PILSTM

April 13, 2025

[1]:

```python
import numpy as np
import os
import sys
import gym
import zipfile
import autograd
import matplotlib.gridspec as gridspec
# Use tf.random.set_seed for TensorFlow 2.0 and above
#from scipy.signal.waveforms import square
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from sklearn.model_selection import train_test_split
import random
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, model_from_json
from keras.layers import Dense
from keras.layers import Input
from tensorflow.keras import layers
```

```
2025-04-13 00:03:15.314761: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2025-04-13 00:03:15.513475: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2025-04-13 00:03:15.590707: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2025-04-13 00:03:15.774702: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2025-04-13 00:03:15.813402: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2025-04-13 00:03:16.077384: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
```

```
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
2025-04-13 00:03:17.664630: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

[15]:

```python
# @title  Hp meristor's state variable:
from IPython.display import display, Math

latex_equation = r"""
\text{State variable:}\quad \frac{dw}{dt} = \mu_\text{v} \cdot \left(␣
↪\frac{R_{\text{on}}}{D^2} \right) \cdot i(t) \cdot f(w) \\
\text{Window function:}\quad f(w) = {w(1 - w)}\\
\text{state variable in this code is w:}\quad w =  \frac{X}{D}
"""
display(Math(latex_equation))
```

$$\text{State variable:}\quad \frac{dw}{dt} = \mu_\text{v} \cdot \left(\frac{R_{\text{on}}}{D^2} \right) \cdot i(t) \cdot f(w)$$
$$\text{Window function:}\quad f(w) = w(1 - w)$$
$$\text{state variable in this code is w:}\quad w = \frac{X}{D}$$

[2]:

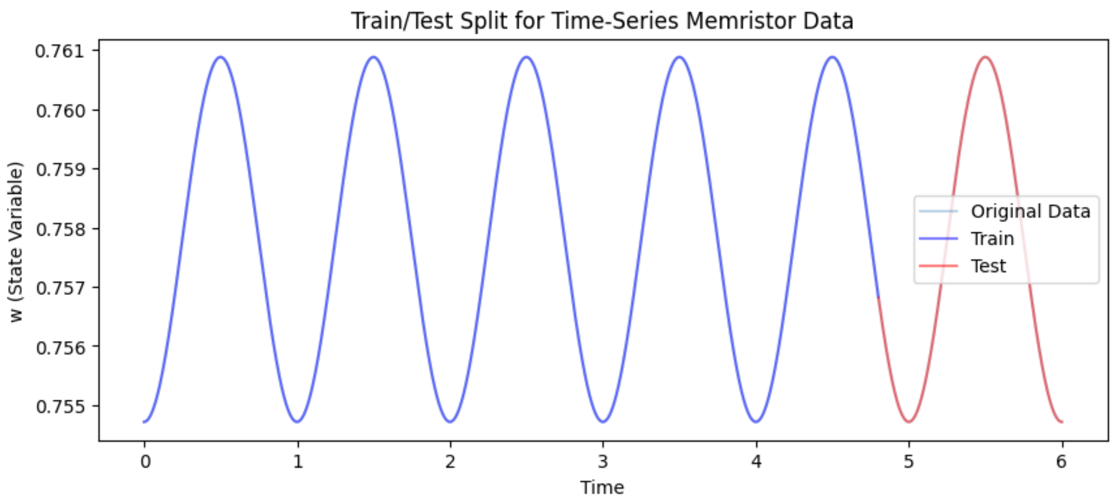
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# -------------- Physical Parameters ----------------
frequency = 1
A_train = 1.5
W_train = 2 * np.pi * frequency

mu_v = 10**4
D = 60
r_on = 0.1
r_off = 16
r0 = 4
w0 = (r0 - r_off) / (r_on - r_off)

points_per_period = 600

total_points = 10 * points_per_period

# -------------- Solving ODE ----------------
def f(t, w, A, W, mu_v, D, r_on, r_off):
    k = mu_v * (r_on / D**2)
```

```
    f_w = w * (1 - w)
    r = r_on * w + r_off * (1 - w)
    I = A * np.sin(W * t) / r
    return I * f_w * k

t_all = np.linspace(0, 6, total_points)
sol_all=solve_ivp(f, (0, 6), [w0], t_eval=t_all, args=(A_train, W_train, mu_v,␣
 ↪D, r_on, r_off),
          method='RK45', max_step=0.001)


w_all = sol_all.y[0]
v_all = A_train * np.sin(W_train * t_all)
r_all = r_on * w_all + r_off * (1 - w_all)
I_all = v_all / r_all

X_all = np.column_stack([t_all[:-1], w_all[:-1], I_all[:-1]])
y_all = w_all[1:]

# -------------- Split Data ----------------
test_ratio = 0.2
test_size = int(test_ratio * len(X_all))


test_index = np.arange(len(X_all) - test_size, len(X_all))
train_index = np.arange(0, len(X_all) - test_size)


X_train, X_test = X_all[train_index], X_all[test_index]
y_train, y_test = y_all[train_index], y_all[test_index]

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
plt.figure(figsize=(10, 4))
plt.plot(t_all[:-1], w_all[:-1], label="Original Data", alpha=0.3)
plt.plot(X_train[:, 0], y_train, color='blue', alpha=0.5, label='Train')
plt.plot(X_test[:, 0], y_test, color='red', alpha=0.5, label='Test')
plt.xlabel("Time")
plt.ylabel("w (State Variable)")
plt.title("Train/Test Split for Time-Series Memristor Data")
#plt.grid()
plt.legend()
plt.savefig("rungkutta_train_test.pdf")
plt.show()
```

