



**KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY**  
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE  
Narayanaguda, Hyderabad.

# **Embedded Learning**

## **Day 3**

### **RNN Exercise**

**22-01-2025**

**BY**  
**ASHA M**  
**ASSISTANT PROFESSOR**  
**CSE(AI&ML)**  
**KMIT**

## Exercise : Time Series Forecasting with RNN

Build an RNN model in PyTorch to predict future values of a simulated ECG signal.

### Steps:

#### 1.Data Preparation:

- Generate a noisy sine wave as ECG signal.
- Convert the signal into sequences of 20 time steps for prediction.

#### 2.RNN Model:

- Define an RNN model with 64 hidden units and a linear output layer for time step prediction.

#### 3.Training:

- Use MSELoss as the loss function and Adam as the optimizer.
- Train the model for 50 epochs and monitor the loss every 10 epochs.

#### 4.Prediction:

- After training, use `model.eval()` and generate the next 50 predictions iteratively.

## Step 1: Import Libraries

Import essential libraries for data processing, model creation, training, and visualization:

- **NumPy (np)** for numerical computations.
- **Pandas (pd)** for handling and preprocessing datasets.
- **PyTorch (torch)** and **torch.nn** for building and training neural networks.
- **torch.optim** for applying optimization algorithms like SGD or Adam.
- **Matplotlib (plt)** for visualizing trends and results.
- **MinMaxScaler** from `sklearn.preprocessing` to normalize data for stable and efficient model training.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
```

- **Load and preprocess data** with pandas and MinMaxScaler.
- **Create neural network** with torch and torch.nn.
- **Train and optimize model** using torch.optim.
- **Visualize results** using matplotlib.pyplot.

## Step 2: Simulate and Preprocess Data

- **Simulate ECG signal:** Create a synthetic ECG signal using a sine wave with added Gaussian noise. This is achieved with **np.linspace** to generate time steps and **np.sin** to produce the sine wave, combined with **np.random.normal** for noise simulation.
- **Convert to DataFrame:** Use **Pandas (pd)** to organize the ECG signal into a DataFrame for easier data manipulation and preprocessing.
- **Preview data:** Print the first 5 rows of the DataFrame using **df.head()** to inspect the generated data.

```
# Simulated ECG signal using a sine wave
time_steps = np.linspace(0, 100, 1000) # 1000 time steps
ecg_signal = np.sin(time_steps) + np.random.normal(scale=0.1, size=len(time_steps)) # Sine wave + noise

# Convert to DataFrame
df = pd.DataFrame({'ECG': ecg_signal})
print(df.head()) # Print first 5 rows
```

	ECG
0	0.053589
1	0.073980
2	0.134531
3	0.231451
4	0.313890

- **Generate Time Steps:** Use `np.linspace` to create 1,000 evenly spaced values between 0 and 100.
- **Simulate ECG Signal:** Combine a sine wave (`np.sin`) with Gaussian noise (`np.random.normal`) to mimic an ECG signal.
- **Create DataFrame:** Use `pd.DataFrame` to store the ECG signal in a structured format.
- **Preview Data:** Display the first 5 rows using `df.head()` to inspect the generated data.

## Step 3: Prepare Data for Model Training

### 1. Convert Data into Sequences

- Define a sequence length (20 in this case) and create sequences of 20 consecutive values from the data (`df['ECG']`) to predict the next value.
- Use the `create_sequences` function to generate input sequences (sequences) and corresponding target values (targets).

### 2. Convert to PyTorch Tensors

- Convert the sequences and targets into PyTorch tensors for training.
- Add an additional dimension using `.unsqueeze(-1)` to shape the data appropriately for PyTorch models.

### 3. Inspect Shapes

- Print the shapes of the input (X) and target (y) tensors to confirm correctness.

## Sequence Creation

- The `create_sequences` function iterates through the data, extracting subsequences of length `sequence_length` (20) and their corresponding next value as the target.
- The function returns these as NumPy arrays, making them ready for further processing.

## Conversion to PyTorch Tensors

- Input sequences (X) and targets (y) are converted to PyTorch tensors using `torch.tensor`.
- The `.unsqueeze(-1)` ensures the data has the correct shape for models expecting input dimensions like (batch\_size, sequence\_length, features).

## Purpose

- This process prepares the data for time-series prediction by transforming the raw ECG data into a supervised learning format, with sequences as inputs and the next value as the output.



```
# Convert data into sequences
sequence_length = 20 # Using last 20 values to predict the next one

def create_sequences(data, seq_length):
    sequences = []
    targets = []

    for i in range(len(data) - seq_length):
        seq = data[i:i+seq_length]
        target = data[i+seq_length] # The next value
        sequences.append(seq)
        targets.append(target)

    return np.array(sequences), np.array(targets)

sequences, targets = create_sequences(df['ECG'].values, sequence_length)

# Convert to PyTorch tensors
X = torch.tensor(sequences, dtype=torch.float32).unsqueeze(-1) # Add feature dimension
y = torch.tensor(targets, dtype=torch.float32).unsqueeze(-1) # Make it (batch_size, 1)

print(f"X shape: {X.shape}, y shape: {y.shape}")
```

```
# Convert data into sequences
sequence_length = 20 # Using last 20 values to predict the next one

def create_sequences(data, seq_length):
    sequences = []
    targets = []
```

set the length of each input sequence for the model. It indicates that the model will use the last 20 values of the ECG signal to predict the next value.

Initialize two empty lists, sequences and targets, which will hold the input sequences (previous 20 ECG values) and their corresponding target values (the next value in the ECG signal).

