lorenz

April 14, 2021

1 Building a simple model with KERAS for the Lorenz System

The Lorenz system is a harmless looking system of ODEs, describing the trajectory of a point in x,y,z-space. It is however infamous, as it can have extremely erratic dynamics. Over a wide range of parameters, the solution oscillates irregularly, but never repeats itself. Remarkably, it stays in a bounded region in phase space, a so-called strange attractor - a fractal with a dimension between 2 and 3.

It is a prototype for non-linear dynamical systems, and we will use it to show two things in this notebook:

a) How simple it is to build an NN using KERAS

b) How NNs can learn complex dynamics

Note: There are no graded parts in this notebook, but you should look around still and explore!

1.1 Solving the Lorenz System with an ODE solver

Here, we show the Lorenz system, its typical solutions and solve the ODE the classical way - with a numerical integrator / quadrature from python. This serves as a generator for the training data. We will then try to build a NN to lear the dynamics of the system:

Given (x,y,z) at timestep t as an input, predict the next state (x,y,z) at timestep t+1.

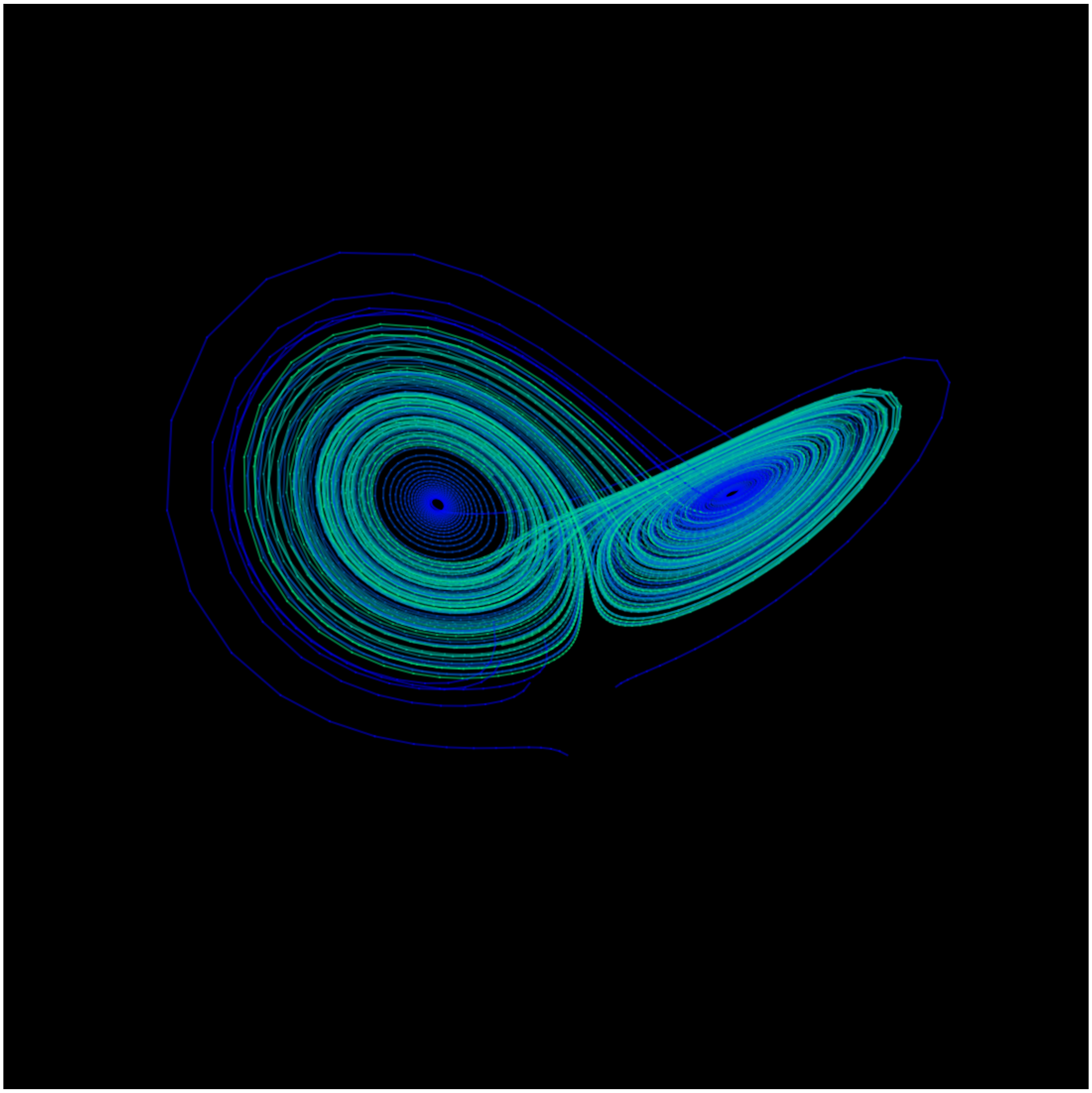
Thus, we can generate the training data for the NN by shifting the solution by one timestep. Since we want to learn the dynamics and avoid overfitting, we will use many runs with different initial conditions.