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import numpy as np

# ----------------------------------
scaler_time = StandardScaler()
X_train[:, 0] = scaler_time.fit_transform(X_train[:, 0].reshape(-1, 1)).flatten()
X_test[:, 0] = scaler_time.transform(X_test[:, 0].reshape(-1, 1)).flatten()

scaler_features = MinMaxScaler(feature_range=(-1, 1))
X_train[:, 1:] = scaler_features.fit_transform(X_train[:, 1:])
X_test[:, 1:] = scaler_features.transform(X_test[:, 1:])

scaler_y = MinMaxScaler(feature_range=(-1, 1))
y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1))
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1))


X_train_scaled = X_train.copy()
X_test_scaled = X_test.copy()



# ---------------- Create Sequences ----------------
sequence_length = 10  #
def create_sequences(X, y, sequence_length):
    X_seq, y_seq = [], []
    for i in range(len(X) - sequence_length):
        X_seq.append(X[i:i+sequence_length])
        y_seq.append(y[i+sequence_length])
    return np.array(X_seq), np.array(y_seq).reshape(-1, 1)



X_train_seq, y_train_seq = create_sequences(X_train_scaled, y_train_scaled, sequence_length)
X_test_seq, y_test_seq = create_sequences(X_test_scaled, y_test_scaled, sequence_length)



print("Mean of X_train_scaled:", np.mean(X_train, axis=0))
print("Std of X_train_scaled:", np.std(X_train, axis=0))
print("Mean of X_test_scaled:", np.mean(X_test, axis=0))
print("Std of X_test_scaled:", np.std(X_test, axis=0))
print("Mean of y_train_scaled:", np.mean(y_train_scaled))
print("Std of y_train_scaled:", np.std(y_train_scaled))
```

```
print("Mean of y_test_scaled:", np.mean(y_test_scaled))
print("Std of y_test_scaled:", np.std(y_test_scaled))



print(f'X_train_seq shape: {X_train_seq.shape}, y_train_seq shape: {y_train_seq.
 ↪shape}')
print(f'X_test_seq shape: {X_test_seq.shape}, y_test_seq shape: {y_test_seq.
 ↪shape}')
```

```
(4800, 3)
(1199, 3)
(4800,)
(1199,)
```




```
Mean of X_train_scaled: [-6.65671222e-17  3.04609012e-02  2.26622042e-02]
Std of X_train_scaled: [1.         0.70298873 0.71031816]
Mean of X_test_scaled: [ 2.16470271 -0.12662702 -0.09072466]
Std of X_test_scaled: [0.24979159 0.70958015 0.68660316]
Mean of y_train_scaled: 0.03060324437440435
Std of y_train_scaled: 0.7028492870931501
Mean of y_test_scaled: -0.12719823824974452
Std of y_test_scaled: 0.7100073018388313
X_train_seq shape: (4790, 10, 3), y_train_seq shape: (4790, 1)
X_test_seq shape: (1189, 10, 3), y_test_seq shape: (1189, 1)
```

[3]:

```
import numpy as np
import random
import matplotlib.pyplot as plt
import tensorflow as tf
```

```python
from tensorflow.keras.layers import Dense, Input, LSTM # Import LSTM here
from tensorflow.keras import Sequential, regularizers
from tensorflow.keras.layers import Dropout

class PhysicsInformedRNN(tf.keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.RNN = Sequential([
            LSTM(35, return_sequences=True, activation='tanh',␣
 ↪kernel_regularizer=tf.keras.regularizers.l2(1e-4)),
            Dropout(0.06),
            LSTM(39, return_sequences=True, activation='tanh',␣
 ↪kernel_regularizer=tf.keras.regularizers.l2(1e-4)),  #
            Dropout(0.06),
            Dense(29, activation='tanh', kernel_regularizer=tf.keras.
 ↪regularizers.l2(1e-4)),
            Dense(1,)
        ])


    def call(self, inputs):
        return self.RNN(inputs)

    def build(self, input_shape):
        self.RNN.build(input_shape)
        super().build(input_shape)


pinn_rnn = PhysicsInformedRNN()
pinn_rnn.build((None, sequence_length, 3))
pinn_rnn.summary()
####################################################################
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint
import tensorflow as tf
import matplotlib.pyplot as plt