```
for i in range(len(data) - seq_length):  
    seq = data[i:i+seq_length]  
    target = data[i+seq_length] # The next value  
    sequences.append(seq)  
    targets.append(target)  
  
return np.array(sequences), np.array(targets)
```

This loop iterates through the ECG signal and creates input-output pairs, where each input is a sequence of the last 20 values and each output is the next value after the sequence.

Converts the lists sequences and targets into NumPy arrays and returns them.

## Shape:

- seq: Each subsequence is a slice of the ECG data with shape  $(\text{seq\_length},) = (20,)$ .
- target: A single value corresponding to the next data point after the subsequence, with shape  $(1,)$ .
- After the loop:
  - sequences: A list of sequences with shape  $(\text{num\_sequences}, \text{seq\_length})$  where  $\text{num\_sequences} = n - 20$  and  $\text{seq\_length} = 20$ .
  - targets: A list of targets with shape  $(\text{num\_sequences},)$ .
- sequences: After conversion to a NumPy array, its shape becomes  $(\text{num\_sequences}, \text{seq\_length}) = (n - 20, 20)$ , where  $n$  is the total number of ECG data points.
- targets: After conversion to a NumPy array, its shape becomes  $(\text{num\_sequences},) = (n - 20,)$ , where each target corresponds to the next ECG value after each sequence.

```
sequences, targets = create_sequences(df['ECG'].values, sequence_length)

# Convert to PyTorch tensors
X = torch.tensor(sequences, dtype=torch.float32).unsqueeze(-1) # Add feature dimension
y = torch.tensor(targets, dtype=torch.float32).unsqueeze(-1) # Make it (batch_size, 1)

print(f"X shape: {X.shape}, y shape: {y.shape}")
```

call the `create_sequences` function with the ECG data and the sequence length of 20, creating the sequences and targets.

Converts the sequences and targets into PyTorch tensors, and adds an additional feature dimension to the sequences (to make it compatible with PyTorch models).

Prints the shapes of X and y to verify that the data is properly formatted for model training.

## Shape:

sequences: Before conversion, its shape is  $(\text{num\_sequences}, \text{seq\_length}) = (n - 20, 20)$ .

- $X$  (after `.unsqueeze(-1)`): The shape becomes  $(\text{num\_sequences}, \text{seq\_length}, 1) = (n - 20, 20, 1)$ . This adds a feature dimension of size 1 for each time step in the sequence.

targets: Before conversion, its shape is  $(\text{num\_sequences},) = (n - 20,)$ .

- $y$  (after `.unsqueeze(-1)`): The shape becomes  $(\text{num\_sequences}, 1) = (n - 20, 1)$ , as we want each target to be a single value for each sequence.

## Output:

- X.shape: (num\_sequences, sequence\_length, 1) = (n - 20, 20, 1), where each sequence has 20 time steps and one feature (the ECG value at each time step).
- y.shape: (num\_sequences, 1) = (n - 20, 1), where each target corresponds to the next ECG value after each sequence.

```
X shape: torch.Size([980, 20, 1]), y shape: torch.Size([980, 1])
```

## Step 4: Define and Instantiate RNN Model

### 1. Define the RNN Model Class

1. Purpose: How is the RNN model defined in this step, and what are the parameters used for the RNN layer and fully connected layer?

### 2. Define the Forward Pass

1. Purpose: What does the forward method do in this model, and how is the output from the RNN layer processed to make the prediction?

### 3. Instantiate the Model

1. Purpose: How is the model instantiated, and what information is printed when you print the model?

### 4. Shape of Data

1. Purpose: What are the expected shapes of the input data to the RNN, the output from the RNN, and the final output after the fully connected layer?



```
class RNNModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, num_layers=2):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, 1) # Fully connected output layer

    def forward(self, x):
        rnn_out, _ = self.rnn(x)
        output = self.fc(rnn_out[:, -1, :]) # Take last timestep output
        return output

