Modell beim Trainieren 5000 Epochen Zeit. (Viele Codes, die wir gesehen hatten, haben ca. 10 Mal mehr Epochen benutzt).

## 3.4 Testen verschiedener Aktivierungsfunktionen

Wie bereits angesprochen hatten wir zwischendurch die Aktivierungsfunktion des Modells von ACTIVATION='RELU' zu ACTIVATION='TANH' geändert. Natürlich hatten wir bereits am Anfang mit unterschiedlichen Aktivierungsfunktionen experimentiert, aber da der Sinus noch relativ simpel war, hatten wir keine qualitativen Unterschiede feststellen können. Keras bietet für diverse Probleme einige Aktivierungsfunktionen[4] an. Um herauszufinden, welche der Aktivierungsfunktionen mit den trigonometrischen Funktionen harmonieren, ließen wir unser Modell mit allen Aktivierungsfunktionen einmal laufen, um diese auszuschließen, die auf keinen Fall auf trigonometrische Funktionen anwendbar sind. Anschließend haben wir dann (mit mehr Epochen) die besten Aktivierungsfunktionen weiter getestet. Das Resultat war, dass TANH und RELU die beiden besten waren, wobei erstere noch einmal minimal besser war.

Anmerkung: RELU ist die deutlich "günstigere" Funktion. Wenn man aber eher auf Genauigkeit als auf Geschwindigkeit setzt, sollte man sich für TANH entscheiden. (Bei Programmen unserer Größe nimmt sich das allerdings wirklich nicht viel). Letztendlich haben wir uns für TANH entschieden.

## 3.5 Testen verschiedener Optimizer und loss-Funktionen

Was wir bisher noch nicht angesprochen haben, sind die *loss*- und *optimizer*-Funktionen, die das Programm beim Kompilieren nutzt. Nun wollen wir aber auch darauf eingehen, wir wir mit ihnen die Programme optimiert haben. Mit der *loss*-Funktion berechnet das Model, wie weit dessen Approximation von den tatsächlichen Werten entfernt ist. Daraufhin wird die Optimierungsfunktion herangezogen, um die Gewichte des Models zu verändern, um bei der nächsten Epoche ein geringeren *loss* zu erzielen. Auch hier bietet Keras eine Vielzahl von Funktionen[5] an, die je nach Model und Problemstellung mehr oder weniger Sinn machen. Da die Aufgabe der Approximation des Einheitskreises stark der Aufgabe mit dem Sinus ähnelt, konnten wir viele Annahmen und Überlegungen übernehmen. So fallen die *loss*- und *optimizer*-Funktionen raus, die eher auf Klassifizierungsprobleme zugeschnitten sind. Um die besten Funktionen aus den restlichen

[4] https://keras.io/activations/
[5] https://keras.io/losses/

Funktionen zu ermitteln, testeten wir mittels *Try-and-Error* alle Kombinationen durch. Hierbei fanden wir heraus, dass für uns die Optimierungsfunktion ADAM[6] und MEAN SQUARED ERROR am besten funktionieren. Das Kompilieren der Modelle für den Sinus und den Kosinus sahen dann wie folgt aus:

```
1 modelSin.compile(optimizer = tf.optimizers.Adam(), loss='
    mean_squared_error')
2 modelCos.compile(optimizer = tf.optimizers.Adam(), loss='
    mean_squared_error')
```

Listing 3.3

## 3.6 Ein Plot der Approximation

Genau wie beim Sinus haben wir uns auch hier immer wieder genauer angeschaut, was Veränderungen der vielen Parameter mit unseren Kurven machen. Wir haben festgestellt, dass der Kosinus sich interessanterweise "schlechter" approximieren lässt als der Sinus. Damit ist gemeint, dass es sehr häufig vorkam, dass bei der gleichen Anzahl an Epochen für die beiden Modelle der Sinus genauer dargestellt wurde als der Kosinus. Gerade an den Randwerten des Intervalls (auch hier wieder hauptsächlich auf der rechten Seite) wich die Näherung des Programms deutlich von den richtigen Werten des Kosinus ab, wie folgende Grafik zeigt:

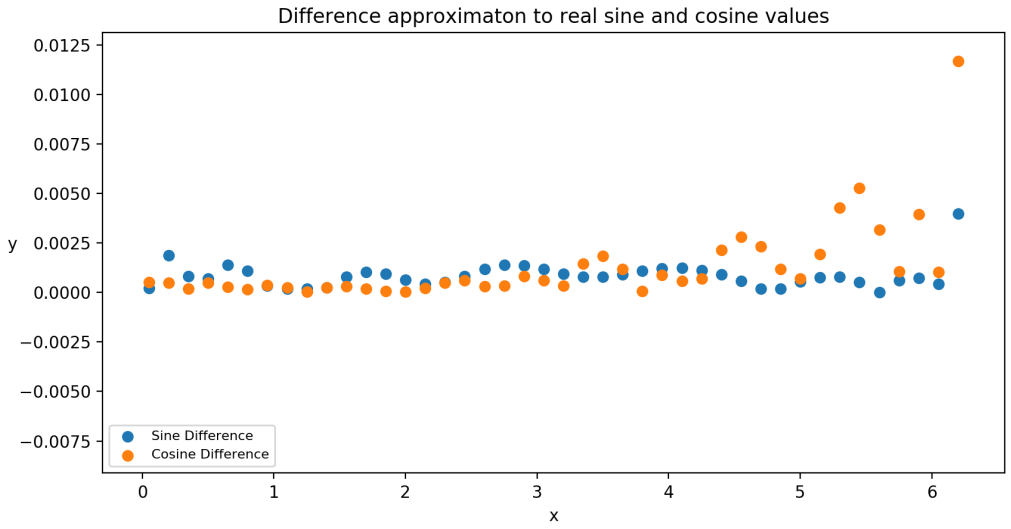


Abbildung 3.1

*Abbildung 3.1* zeigt sehr gut, dass zum einen die *Randwertproblematik* zum Vorschein kommt und zum anderen, dass der Kosinus schon ab $\pi$ ungenauer wird, während der

[6] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Sinus lange Zeit noch sehr gut angenähert wird. Nachdem wir alle Optimierungen in unser Programm für den Einheitskreis eingebunden haben, wollten wir am Ende auch eine Kurve plotten, die unsere Approximation gut darstellt:

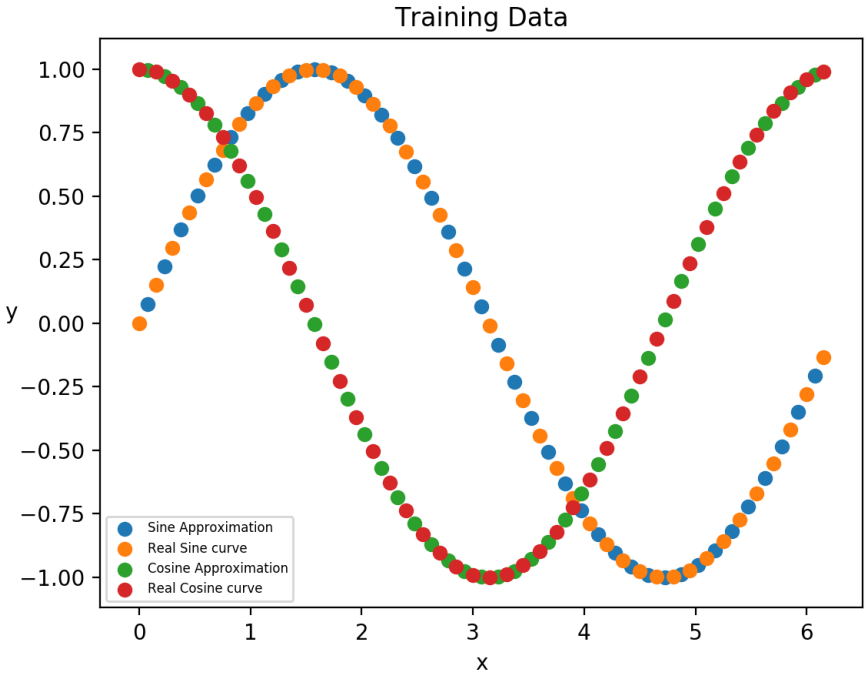


Abbildung 3.2

Anmerkung: Die in *Abbildung 3.2* dargestellten Kurven zeigen in diesem Fall eine sehr gute Approximation der beiden Kurven. Es kam aber auch oft genug vor, dass die Kurven nicht ganz so schön aussahen, wie *Abbildung 3.1* schon gezeigt hat.

# Kapitel 4

# Approximationen des Torus

## 4.1 Motivation zur Darstellung

Als wir uns an die dritte Aufgabe "Approximation des Torus" gesetzt hatten, stellten wir uns zuerst die Frage, wie wir einen 3-dimensionalen Körper approximieren wollen. In den ersten beiden Aufgaben hatten wir immer einen Eingabe- und einen Ausgabewert, was ohne Probleme funktionierte, nun jedoch benötigen wir drei Koordinaten, was uns vor eine Herausforderung stellte. Zudem hatten wir die Sorge, dass die Approximation eines 3-D-Objekts mit unseren bisher verwendeten Werkzeugen nicht exakt genug werden könnte. Bevor wir jedoch mit dem Schreiben des Codes anfingen, informierten wir uns erstmal über den Torus[1]. Wir hatten uns dazu entschieden, erstmal die Parametrisierung des Torus in kartesischen Koordinaten zu approximieren.

## 4.2 Parametrisierung in kartesischen Koordinaten

Betrachten wir zuerst mal die Umrechnung von Toruskoordinaten in kartesische Koordinaten:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \cdot \begin{pmatrix} \cos(t) \\ \sin(t) \\ 0 \end{pmatrix} + r \cdot \begin{pmatrix} \cos(t) \cdot \cos(p) \\ \sin(p) \cdot \cos(t) \\ \sin(p) \end{pmatrix} = \begin{pmatrix} (R + r \cdot \cos(p)) \cos(t) \\ (R + r \cdot \cos(p)) \sin(t) \\ r \cdot \sin(p) \end{pmatrix}$$

Wir mussten uns also als allererstes damit befassen, wie wir diese vier Parameter $R, r, p, t$ in unser Programm integrieren, wobei die Bedeutung der Parameter folgende ist:

- $R$ steht für den Abstand zwischen Kreismittelpunkt und Achse
- $r$ steht für den Radius des ursprünglichen Kreises
- $p$ steht für den Winkel im Torus
- $t$ steht für den Winkel für den Kreis um die Drehachse

Die beiden Winkel laufen jeweils von 0 bis $2\pi$. Zur Veranschaulichung des Ganzen hilft folgende Grafik[2]:

[1] https://de.wikipedia.org/wiki/Torus
[2] https://de.wikipedia.org/wiki/Torus#/media/Datei:Torus_3d.png