# --------
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1744490008.960806    6129 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2025-04-13 00:03:28.961320: W
```

```
tensorflow/core/common_runtime/gpu/gpu_device.cc:2343] Cannot dlopen some GPU
libraries. Please make sure the missing libraries mentioned above are installed
properly if you would like to use GPU. Follow the guide at
https://www.tensorflow.org/install/gpu for how to download and setup the
required libraries for your platform.
Skipping registering GPU devices…
```

Model: "physics_informed_rnn"

Layer (type)	Output Shape	Param #
sequential (Sequential)	?	18,350

Total params: 18,350 (71.68 KB)

Trainable params: 18,350 (71.68 KB)

Non-trainable params: 0 (0.00 B)

[4]:
```
NO = 1
Nf = X_train_seq.shape[0]
Nd = y_train_seq.shape[0]

#col_weights = tf.Variable(1.0)   # for ode loss
#u_weights = tf.Variable(1.0)     # for ic loss
#data_weights = tf.Variable(1.5)  # for data loss
#_______________________________________
col_weights = tf.Variable(tf.ones(Nf), dtype=tf.float32)   #weight of ODE
data_weights = tf.Variable(tf.ones(Nd), dtype=tf.float32)  #weight of data
u_weights = tf.Variable(tf.ones(NO), dtype=tf.float32)

optimizer_col_weights = tf.keras.optimizers.Adam(learning_rate=1e-2)
optimizer_data_weights = tf.keras.optimizers.Adam(learning_rate=1e-4)

print("Shape of col_weights:", col_weights.shape)
print("Shape of ode_res:", data_weights .shape)

print("done")
```

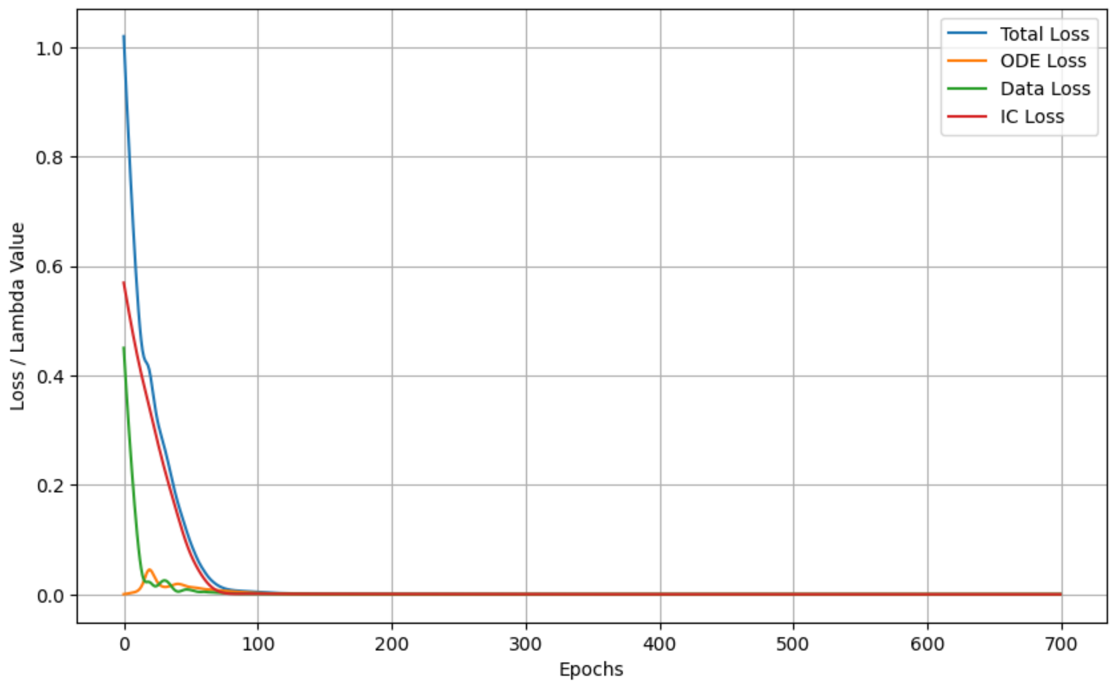
```
Shape of col_weights: (4790,)
Shape of ode_res: (4790,)
done
```

[5]:
```python
def compute_loss(X, y_true, model, col_weights, u_weights, data_weights):
    X = tf.convert_to_tensor(X, dtype=tf.float32)
    y_true = tf.convert_to_tensor(y_true, dtype=tf.float32)

    with tf.GradientTape(persistent=True) as tape:

        w_pred_sequence = model(X)
        w_pred = w_pred_sequence[:, -1, :]