[1]:
```
# See also the blog on https://scipython.com/blog/the-lorenz-attractor/
# Christian Hill, January 2016 as well as the website http://www.databookuw.com/
↪



%matplotlib inline
import matplotlib
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import pylab
img_width, img_heigth, img_dpi = 2000, 1500, 100


# Lorenz parameters and initial conditions, changed somewhat from the standard␣
↪set
sigma, beta, rho = 10, 2.667, 32
(x0, y0, z0) = (2, 2, 2)

# define the ODE system
def lorenz_system(t, X, sigma, beta, rho):
    x, y, z = X
    dxdt = -sigma*(x - y)
    dydt = rho*x - y - x*z
    dzdt = -beta*z + x*y
    return dxdt, dydt, dzdt


# t_end and the number of timesteps and runs of the system
tend, n, runs = 32, 2000, 100
t = np.linspace(0, tend, n)

# solution, 3 coordinates, timesteps, runs
solution=np.zeros((3, n,runs))


for run in range(0,runs):
# Integrate the Lorenz equations with the ODE solver of python
    print("solving Lorenz system no. ",run,end='\r')
    # set up the solver, default is RungeKutta 45
    solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma,␣
↪beta, rho),dense_output=True)
    solution[:,:,run] = solver.sol(t)
    # randomize initial condition
    (x0, y0, z0) = 20*(np.random.rand(3)-0.5)


# Plot some of the runs
printfig=1
s, k = 1, 15 # s: plot every sth step, k: plot every kth run
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
```

```
    cmap = plt.cm.winter
    for run in range(0,runs-k,k):
        for i in range(0,n-s,s):

            #vals = np.linspace(0,1,256)
            #np.random.shuffle(vals)
            #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

            ax.plot(solution[0,i:i+s+1,run], solution[1,i:i+s+1,run],␣
 ↪solution[2,i:i+s+1,run], color=cmap(i/n), alpha=0.5,linewidth=2.0)
    ax.set_axis_off()
    plt.show()
```

```
solving Lorenz system no.  99
Preparing plots
```



2 Solving the Lorenz System with an ANN

It's much easier using a package for creating and using a neural network. *Keras* was developed as a high-level API that supports multiple backends. Recently, it was incorporated into the *TensorFlow* framework from Google. Roughly, the pipeline for an ANN project is

- Define the model.
- Compile the model.
- Fit the model.
- Evaluate the model.
- Make predictions.

[ ]:

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.utils import shuffle

# keras is part of TensorFlow, so importing things is done via tensorflow
# Sequential is used for creating sequential models, that are built layer
# by layer as seen below
# actions in such a model are created by adding layers to a model. Dense
# layers are normal neural layers which we've seen in the lecture

# the following function clears a keras session, such that all old models
# get deleted
from tensorflow.keras.backend import clear_session

clear_session()

# We want to learn the mapping from X_t -> X_(t+1), so we shift the outputs Y␣
↪by one
X=solution[:,0:solution.shape[1]-1,:]
Y=solution[:,1:solution.shape[1],:]
X=np.transpose(X.reshape(3,-1))
Y=np.transpose(Y.reshape(3,-1))

