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 $\rho = 28$ case is considered. First, import the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
from mpl_toolkits.mplot3d import Axes3D
import keras
from keras.models import Sequential
from keras.layers import Dense, Input, Activation
from keras import optimizers
```

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,
seed_offset=0):
    """
    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+seed_offset)  # For reproducibility
        x0 = np.random.uniform(-15, 15, 3)  # Random initial condition
        x[i] = integrate.odeint(
```

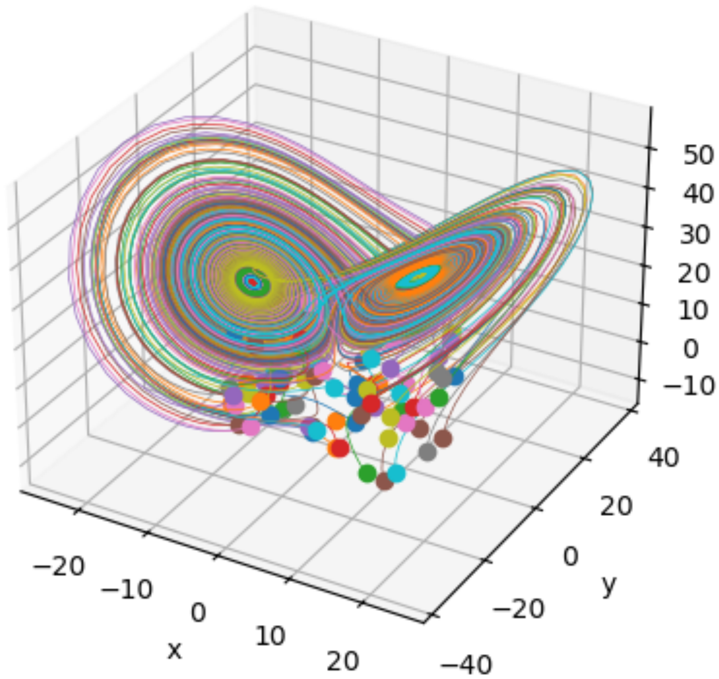
```python
            lorenz,  # Dynamics to integrate
            x0,  # Initial condition
            t,  # Time array
            args=(sigma, beta, rho)  # Parameters for the Lorenz equations
        )

    return x
```

Generate the training data:

```python
n_samples = 100  # Number of samples
n_t = 1000  # Number of time steps
dt = 0.01  # Time step
rhos_train = [10, 28, 40]  # Values of rho for training data
n_rhos = len(rhos_train)
if regenerate_data:
    data = np.zeros((n_rhos, n_samples, n_t, 3))
    for i, rho in enumerate(rhos_train):
        data[i] = generate_data(n_samples, n_t, dt, rho=rho)
    np.save('training-data.npy', data)
else:
    data = np.load('training-data.npy')
```

Plot the integrated trajectories of the Lorenz variables for rho = 28:

```python
rhoi = 1  # Index of rho value
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(n_samples):
    ax.plot(
        data[rhoi, i, :, 0],
        data[rhoi, i, :, 1],
        data[rhoi, i, :, 2],
        lw=0.5
    )
    ax.plot(
        data[rhoi, i, 0, 0], data[rhoi, i, 0, 1], data[rhoi, 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.draw()
```