        I_t = X[:, -1, 0:1]
        w_prev = X[:, -1, 1:2]
        T = X[:, -1, 2:3]

        f_w = w_pred * (1 - w_pred)

        with tf.GradientTape() as g:
            g.watch(T)

            inputs = tf.concat([I_t,w_prev, T], axis=1)
            w_pred_g = model(tf.expand_dims(inputs, axis=1))  # (batch_size, 1,␣
↪3)
        dw_dt = g.gradient(w_pred_g, T)

        ode_res = dw_dt - mu_v * (r_on / D**2) * I_t * f_w

        ode_loss = tf.reduce_mean(tf.square(col_weights[:, tf.newaxis] *␣
↪ode_res))

        data_loss = tf.reduce_mean(tf.square(data_weights * (w_pred - y_true)))


        #ic_input = tf.convert_to_tensor([[0.0, y_train[0], X_train[0, 2]]],␣
↪dtype=tf.float32)
        ic_input = X[:, 0:1, :]
        ic_pred = model(ic_input)[:, -1, :]

        ic_true = tf.convert_to_tensor(y_train[0], dtype=tf.float32)
        ic_loss = tf.reduce_mean(tf.square(u_weights * (ic_pred - ic_true)))

        total_loss = data_loss + ode_loss + ic_loss

        return total_loss, ode_loss, data_loss, ic_loss


#################################################### trainig loop
```

```
from tensorflow.keras.optimizers.schedules import ExponentialDecay
#earning_rate = tf.keras.optimizers.schedules.ExponentialDecay(
   #initial_learning_rate=1e-3
    #ecay_steps=10000 ,
    #ecay_rate=0.75
#
learning_rate=1e-3



optimizer=tf.keras.optimizers.Adam(learning_rate, clipnorm=1.0)

epochs=700

train_loss_record,ode_loss_record,data_loss_record,ic_loss_record=[],[],[],[]
for epoch in range(epochs):
    with tf.GradientTape(persistent=True) as tape:
        total_loss, ode_loss, data_loss, ic_loss = compute_loss(X_train_seq,␣
 ↪y_train_seq
                                                , pinn_rnn, col_weights,␣
 ↪u_weights, data_weights)

    grads = tape.gradient(total_loss, pinn_rnn.trainable_variables)

    grads_data = tape.gradient(data_loss, pinn_rnn.trainable_variables)
    grads_ode = tape.gradient(ode_loss, pinn_rnn.trainable_variables)

    grads_col = tape.gradient(ode_loss, col_weights)
    grads_data_weights = tape.gradient(data_loss, data_weights)


    optimizer.apply_gradients(zip(grads, pinn_rnn.trainable_variables))

    if grads_col is not None:
        optimizer_col_weights.apply_gradients(zip([grads_col], [col_weights]))
    if grads_data_weights is not None:
        optimizer_data_weights.apply_gradients(zip([grads_data_weights],␣
 ↪[data_weights]))