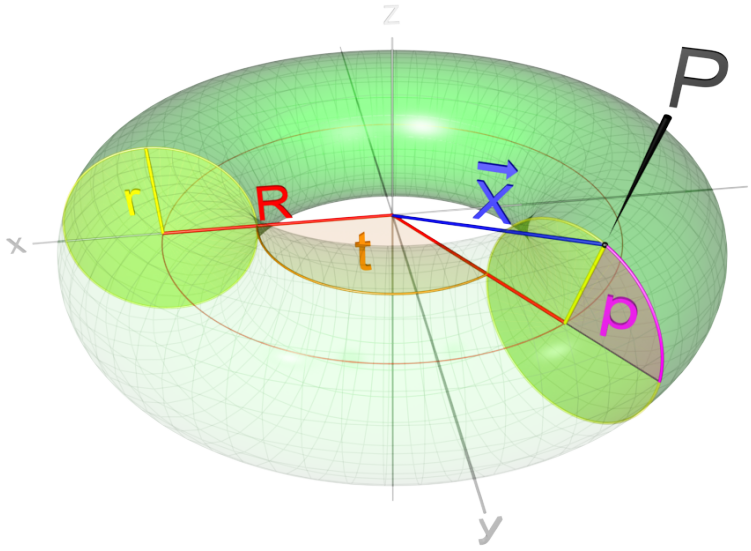


Abbildung 4.1

Einfachheitshalber hatten wir uns dazu entschieden den *Einheitstorus* zu benutzen, das heißt wir stellen uns vor, dass wir einen Einheitskreis haben, um den sich ein anderer Einheitskreis dreht. Daraus folgt, dass wir $R = 2$ und $r = 1$ setzen, also mussten wir nur noch die beiden Winkel zwischen 0 und $2\pi$ laufen lassen. Hier tauchte das erste Problem auf: Dadurch, dass wir beide Winkel $p$ und $t$ zwischen 0 und $2\pi$ liefen ließen, hatte unser Modell für jedes Paar $(x, y)$ zwei $z$ Werte, einen positiven und einen negativen, was zur Folge hatte, dass die Approximation fehlschlug. Das Programm hatte in diesem Fall für alle Paare $(x, y)$ einen $z$-Wert von ungefähr 0.5 approximiert. Um dieses Problem zu lösen haben wir uns dazu entschieden, nur einen halben Torus zu approximieren und damit $p$ nur zwischen 0 und $\pi$ laufen zu lassen, sodass wir nur einen Wert $z$ mit $z \geq 0$ approximiert haben. Durch diese Methode hatte unser Modell zwei Eingabewerte $(x, y)$ und nur noch einen Ausgabewert $z$. Nachdem wir das Problem mit den zwei $z$-Werten lösten, mussten wir noch eine kleine Änderung an unserem Programmcode vornehmen, um den Torus darstellen zu können. Entsprechend war es notwendig die Anzahl der Knoten weiter zu erhöhen, damit die Approximation besser wird. Unser Modell für den Torus sah dann wie folgt aus:

```
model = keras.Sequential([
    tf.keras.layers.Dense(2048, activation='relu', input_shape=(2,))
    ,
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1)
    ])
```

Listing 4.1

Ansonsten konnten wir mit unseren bisherigen Funktionen, die wir beim Einheitskreis benutzt haben, das Programm des Torus vervollständigen. Das heißt, es gab an dieser Stelle keine großen Änderungen bezüglich des Trainingsprozesses oder der anderen Parameter. Auch hier wollten wir unser Programm grafisch darstellen lassen, und erhielten folgenden Plot:

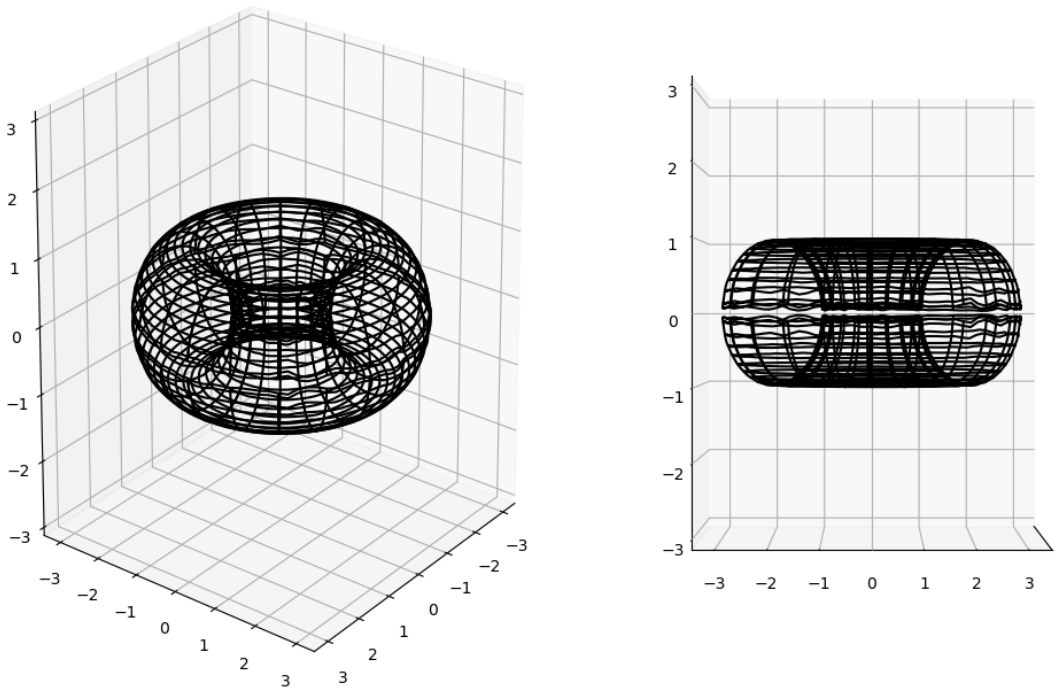


Abbildung 4.2

Wie man im rechten Bild von *Abbildung 4.2* sehr gut erkennen kann, hat unsere Approximation große Schwierigkeiten, wenn es um den Bereich um 0 herum geht. Eine Lösung dafür wäre den Torus nicht durch kartesische Koordinaten zu parametrisieren, sondern anhand zweier Winkel $\theta$ und $\phi$. Dazu mehr in *Abschnitt 4.6*.

## 4.3 Speichern der Gewichte - JSON

Eines der größten Probleme mit dem Arbeiten von *TensorFlow* am Anfang war, dass wir bei jeder kleinen Änderung der Visualisierung oder Parametrisierung das Programm neu laufen lassen mussten, was relativ zeitaufwendig und mühsam war. Deswegen hatten wir die Entscheidung getroffen, das Modell zu speichern. Dies hatte gewisse Vorteile:

- Als wir das Modell speicherten konnten wir ohne Probleme an einem anderen Zeitpunkt damit weiterarbeiten, ohne das Neuronale Netzwerk neu trainieren zu müssen

- Wollten wir etwas an der Visualisierung des Torus verändern, mussten wir nun nicht mehr das ganze Modell neu trainieren.

Ein Modell ist herunter gebrochen nur eine Ansammlung von Knoten, Kanten und Gewichten. Dadurch ist das Speichern eines Models relativ einfach, daher hatten wir das Modell als *JSON*-Datei gespeichert. Diese Datei beinhaltet die groben Eigenschaften unseres Modells, wie die Anzahl der Layer und deren Charakteristiken (den Input, den sie erwarten) und die Anzahl von Neuronen. Die einzelnen Kanten- und Knotengewichte speicherten wir in einer separaten .HDF5-Datei. Diese .HDF5-Datei wird beim Training an der Stelle gespeichert, an der das Model sich nicht mehr verbessert (vgl. *Listing 3.2*: FILEPATH...). Für den Torus sah der Code für das Abspeichern in eine *JSON*-Datei dann wie folgt aus:

```
# serialize torus model to JSON
torus_json = modelTorus.to_json()
with open('modelTorus.json', 'w') as json_file:
    json_file.write(torus_json)
print('Saved the model to disk')
```

