Brunton and Kutz Problem 6.1 part d: Lorenz System Prediction

Source Filename: /main.py

Rico A. R. Picone

This is the solution for Brunton and Kutz (2022), exercise 6.1, part d regarding the Lorenz equations. Only the case is considered. First, import the necessary libraries:

import numpy as np  
import matplotlib.pyplot as plt  
# \_\_import\_\_("matplotlib").use("TkAgg") # Use this to rotate the 3D plot  
# However, it doesn't publish well.  
from scipy import integrate  
from mpl\_toolkits.mplot3d import Axes3D  
import keras  
from keras.models import Sequential  
from keras.layers import Dense, Input  
from keras import optimizers  
from keras.layers import Activation  
from keras import backend as K

Set script options:

regenerate\_data = True # Regenerate the training data  
retrain = True # Retrain the model

Define the Lorenz equations:

def lorenz(x\_, t, sigma=10, beta=8/3, rho=28):  
 """  
 Lorenz equations dynamics (dx/dt, dy/dt, dz/dt)  
 """  
 x, y, z = x\_  
 dx = sigma \* (y - x)  
 dy = x \* (rho - z) - y  
 dz = x \* y - beta \* z  
 return [dx, dy, dz]

Define a function to generate the training data by numerically solving the Lorenz equations for a given initial condition:

def generate\_data(n\_samples, n\_timesteps, dt, sigma=10, beta=8/3, rho=28):  
 """  
 Generate training data for the Lorenz equations  
 """  
 t = np.linspace(0, (n\_timesteps-1)\*dt, n\_timesteps) # Time array  
 x = np.zeros((n\_samples, n\_timesteps, 3)) # Array to store the data  
 for i in range(n\_samples):  
 np.random.seed(i) # For reproducibility  
 x0 = np.random.uniform(-15, 15, 3) # Random initial condition  
 x[i] = integrate.odeint(  
 lorenz, # Dynamics to integrate  
 x0, # Initial condition  
 t, # Time array  
 args=(sigma, beta, rho) # Parameters for the Lorenz equations  
 )  
  
 return x

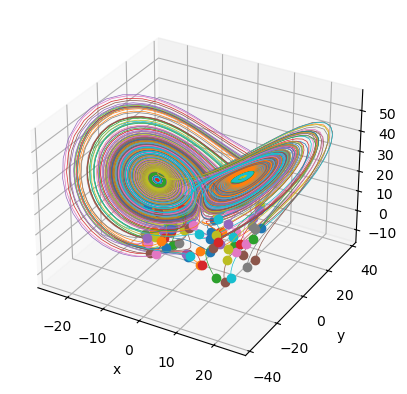
Generate the training data:

n\_samples = 100 # Number of samples  
n\_t = 1000 # Number of time steps  
dt = 0.01 # Time step  
if regenerate\_data:  
 data = generate\_data(n\_samples, n\_t, dt)  
 np.save('training-data.npy', data)  
else:  
 data = np.load('training-data.npy')

Plot the integrated trajectories of the Lorenz variables:

fig = plt.figure()  
ax = fig.add\_subplot(111, projection='3d')  
for i in range(n\_samples):  
 ax.plot(data[i, :, 0], data[i, :, 1], data[i, :, 2], lw=0.5)  
 ax.plot(  
 data[i, 0, 0], data[i, 0, 1], data[i, 0, 2],   
 lw=0.5, marker='o', color=ax.lines[-1].get\_color()  
 )  
ax.set\_xlabel('x')  
ax.set\_ylabel('y')  
ax.set\_zlabel('z')  
# plt.show()

Text(0.5, 0, 'z')



Transform the data into a format suitable for training a neural network. The input to the network will be the states of the Lorenz variables at time and the output will be the states at time . The samples are concatenated along the first axis:

X = np.zeros(((n\_t-1)\*n\_samples, 3))  
Y = np.zeros(((n\_t-1)\*n\_samples, 3))  
for i in range(n\_samples):  
 X[i\*(n\_t-1):(i+1)\*(n\_t-1)] = data[i, :-1]  
 Y[i\*(n\_t-1):(i+1)\*(n\_t-1)] = data[i, 1:]

Define the neural network architecture:

def build\_model():  
 """  
 Build the feedforward neural network model  
 """  
 model = Sequential()  
 model.add(Input(shape=(3,)))  
 model.add(Dense(10))  
 model.add(Activation('relu'))  
 model.add(Dense(10))  
 model.add(Activation('relu'))  
 model.add(Dense(10))  
 model.add(Activation('relu'))  
 model.add(Dense(3))  
 return model

Compile the model:

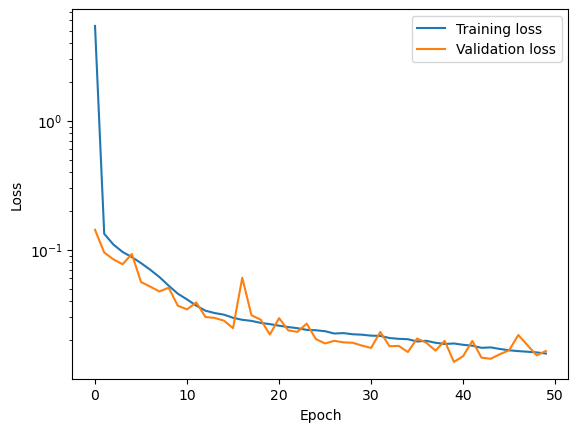
model = build\_model()  
model.compile(  
 optimizer=optimizers.Adam(learning\_rate=0.001),  
 loss='mean\_squared\_error', # Loss function  
 metrics=['mean\_absolute\_error'], # Metrics to monitor  
)