    train_loss_record.append(total_loss.numpy())
    ode_loss_record.append(ode_loss.numpy())
    data_loss_record.append(data_loss.numpy())
    ic_loss_record.append(ic_loss.numpy())
```

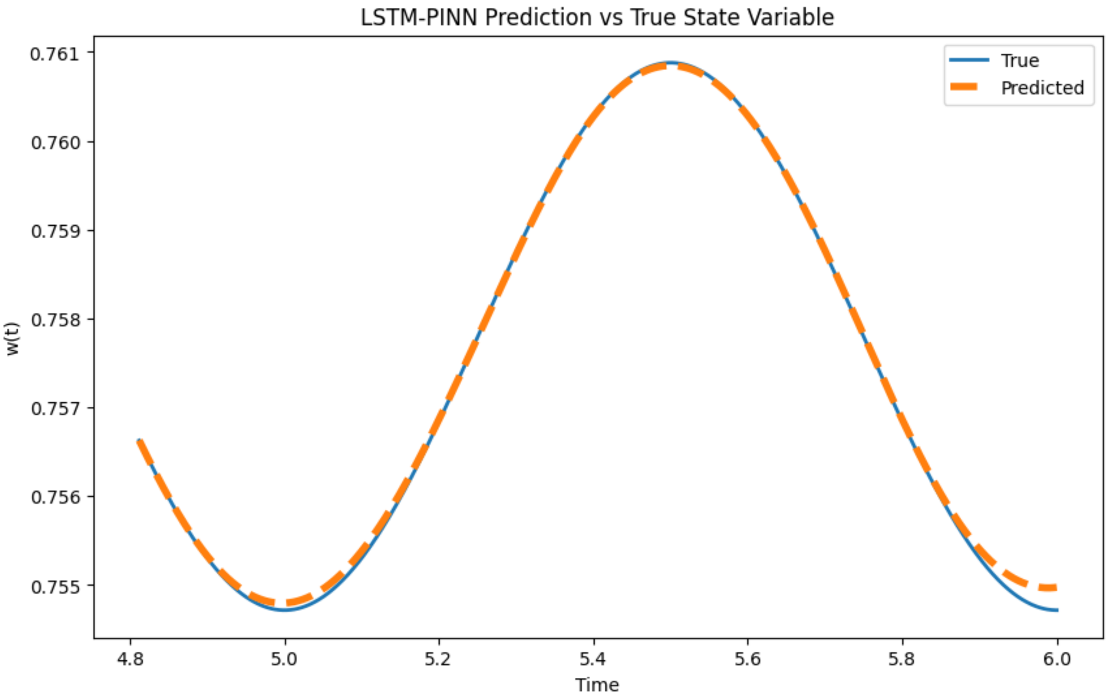
```python
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1}/{epochs} | "
              f"Total Loss: {train_loss_record[-1]:.6f} | "
              f"ODE Loss: {ode_loss_record[-1]:.6f} | "
              f"Data Loss: {data_loss_record[-1]:.6f} | "
              f"IC Loss: {ic_loss_record[-1]:.6f} | ")


# ---------------- Plot Adaptive Lambda Values ----------------
plt.figure(figsize=(10, 6))
plt.plot(train_loss_record, label='Total Loss')
plt.plot(ode_loss_record, label='ODE Loss')
plt.plot(data_loss_record, label='Data Loss')
plt.plot(ic_loss_record, label='IC Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss / Lambda Value")
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 100/700 | Total Loss: 0.004325 | ODE Loss: 0.002554 | Data Loss: 0.000665
| IC Loss: 0.001106 |
Epoch 200/700 | Total Loss: 0.000526 | ODE Loss: 0.000045 | Data Loss: 0.000055
| IC Loss: 0.000426 |
Epoch 300/700 | Total Loss: 0.000265 | ODE Loss: 0.000013 | Data Loss: 0.000020
| IC Loss: 0.000232 |
Epoch 400/700 | Total Loss: 0.000132 | ODE Loss: 0.000007 | Data Loss: 0.000009
| IC Loss: 0.000116 |
Epoch 500/700 | Total Loss: 0.000061 | ODE Loss: 0.000004 | Data Loss: 0.000005
| IC Loss: 0.000052 |
Epoch 600/700 | Total Loss: 0.000027 | ODE Loss: 0.000003 | Data Loss: 0.000003
| IC Loss: 0.000021 |
Epoch 700/700 | Total Loss: 0.000012 | ODE Loss: 0.000002 | Data Loss: 0.000002
| IC Loss: 0.000008 |
```



[6]:
```
w_pred_seq = pinn_rnn.predict(X_test_seq)
w_pred = w_pred_seq[:, -1, :]

w_pred_original = scaler_y.inverse_transform(w_pred)


y_test_original = scaler_y.inverse_transform(y_test_seq)


t_test_plot = t_all[len(t_all) - len(w_pred_original):]


# --------------- Plotting ----------------
plt.figure(figsize=(10, 6))
plt.plot(t_test_plot, y_test_original, label='True', linewidth=2)
plt.plot(t_test_plot, w_pred_original, label='Predicted', linestyle='--',
 ↪linewidth=4)
plt.xlabel('Time')
plt.ylabel('w(t)')
plt.title('LSTM-PINN Prediction vs True State Variable')
plt.savefig("testlstm.pdf")
plt.legend()

plt.show()
```

```
38/38                    1s 8ms/step
```



[7]:
```python
import os

save_dir = "./models"
if not os.path.exists(save_dir):
    os.makedirs(save_dir)

save_path = os.path.join(save_dir, "my_pinn_model.keras")

pinn_rnn.save(save_path)
print(f"Model saved at: {save_path}")
from tensorflow.keras.models import load_model

save_path = "./models/my_pinn_model.keras"

if os.path.exists(save_path):
    loaded_pinn_model = load_model(save_path,␣
 ↪custom_objects={'PhysicsInformedRNN': PhysicsInformedRNN})
    print("  Model loaded successfully!")
else:
    print("  Error: Model file not found!")
```