# Shuffling is important to avoid bias in the test set
X, Y = shuffle(X,Y)
input_shape = (3,)

# Here we build the KERAS model - it should be self explanatory!
# Layers are added just by stacking them as shown below
# "Dense" adds a fully connected layer of neurons. Activation is given
# for all neurons simultaneously, although there are other options.
# input_shape in the first layer determines the shape that is
# expected from the input array.


model = keras.Sequential(
    [
        layers.Dense(3, input_shape=input_shape, name="input"),
        layers.Dense(20, activation="relu", name="layer2"),
        layers.Dense(20, activation="relu", name="layer3"),
        layers.Dense(20, activation="relu", name="layer4"),
        layers.Dense(3, name="output"),
    ]
)
```

```python
# This is so called "call back". It is used to influence the model during the␣
↪training stage. Here, we use it
# to introduce a decay of the learning rate lr

def lr_scheduler(epoch, lr):
    decay_rate = 0.99
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr

# hook the call backs up
callbacks = [
    keras.callbacks.LearningRateScheduler(lr_scheduler, verbose=1)
]
# We now compile the model. You can set the loss function, the optimizer and␣
↪many other things here
model.compile(loss='mean_absolute_error', optimizer='adam',␣
↪metrics=['mean_squared_error'])

# The next line starts the training. We do not need to take care about␣
↪backpropagation at all.
model.fit(X, Y, epochs=20, verbose=1, validation_split=0.2,callbacks=callbacks)
print(model.summary())
```

```
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:517: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:518: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing
```

```
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.7/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])

WARNING:tensorflow:From /usr/local/lib/python3.7/dist-
packages/tensorflow/python/ops/init_ops.py:1251: calling
VarianceScaling.__init__ (from tensorflow.python.ops.init_ops) with dtype is
deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
constructor
Train on 159920 samples, validate on 39980 samples
```

```
Epoch 00001: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 1/20
159920/159920 [==============================] - 18s 113us/sample - loss: 0.5197
- mean_squared_error: 5.2149 - val_loss: 0.1515 - val_mean_squared_error: 0.0806

Epoch 00002: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 2/20
159920/159920 [==============================] - 18s 112us/sample - loss: 0.1531
- mean_squared_error: 0.0803 - val_loss: 0.1465 - val_mean_squared_error: 0.0554

Epoch 00003: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 3/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.1217
- mean_squared_error: 0.0482 - val_loss: 0.1138 - val_mean_squared_error: 0.0369

Epoch 00004: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 4/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.1058
- mean_squared_error: 0.0360 - val_loss: 0.0994 - val_mean_squared_error: 0.0321

Epoch 00005: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 5/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0953
- mean_squared_error: 0.0296 - val_loss: 0.1038 - val_mean_squared_error: 0.0285

Epoch 00006: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 6/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0899
- mean_squared_error: 0.0261 - val_loss: 0.0702 - val_mean_squared_error: 0.0190

Epoch 00007: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 7/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0857
- mean_squared_error: 0.0237 - val_loss: 0.0861 - val_mean_squared_error: 0.0220

Epoch 00008: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 8/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0818
- mean_squared_error: 0.0215 - val_loss: 0.0787 - val_mean_squared_error: 0.0190
```

```
Epoch 00009: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 9/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0807
- mean_squared_error: 0.0204 - val_loss: 0.0895 - val_mean_squared_error: 0.0219

Epoch 00010: LearningRateScheduler reducing learning rate to
0.0010000000474974513.
Epoch 10/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0778
- mean_squared_error: 0.0189 - val_loss: 0.0713 - val_mean_squared_error: 0.0169

Epoch 00011: LearningRateScheduler reducing learning rate to
0.0009900000470224768.
Epoch 11/20
159920/159920 [==============================] - 17s 109us/sample - loss: 0.0752
- mean_squared_error: 0.0178 - val_loss: 0.0872 - val_mean_squared_error: 0.0226

Epoch 00012: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 12/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0719
- mean_squared_error: 0.0162 - val_loss: 0.0712 - val_mean_squared_error: 0.0152

Epoch 00013: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 13/20
159920/159920 [==============================] - 17s 108us/sample - loss: 0.0699
- mean_squared_error: 0.0152 - val_loss: 0.0840 - val_mean_squared_error: 0.0172