Listing 4.2

## 4.4 Das beste Modell

Wir waren an dieser Stelle also an einem Punkt, an dem wir unsere Modelle mitsamt den Gewichten extern abgespeichert hatten. Wir haben aber bei einigen Tutorials[3] gesehen, dass viele diese, nachdem sie ihre Modelle abgespeichert haben, nochmal neu geladen und kompiliert haben. Deshalb haben wir ebenfalls die Gewichte nochmal neu geladen. Das sah dann wie folgt aus:

```
# create a new torus model with the best weights
bestTorus = keras.Sequential([
    tf.keras.layers.Dense(2048, activation='relu', input_shape=(2,))
    ,
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1)
    ])

bestTorus.load_weights('weightsTorus.best.hdf5')
bestTorus.compile(optimizer=tf.optimizers.Adam(), loss='
    mean_squared_error')
resBestTorus = bestTorus.evaluate(xy_test, z_test, verbose=0)
```

Listing 4.3

[3] z.B: https://machinelearningmastery.com/save-load-keras-deep-learning-models/

bestTorus ist hier das neue Modell mit den vorher trainierten Gewichten. Auch das wird anschließend nochmal kompiliert und evaluiert. Da aber die Gewichte dieselben sind wie vorher, mussten wir das Ganze nicht mehr trainieren.

Anmerkung: Man würde vermuten, dass wenn man sich resBestTorus ausgeben lässt, man dieselben Ergebnisse bekommt, wie beim Torus-Modell, das man in die *JSON*-Datei geschrieben hatte. Interessanterweise sorgt das erneute Kompilieren aber dafür, dass sich irgendwas innerhalb des Neuronalen Netzwerks ändert. Eine Verschlechterung des Modells liegt dadurch aber nicht vor.

## 4.5 Reader Programme

Durch unsere Herangehensweise bei der Erstellung eines Modells, konnten wir die Gewichte des Modells mit den bestmöglichen Gewichten speichern. Jedoch wollten wir nicht jedes Mal die ganzen Programme neu ausführen, deshalb haben wir uns für das Kreieren eines "besten Modells" neue Programme geschrieben. Daraufhin haben wir sowohl für den Sinus, den Einheitskreis, als auch für den Torus *Reader* geschrieben, mit denen wir dann die Gewichte eingelesen und in einem besten Modell gespeichert haben. Dabei war es wichtig, das Model aus der *JSON*-Datei zu laden:

```
# load torus model from JSON file and create Torus model with best
    weights
json_file = open('bestTorus.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
bestTorus = tf.keras.models.model_from_json(loaded_model_json)
bestTorus.load_weights("weightsTorus.best.hdf5")
print("Loaded model from disk")
```

Listing 4.4

Der Vorteil der *Reader*-Programme ist, dass wir nur einmal trainieren müssen bis wir sehr gute Gewichte haben und dann für verschiedene Plots nur die *Reader*-Programme ausführen müssen. Daraufhin haben wir den Code für die Plots und das Kreieren eines besten Modells in die *Reader*-Dateien verschoben, so dass das Trainieren und das Arbeiten mit den Modells separat ablief.

## 4.6 Parametrisierung anhand der Winkel $\theta$ und $\phi$

Durch die Parametrisierung in kartesischen Koordinaten war unsere Approximation in unserem Ermessen gut, aber nicht sehr gut; wir hatten ein sehr großes Problem mit den $z$-Werten, die nahe der 0 waren (siehe *Abbildung 4.2*). Bei unserem Treffen mit Prof. Guido Kanschat wurde uns empfohlen statt einem Wertepaar $(x, y)$ als Input und $z$ als Output, lieber die Winkel $\theta$ und $\phi$ als Input und das Tripel $(x, y, z)$ als Output zu

verwenden. Dies resultierte darin, dass die Approximation viel besser wurde, was man an folgender Grafik hervorragend sehen kann:

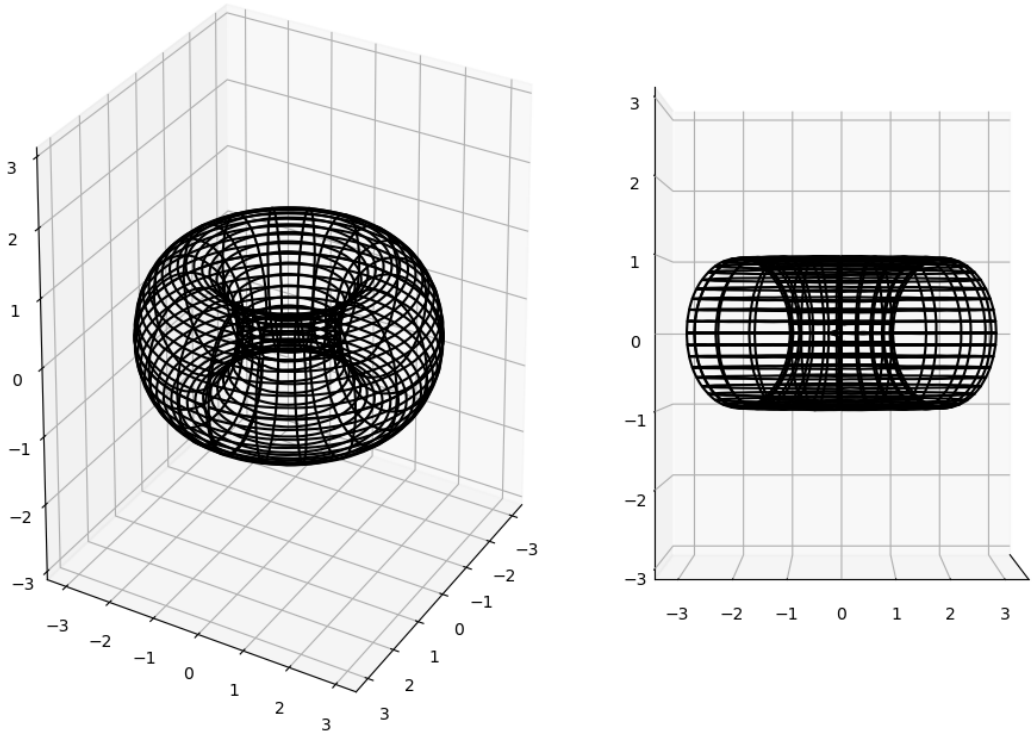


Abbildung 4.3

Das Problem mit den $z$-Werten in der Nullumgebung wurde damit behoben. Wir hatten es also geschafft auch für den Torus mit Hilfe unseres Modells einen schönen Plot zu erstellen, obwohl wir anfangs ein paar Zweifel hatten, ob wir das gut hinbekommen würden.

Anmerkung: Dadurch mussten wir uns für den *Reader* für das neue Torus Modell noch überlegen, wie genau wir das visualisieren wollen. Da das Model als Eingabe einen Wert für $\theta$ und für $\phi$ erwartete, mussten wir diese mittels *numpy* generieren. Dabei hatten wir uns für 10000 $(\theta, \phi)$-Wertepaare entschieden, da man dadurch genug Werte hat, um den Torus zu visualisieren, aber trotzdem nicht zu lange rechnen muss. Nachdem wir das gemacht hatten, mussten wir für die 10000 $(\theta, \phi)$-Wertepaare die *TensorFlow* Funktion PREDICT() verwenden, die uns anhand $\theta$ und $\phi$ das Tripel $(x, y, z)$ für den Torus ausgab. Hatten wir nun alle $x, y$ und $z$ Werte gespeichert, konnten wir mittels der *matplotlib*-Bibliothek diese Werte visualisieren. Dabei war es wichtig eine klare Darstellung zu finden, die auch gut zeigen konnte, wenn Imperfektionen durch ein fehlerhaftes Model entstehen. Hierbei entschieden wir uns für eine *Wireframe*-Oberfläche, bei der man schnell Fehler und Ungenauigkeiten erkennen konnte. Analog haben wir diese Visualisierungsmethode auch auf die Parametrisierung des Torus mit kartesischen Koordinaten angewendet, wie man in *Abbildung 4.2* sehen kann.