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 t and the output will be the states at time $t+1$. The samples are concatenated along the first axis:

```
X = np.zeros(((n_t-1)*n_samples*n_rhos, 3))
Y = np.zeros(((n_t-1)*n_samples*n_rhos, 3))
for j in range(n_rhos):
    for i in range(n_samples):
        X[(j*n_samples+i)*(n_t-1):(j*n_samples+i+1)*(n_t-1)] = \
            data[j, i, :-1]
        Y[(j*n_samples+i)*(n_t-1):(j*n_samples+i+1)*(n_t-1)] = \
            data[j, 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:

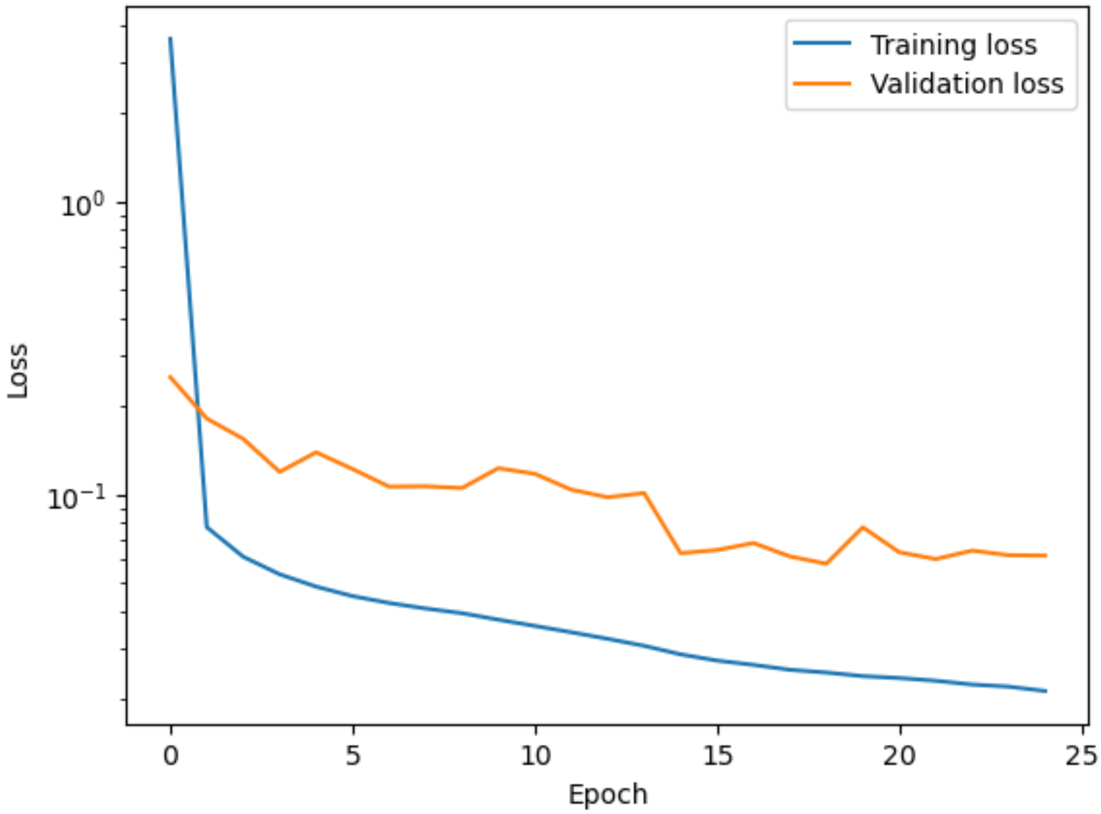
```
model = build_model()
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.001),
    loss='mean_squared_error',  # Loss function
    metrics=['mean_absolute_error'],  # Metrics to monitor
)
```

Train the model:

```
if retrain:
    history = model.fit(
        X,  # Input data
        Y,  # Target data
        epochs=25,  # 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.draw()
```



Generate new test trajectories using the trained model:

```
n_test_samples = 20  # Number of test samples
rhos_test = [17, 35]  # Values of rho for test data
n_test_rhos = len(rhos_test)
if regenerate_data:
    data_test = np.zeros((n_test_rhos, n_test_samples, n_t, 3))
    for i, rho in enumerate(rhos_test):
        data_test[i] = generate_data(n_test_samples, n_t, dt, rho=rho,
seed_offset=2*n_samples*i)
    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*n_test_rhos, 3))
Y_test = np.zeros(((n_t-1)*n_test_samples*n_test_rhos, 3))
for j in range(n_test_rhos):
    for i in range(n_test_samples):
        X_test[(j*n_test_samples+i)*(n_t-1):(j*n_test_samples+i+1)*(n_t-1)] =
\
            data_test[j, i, :-1]
        Y_test[(j*n_test_samples+i)*(n_t-1):(j*n_test_samples+i+1)*(n_t-1)] =
\
            data_test[j, i, 1:]
```

Predict the next state using the trained model:

```
Y_pred = model.predict(X_test)
```

```
   1/1249 ━━━━━━━━━━━━━━━━━━━━ 35s 28ms/step
 153/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 330us/step
 284/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step
 442/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 365us/step
 611/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 346us/step
 786/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 333us/step
 956/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 326us/step
1134/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 319us/step
1249/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 331us/step
1249/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 332us/step
```

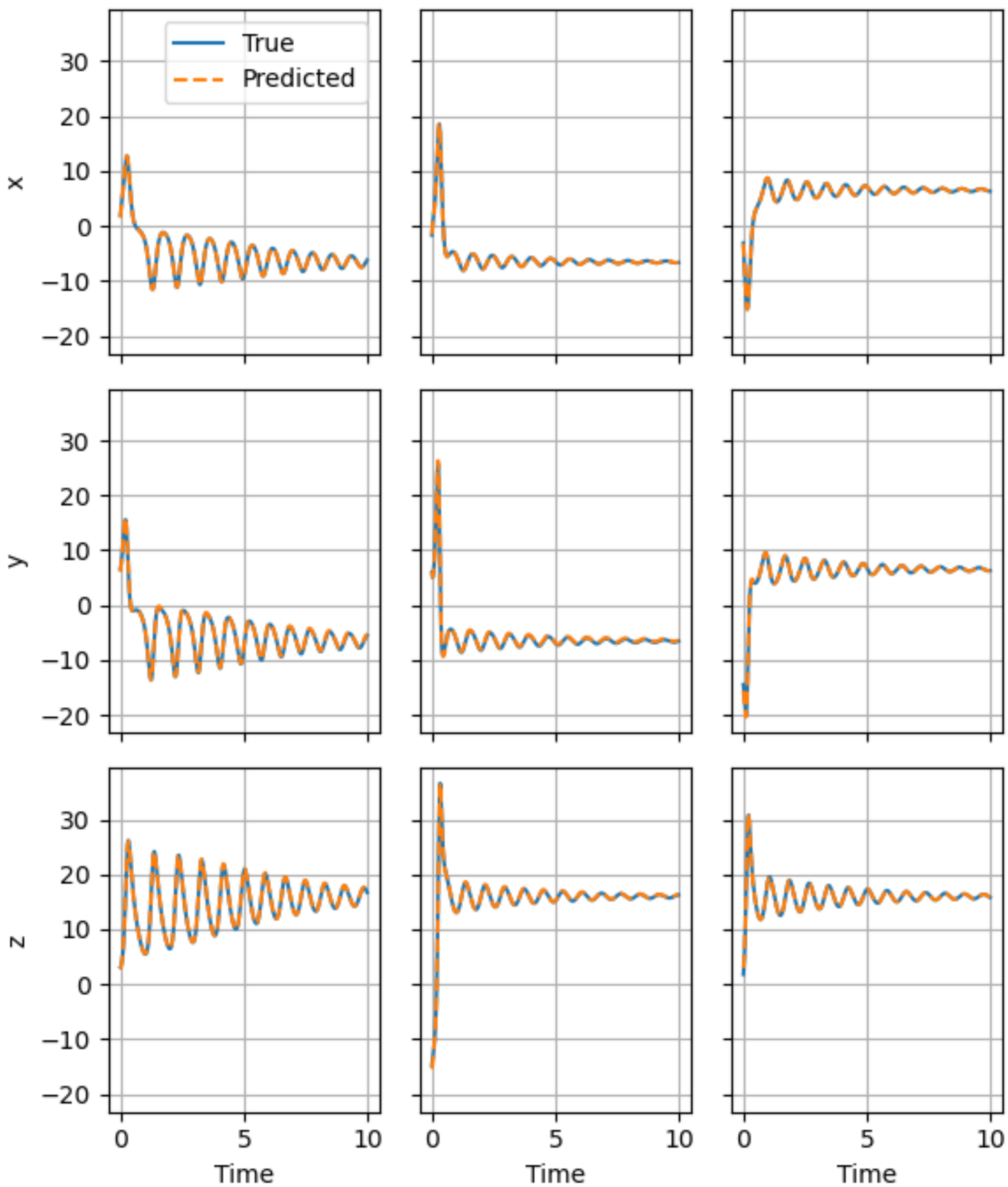
Compute the mean absolute error (MAE) between the predicted and true states:

```
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: 0.07700586249013075
```

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. The test values of ρ are 17 and 35, different than the training values of 10, 28, and 40. This demonstrates the ability of the neural network to generalize to new values of ρ.