```
Model saved at: ./models/my_pinn_model.keras
  Model loaded successfully!
```

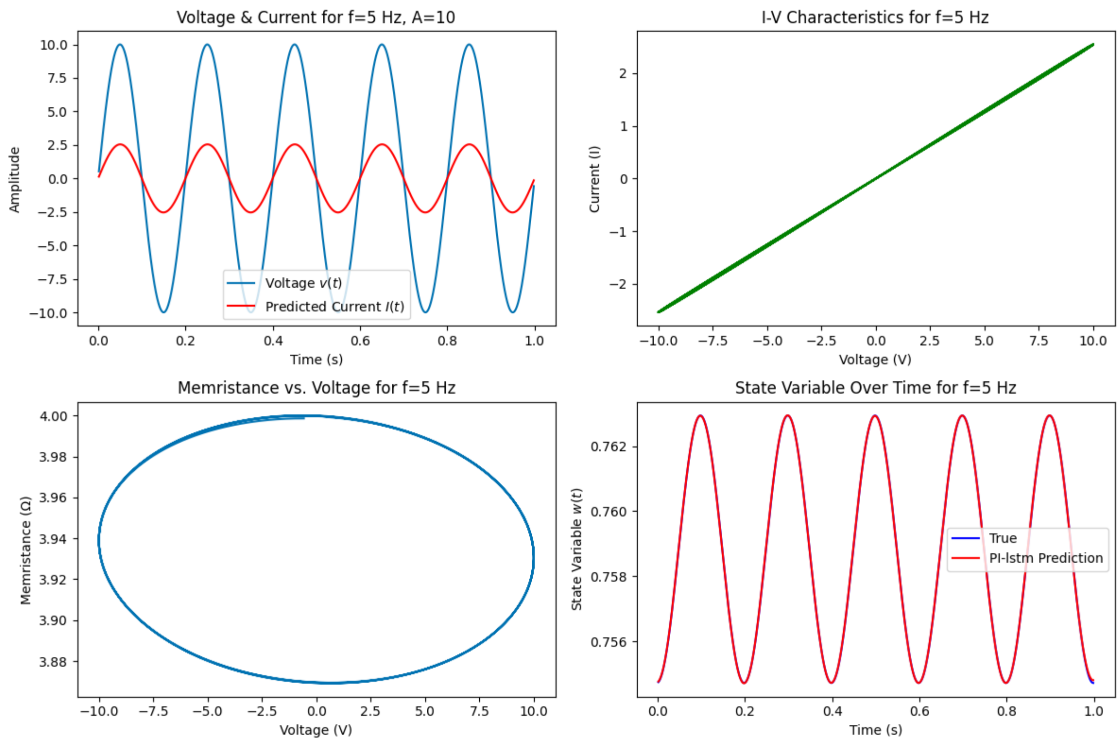
[9]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import tensorflow as tf

# ----------------- Physical Parameters -----------------
frequency = 5
A_train = 10
W_train = 2 * np.pi * frequency
mu_v = 10**4
D = 60
r_on = 0.1
r_off = 16
r0 = 4
w0 = (r0 - r_off) / (r_on - r_off)

points_per_period = 600
total_points = 10 * points_per_period

# ------------------ ODE Solver ------------------
def f(t, w, A, W, mu_v, D, r_on, r_off):
    k = mu_v * (r_on / D**2)
    f_w = w * (1 - w)
    r = r_on * w + r_off * (1 - w)
    I = A * np.sin(W * t) / r
    return I * f_w * k

t_all = np.linspace(0, 1, total_points)
sol_all = solve_ivp(
    f, (0, 1), [w0], t_eval=t_all,
    args=(A_train, W_train, mu_v, D, r_on, r_off),
    method='RK45',
    max_step=0.001
)

# ----------------- Data Preparation -----------------
w_all = sol_all.y[0]
v_all = A_train * np.sin(W_train * t_all)
r_all = r_on * w_all + r_off * (1 - w_all)
I_all = v_all / r_all

X_all = np.column_stack([t_all[:-1], w_all[:-1], I_all[:-1]])
y_all = w_all[1:]

# ----------------- Data Normalization -----------------
X_all_scaled = X_all.copy()
```

```python
scaler_time = StandardScaler()
X_all_scaled[:, 0] = scaler_time.fit_transform(X_all_scaled[:, 0].reshape(-1,␣
 ↪1)).flatten()

scaler_features = MinMaxScaler(feature_range=(-1, 1))
X_all_scaled[:, 1:] = scaler_features.fit_transform(X_all_scaled[:, 1:])

scaler_y = MinMaxScaler(feature_range=(-1, 1))
y_all_scaled = scaler_y.fit_transform(y_all.reshape(-1, 1))

# ----------------- Create Sequences -----------------
sequence_length = 10
def create_sequences(X, y, sequence_length):
    X_seq, y_seq = [], []
    for i in range(len(X) - sequence_length):
        X_seq.append(X[i:i+sequence_length])
        y_seq.append(y[i+sequence_length])
    return np.array(X_seq), np.array(y_seq).reshape(-1, 1)

X_seq, y_seq = create_sequences(X_all_scaled, y_all_scaled, sequence_length)

# ----------------- PINN Prediction -----------------
w_pred_seq_new = pinn_rnn.predict(X_seq)
w_pred = w_pred_seq_new[:, -1, :]
w_pred_original_new = scaler_y.inverse_transform(w_pred)
y_all_rescaled = scaler_y.inverse_transform(y_seq)

t_test_plot = t_all[:len(w_pred_original_new)]