Epoch 00014: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 14/20
159920/159920 [==============================] - 17s 109us/sample - loss: 0.0666
- mean_squared_error: 0.0142 - val_loss: 0.0648 - val_mean_squared_error: 0.0122

Epoch 00015: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 15/20
159920/159920 [==============================] - 18s 110us/sample - loss: 0.0656
- mean_squared_error: 0.0138 - val_loss: 0.0670 - val_mean_squared_error: 0.0124

Epoch 00016: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 16/20
159920/159920 [==============================] - 18s 111us/sample - loss: 0.0639
- mean_squared_error: 0.0133 - val_loss: 0.0503 - val_mean_squared_error: 0.0100
```

```
Epoch 00017: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 17/20
159920/159920 [==============================] - 17s 107us/sample - loss: 0.0622
- mean_squared_error: 0.0127 - val_loss: 0.0540 - val_mean_squared_error: 0.0120

Epoch 00018: LearningRateScheduler reducing learning rate to
0.0009900000877678394.
Epoch 18/20
148064/159920 [==========================>…] - ETA: 1s - loss: 0.0625 -
mean_squared_error: 0.0126
```

3 Evaluate model on test run

We evalute the model using model.predict. Here, we generate a new test run from random initial conditions, solve for the exact solution with the ODE solver and then apply the NN. Note that since the NN has make predictions one step at a time, this is quite a lot slower than the training, where we could put in many samples at the same time However, here we want the net to make iterative predictions starting just from x0,y0,z0, so we are stuck with this approach:

X0-> X1=model(X0) -> X2=model(X1) …

[ ]:
```
# Generate test run
# randomize initial condition
(x0, y0, z0) = 20*(np.random.rand(3)-0.5)
solver = solve_ivp(lorenz_system, (0, tend), (x0, y0, z0), args=(sigma, beta,␣
 ↪rho),dense_output=True)
test_solution=np.zeros((3, n))
test_solution[:,:] = solver.sol(t)
inp=np.array([[x0, y0, z0]])
predout=np.zeros((3, n))

for i in range(0,n):
    if (i % k) ==0: print("Prediction of time step ",i," of ",n,end='\r')
    predout[:,i]=model.predict(inp)
    inp=np.array(predout[:,i])[None,:]



s=1
print("\nPreparing plots \n")
if printfig:
    fig = plt.figure(facecolor='k', figsize=(img_width/img_dpi, img_heigth/
 ↪img_dpi))
    ax = fig.gca(projection='3d')
    ax.set_facecolor('k')
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    cmap = plt.cm.summer
```

```
    for i in range(0,n-s,s):

        #vals = np.linspace(0,1,256)
        #np.random.shuffle(vals)
        #cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

        ax.plot(test_solution[0,i:i+s+1], test_solution[1,i:i+s+1],␣
↪test_solution[2,i:i+s+1], color=cmap(1), alpha=0.4,linewidth=3.0)
        ax.plot(predout[0,i:i+s+1], predout[1,i:i+s+1], predout[2,i:i+s+1],␣
↪color='red', alpha=0.4,linewidth=3.0)
    ax.set_axis_off()
    plt.show()
```

4 Evalute on x-component of test run

[ ]:
```
print(test_solution[0,:])
print(predout[0,:])
fig = plt.figure(facecolor='w', figsize=(img_width/img_dpi, img_heigth/img_dpi))
plt.plot(t,test_solution[0,:])
plt.plot(t,predout[0,:])
```

5 Things to investigate / think about

- Build more complex networks: Which one works better: tall and skinny or short and fat ones?
- Train a simpler ODE! Look at the Lorenz system and think about how to make it less sensitive (remove / weaken nonlinearities)
- Effects of activation functions, optimizers, metrics, …
- Which improves the training: more runs of the Lorenz system, or more steps per run (runs vs. n?)
- Influence of timestep size; can a model trained on delta_t work on 2 delta_t?
- Getting more familiar with KERAS
- Sequence methods for temporal data
- Physics informed NNs
- We have essentially replace the ODE solver of python with an NN. Are you impressed with the speed and accuracy of the NN, or not so much?

[ ]:
```
#model.save("model_vall2_0.0029")
#reconstructed_model = keras.models.load_model("model_vall2_0.0023")
```