# Kapitel 5

# Fazit und persönliche Erfahrungen

Zusammenfassend bleibt zu sagen, dass das Projekt sehr lehrreich war. Es war ein angenehmer Einstieg in das Feld von *Machine Learning*, obwohl es an manchen Stellen eine echte Herausforderung war. Auch wenn das Interesse an *Machine Learning* schon länger da war, konnten wir uns vor dem Praktikum noch nicht genau vorstellen, wie das im Detail funktioniert. Nach unserem allerersten Treffen waren wir uns nicht sicher, ob wir es schaffen werden uns in *Machine Learning* und *TensorFlow* einzuarbeiten, da die offizielle Dokumentation von *TensorFlow* sehr spärlich und die Anzahl der Tutorials relativ klein war. Zum Beispiel halfen die Fehlermeldungen von *TensorFlow* auch nicht, da wenn man nach dieser gesucht hat, keine vernünftige Lösung gefunden hat. Nichts desto trotz waren wir der festen Überzeugung, dass wir diese Herausforderung zu dritt meistern können. Um die drei erhaltenen Aufgaben, (die Approximation des Sinus, des Einheitskreises und des Torus) zu bearbeiten, hatten wir uns immer freitags in einem Arbeitsraum der Universitätsbibliothek getroffen. Das Arbeiten in der Gruppe war eine sehr positive Erfahrung, da man hier die Möglichkeit hatte zusammen an Problemen zu arbeiten, was es zu einer angenehmen Erfahrung machte. So schauten immer mehrere Augen auf ein Problem und man war nicht auf sich alleine gestellt. Genauso gut konnte man aber Aufgabenteile auch aufteilen, so dass jeder sich auf einen Part spezialisieren konnte.

Trotz aller Hürden und Schwierigkeiten sind wir der Meinung, dass wir durch das Arbeiten im Team, alle Aufgaben erfolgreich abgeschlossen und einiges dabei gelernt haben, was uns sicherlich einen guten Einstieg ins Thema *Machine Learning* und eine gute Grundlage für folgende Vorlesungen bezüglich dieser Thematik ermöglicht hat. Während wir angefangen haben die ersten Konzepte zu verstehen und die diversen Anwendungsgebiete von *Machine Learning* kennenlernten, konnten wir uns immer mehr für das Thema begeistern. Spätestens nachdem wir festgestellt haben, wie mächtig *Machine Learning* seien kann, kam uns ernsthaft der Gedanke in der Zukunft weiter in dem Gebiet zu arbeiten.

# Kapitel 6

# Quellen


[1] https://machinelearningmastery.com/save-load-keras-deep-learning-models/

[2] https://machinelearningmastery.com/check-point-deep-learning-models-keras

[3] https://www.tensorflow.org

[4] https://www.tensorflow.org/tutorials/distribute/keras

[5] https://keras.io

[6] https://de.wikipedia.org/wiki/Einheitskreis#Rationale_Parametrisierung

[7] https://de.wikipedia.org/wiki/Torus

[8] https://machinelearningmastery.com

[9] https://scikit-learn.org/stable/modules/generated/sklearn.utils.shuffle.html

[10] https://scikit-learn.org/stable/index.html

[11] https://www.youtube.com/watch?time_continue=412&v=KNAWp2S3w94

[12] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[13] https://www.youtube.com/watch?v=aircAruvnKk

[14] http://neuralnetworksanddeeplearning.com/

[15] https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss

[16] https://datascience.stackexchange.com/questions/19365/predict-sinus-with-keras-feed-forward-neural-network/19399