# ----------------- Compute Memristance and Current -----------------
r_M_pred = r_on * w_pred_original_new + r_off * (1.0 - w_pred_original_new)

print(f"Shape of v_all[:-1]: {v_all[:-1].shape}")
print(f"Shape of r_M_pred.flatten(): {r_M_pred.flatten().shape}")

min_length = min(len(v_all[:-1]), len(r_M_pred.flatten()))
v_all_adjusted = v_all[:min_length]
r_M_pred_adjusted = r_M_pred.flatten()[:min_length]

I_pred_new = v_all_adjusted / r_M_pred_adjusted

# ----------------- Data Trimming -----------------
t_trimmed = t_all[:min_length][sequence_length:]
v_trimmed = v_all[:min_length][sequence_length:]
I_trimmed = I_pred_new[sequence_length:]
w_trimmed = w_pred_original_new[:min_length][sequence_length:]
y_trimmed = y_all_rescaled[:min_length][sequence_length:]
```

```python
# ----------------- Visualization -----------------
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# Voltage and Current vs. Time
axes[0, 0].plot(t_trimmed, v_trimmed, '#0072B2', label='Voltage $v(t)$')
axes[0, 0].plot(t_trimmed, I_trimmed, 'r', label='Predicted Current $I(t)$')
axes[0, 0].set_xlabel('Time (s)')
axes[0, 0].set_ylabel('Amplitude')
axes[0, 0].set_title(f'Voltage & Current for f={frequency} Hz, A={A_train}')
axes[0, 0].legend()


# I-V Characteristics
axes[0, 1].plot(v_trimmed, I_trimmed, 'g-', label='PI-lstm  Prediction')
axes[0, 1].set_xlabel('Voltage (V)')
axes[0, 1].set_ylabel('Current (I)')
axes[0, 1].set_title(f'I-V Characteristics for f={frequency} Hz')


# Memristance vs. Voltage
axes[1, 0].plot(v_trimmed, r_M_pred_adjusted[sequence_length:], '#0072B2')
axes[1, 0].set_xlabel('Voltage (V)')
axes[1, 0].set_ylabel('Memristance (Ω)')
axes[1, 0].set_title(f'Memristance vs. Voltage for f={frequency} Hz')


# State Variable w(t) Comparison
axes[1, 1].plot(t_trimmed, y_trimmed, label='True', color='blue')
axes[1, 1].plot(t_trimmed, w_trimmed, 'r', label='PI-lstm Prediction')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('State Variable $w(t)$')
axes[1, 1].set_title(f'State Variable Over Time for f={frequency} Hz')
axes[1, 1].legend()


plt.tight_layout()
plt.savefig(f"piMemristor_Plots_square{frequency}Hz.pdf", format="pdf",␣
 ↪bbox_inches="tight")
plt.show()
```

```
188/188              0s 2ms/step
Shape of v_all[:-1]: (5999,)
Shape of r_M_pred.flatten(): (5989,)
```