Train the model:

if retrain:  
 history = model.fit(  
 X, # Input data  
 Y, # Target data  
 epochs=50, # Number of epochs  
 batch\_size=32, # Batch size  
 validation\_split=0.2, # Validation split  
 shuffle=True, # Shuffle the data  
 )  
 model.save('model.keras')  
 history = True  
else:  
 model = keras.models.load\_model('model.keras')  
 history = False

Plot the training and validation loss versus the epoch:

if history:  
 fig, ax = plt.subplots()  
 ax.set\_yscale('log')  
 ax.plot(model.history.history['loss'], label='Training loss')  
 ax.plot(model.history.history['val\_loss'], label='Validation loss')  
 ax.set\_xlabel('Epoch')  
 ax.set\_ylabel('Loss')  
 ax.legend()  
 # plt.show()



Generate new test trajectories using the trained model:

n\_test\_samples = 20 # Number of test samples  
if regenerate\_data:  
 data\_test = generate\_data(n\_samples, n\_t, dt)  
 np.save('test-data.npy', data\_test)  
else:  
 data\_test = np.load('test-data.npy')

Transform the data into a format suitable for the neural network:

X\_test = np.zeros(((n\_t-1)\*n\_test\_samples, 3))  
Y\_test = np.zeros(((n\_t-1)\*n\_test\_samples, 3))  
for i in range(n\_test\_samples):  
 X\_test[i\*(n\_t-1):(i+1)\*(n\_t-1)] = data\_test[i, :-1]  
 Y\_test[i\*(n\_t-1):(i+1)\*(n\_t-1)] = data\_test[i, 1:]

Predict the next state using the trained model:

Y\_pred = model.predict(X\_test)

1/625 ━━━━━━━━━━━━━━━━━━━━ 50s 80ms/step

89/625 ━━━━━━━━━━━━━━━━━━━━ 0s 580us/step

146/625 ━━━━━━━━━━━━━━━━━━━━ 0s 699us/step

277/625 ━━━━━━━━━━━━━━━━━━━━ 0s 549us/step

406/625 ━━━━━━━━━━━━━━━━━━━━ 0s 498us/step

530/625 ━━━━━━━━━━━━━━━━━━━━ 0s 476us/step

625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 494us/step

625/625 ━━━━━━━━━━━━━━━━━━━━ 0s 495us/step

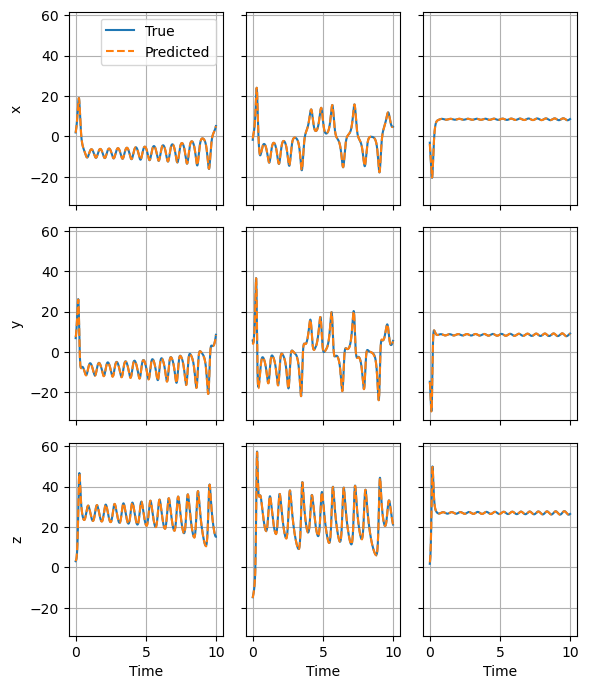
Compute the mean absolute error (MAE) between the predicted and true states:

mae = np.mean(np.abs(Y\_test - Y\_pred))  
print(f'Mean absolute error (MAE) for test trajectories: {mae}')

Mean absolute error (MAE) for test trajectories: 0.07603642006661351

Plot the x, y, and z coordinates of the true and predicted trajectories for 3 test samples:

t = np.linspace(0, (n\_t-1)\*dt, n\_t) # Time array  
labels = ['x', 'y', 'z']  
fig, axs = plt.subplots(3, 3, figsize=(6, 7), sharex=True, sharey=True)  
for i in range(3):  
 for j in range(3):  
 axs[j, i].plot(  
 t[:-1], Y\_test[i\*(n\_t-1):(i+1)\*(n\_t-1), j], label='True'  
 )  
 axs[j, i].plot(  
 t[:-1], Y\_pred[i\*(n\_t-1):(i+1)\*(n\_t-1), j], label='Predicted',   
 linestyle='--'  
 )  
 axs[j, i].grid()   
 if i == 0:  
 axs[j, i].set\_ylabel(labels[j])  
 axs[0, i].legend()  
 axs[2, i].set\_xlabel('Time')  
plt.tight\_layout()  
plt.show()



We achieve excellent agreement between the true (numerically integrated) and predicted trajectories, even for lobe transitions.