[14]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import tensorflow as tf

# ----------------- Physical Parameters ------------------
frequency = 5
A_train = 10
W_train = 2 * np.pi * frequency
mu_v = 10**4
D = 60
r_on = 0.1
r_off = 16
r0 = 4
w0 = (r0 - r_off) / (r_on - r_off)

points_per_period = 600
total_points = 10 * points_per_period


# Voltage function
def v(t):
```

```python
    return A_train* signal.square(W_train  * t)

# ------------------ ODE Solver ------------------
def f(t, w, A, W, mu_v, D, r_on, r_off):
    k = mu_v * (r_on / D**2)
    f_w = w * (1 - w)
    r = r_on * w + r_off * (1 - w)
    I = v(t) / r
    return I * f_w * k

t_all = np.linspace(0, 1, total_points)
sol_all = solve_ivp(
    f, (0, 1), [w0], t_eval=t_all,
    args=(A_train, W_train, mu_v, D, r_on, r_off),
    method='RK45',
    max_step=0.001
)

# ----------------- Data Preparation -----------------
w_all = sol_all.y[0]
v_all = v(t_all)
r_all = r_on * w_all + r_off * (1 - w_all)
I_all = v_all / r_all

X_all = np.column_stack([t_all[:-1], w_all[:-1], I_all[:-1]])
y_all = w_all[1:]

# ----------------- Data Normalization -----------------
X_all_scaled = X_all.copy()
scaler_time = StandardScaler()
X_all_scaled[:, 0] = scaler_time.fit_transform(X_all_scaled[:, 0].reshape(-1,
 ↪1)).flatten()

scaler_features = MinMaxScaler(feature_range=(-1, 1))
X_all_scaled[:, 1:] = scaler_features.fit_transform(X_all_scaled[:, 1:])

scaler_y = MinMaxScaler(feature_range=(-1, 1))
y_all_scaled = scaler_y.fit_transform(y_all.reshape(-1, 1))

# ----------------- Create Sequences -----------------
sequence_length = 10
def create_sequences(X, y, sequence_length):
    X_seq, y_seq = [], []
    for i in range(len(X) - sequence_length):
        X_seq.append(X[i:i+sequence_length])
        y_seq.append(y[i+sequence_length])
    return np.array(X_seq), np.array(y_seq).reshape(-1, 1)
```

```python
X_seq, y_seq = create_sequences(X_all_scaled, y_all_scaled, sequence_length)

# ----------------- PINN Prediction -----------------
w_pred_seq_new = pinn_rnn.predict(X_seq)
w_pred = w_pred_seq_new[:, -1, :]
w_pred_original_new = scaler_y.inverse_transform(w_pred)
y_all_rescaled = scaler_y.inverse_transform(y_seq)

t_test_plot = t_all[:len(w_pred_original_new)]

# ----------------- Compute Memristance and Current -----------------
r_M_pred = r_on * w_pred_original_new + r_off * (1.0 - w_pred_original_new)

print(f"Shape of v_all[:-1]: {v_all[:-1].shape}")
print(f"Shape of r_M_pred.flatten(): {r_M_pred.flatten().shape}")

min_length = min(len(v_all[:-1]), len(r_M_pred.flatten()))
v_all_adjusted = v_all[:min_length]
r_M_pred_adjusted = r_M_pred.flatten()[:min_length]

I_pred_new = v_all_adjusted / r_M_pred_adjusted

# ----------------- Data Trimming -----------------
t_trimmed = t_all[:min_length][sequence_length:]
v_trimmed = v_all[:min_length][sequence_length:]
I_trimmed = I_pred_new[sequence_length:]
w_trimmed = w_pred_original_new[:min_length][sequence_length:]
y_trimmed = y_all_rescaled[:min_length][sequence_length:]

# ----------------- Visualization -----------------
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# Voltage and Current vs. Time
axes[0, 0].plot(t_trimmed, v_trimmed, '#0072B2', label='Voltage $v(t)$')
axes[0, 0].plot(t_trimmed, I_trimmed, 'r', label='Predicted Current $I(t)$')
axes[0, 0].set_xlabel('Time (s)')
axes[0, 0].set_ylabel('Amplitude')
axes[0, 0].set_title(f'Voltage & Current for f={frequency} Hz, A={A_train}')
axes[0, 0].legend()


# I-V Characteristics
axes[0, 1].plot(v_trimmed, I_trimmed, 'g-', label='PI-lstm  Prediction')
axes[0, 1].set_xlabel('Voltage (V)')
axes[0, 1].set_ylabel('Current (I)')
axes[0, 1].set_title(f'I-V Characteristics for f={frequency} Hz')
```

```
# Memristance vs. Voltage
axes[1, 0].plot(v_trimmed, r_M_pred_adjusted[sequence_length:], '#0072B2')
axes[1, 0].set_xlabel('Voltage (V)')
axes[1, 0].set_ylabel('Memristance (Ω)')
axes[1, 0].set_title(f'Memristance vs. Voltage for f={frequency} Hz')


# State Variable w(t) Comparison
axes[1, 1].plot(t_trimmed, y_trimmed, label='True', color='blue')
axes[1, 1].plot(t_trimmed, w_trimmed, 'r', label='PI-lstm Prediction')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('State Variable $w(t)$')
axes[1, 1].set_title(f'State Variable Over Time for f={frequency} Hz')
axes[1, 1].legend()


plt.tight_layout()
plt.savefig(f"piMemristor_Plots_square{frequency}Hz.pdf", format="pdf",␣
 ↪bbox_inches="tight")
plt.show()
```

```
188/188              0s 2ms/step
Shape of v_all[:-1]: (5999,)
Shape of r_M_pred.flatten(): (5989,)
```