# Instantiate model
model = RNNModel()
print(model)
```

Def `__init__`: Defines the architecture of the Recurrent Neural Network (RNN) model. The model consists of an RNN layer followed by a fully connected output layer.

Def `forward` : Defines how data flows through the network

Instantiates the model. The `RNNModel()` class is called to create an object of the RNN model.

- `input_size=1`: This specifies that the model expects one feature per timestep.
- `hidden_size=64`: This is the number of hidden units in the RNN layer, determining the model's capacity to learn temporal dependencies.
- `num_layers=2`: This indicates the number of stacked RNN layers. More layers allow the model to capture more complex patterns.
- `self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)`: This defines the RNN layer, where `batch_first=True` ensures that the input data will be expected in the shape `(batch_size, seq_length, input_size)`.
- `self.fc = nn.Linear(hidden_size, 1)`: This defines a fully connected layer that will output a single value (the prediction for the next time step in the ECG sequence).

- `rnn_out, _ = self.rnn(x)`: Passes the input `x` through the RNN layer. The RNN returns the output at each time step (`rnn_out`) and the hidden state (`_`), but we are only interested in the output.
- `output = self.fc(rnn_out[:, -1, :])`: Since the task is to predict the next ECG value, the model takes the last output of the sequence (`rnn_out[:, -1, :]`), which represents the output at the final time step, and passes it through the fully connected layer to make the prediction.

## Shape at Each Step:

- **Input to the Model (x):**

- Shape: (batch\_size, seq\_length, input\_size), where batch\_size is the number of samples, seq\_length is the sequence length (20), and input\_size=1 is the number of features (ECG value).

- **Output of the RNN Layer (rnn\_out):**

- Shape: (batch\_size, seq\_length, hidden\_size), where hidden\_size=64 is the number of hidden units. This represents the output at each time step for every input sequence.

- **Output after Fully Connected Layer (output):**

- Shape: (batch\_size, 1), as we are predicting a single value (the next ECG value) for each input sequence.

## Summary of Shapes:

- **Input to the model:**  $(\text{batch\_size}, \text{seq\_length}, \text{input\_size}) = (n - 20, 20, 1)$
- **Output after RNN Layer:**  $(\text{batch\_size}, \text{seq\_length}, \text{hidden\_size}) = (n - 20, 20, 64)$
- **Output after Fully Connected Layer:**  $(\text{batch\_size}, 1) = (n - 20, 1)$

## Step 5: Training the Model

### 1. Define the Loss Function and Optimizer

1. What loss function is used in this step for training the model, and why is it appropriate for this task?
2. Which optimizer is chosen for training, and what does the learning rate control?

### 2. Training Loop

1. What operations are performed in each epoch during the training loop (forward pass, loss computation, backpropagation, and weight update)?

### 3. Loss Monitoring

1. Why is the loss printed every 10 epochs, and how does this help in tracking the training progress?

```
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
epochs = 50
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    outputs = model(X) # Forward pass
    loss = criterion(outputs, y) # Compute loss
    loss.backward() # Backpropagation
    optimizer.step() # Update weights

    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.5f}")
```

Initializes the loss function as Mean Squared Error (MSE) Loss. This function computes the average squared differences between predicted and actual values, suitable for regression tasks.

Initializes the Adam optimizer to optimize the model's parameters. The learning rate (lr=0.001) controls how much the model's parameters are adjusted during each update.

Sets the number of epochs (iterations through the entire dataset) for training. Here, it is set to 50.

Zero\_grad: Clears the previous gradients stored in the optimizer to prevent them from accumulating across iterations.

## Step 6: Model Evaluation and Prediction

### 1. Set the model to evaluation mode:

- Use `model.eval()` and `torch.no_grad()`.

### 2. Make predictions iteratively:

- Use the last sequence from `X[-1]` to predict the next 50 time steps.

### 3. Update the input sequence:

- Append the predicted value and update the sequence for the next prediction.

### 4. Print predictions:

- Print each predicted value and store them for analysis.



```
model.eval()
with torch.no_grad():
    test_seq = X[-1].unsqueeze(0) # Take the last sequence as input
    predictions = []

    print("\nGenerated Predictions:")
    for i in range(50): # Predict next 50 time steps
        pred = model(test_seq) # Output shape: (1, 1)
        predictions.append(pred.item())

    print(f"Step {i+1}: Predicted Value: {pred.item():.5f}")

    # Prepare next input
    pred = pred.unsqueeze(-1) # Ensure shape (1, 1, 1)
    test_seq = torch.cat((test_seq[:, 1:, :], pred), dim=1) # Maintain sequence length
```

- **Set model to evaluation mode:** `model.eval()`

Prepares the model for inference, disabling dropout and batch normalization updates.

- **Disable gradient calculation:** with `torch.no_grad()`:

Saves memory and computation by not calculating gradients during prediction.

- **Prepare last sequence for input:** `test_seq = X[-1].unsqueeze(0)`

Takes the last training sequence and reshapes it to include a batch dimension.

- **Initialize prediction list:** `predictions = []`

Creates a list to store predicted values.

- **Predict 50 time steps in a loop:** for `i` in `range(50)`:

Makes 50 predictions, storing and printing each predicted value.

- **Reshape the predicted value:** `pred = pred.unsqueeze(-1)`

Ensures the prediction has the correct dimensions for the next input.

- **Update input sequence:** `test_seq = torch.cat((test_seq[:, 1:, :], pred), dim=1)`

Updates the input sequence by removing the first value and adding the predicted value, maintaining the sequence length.

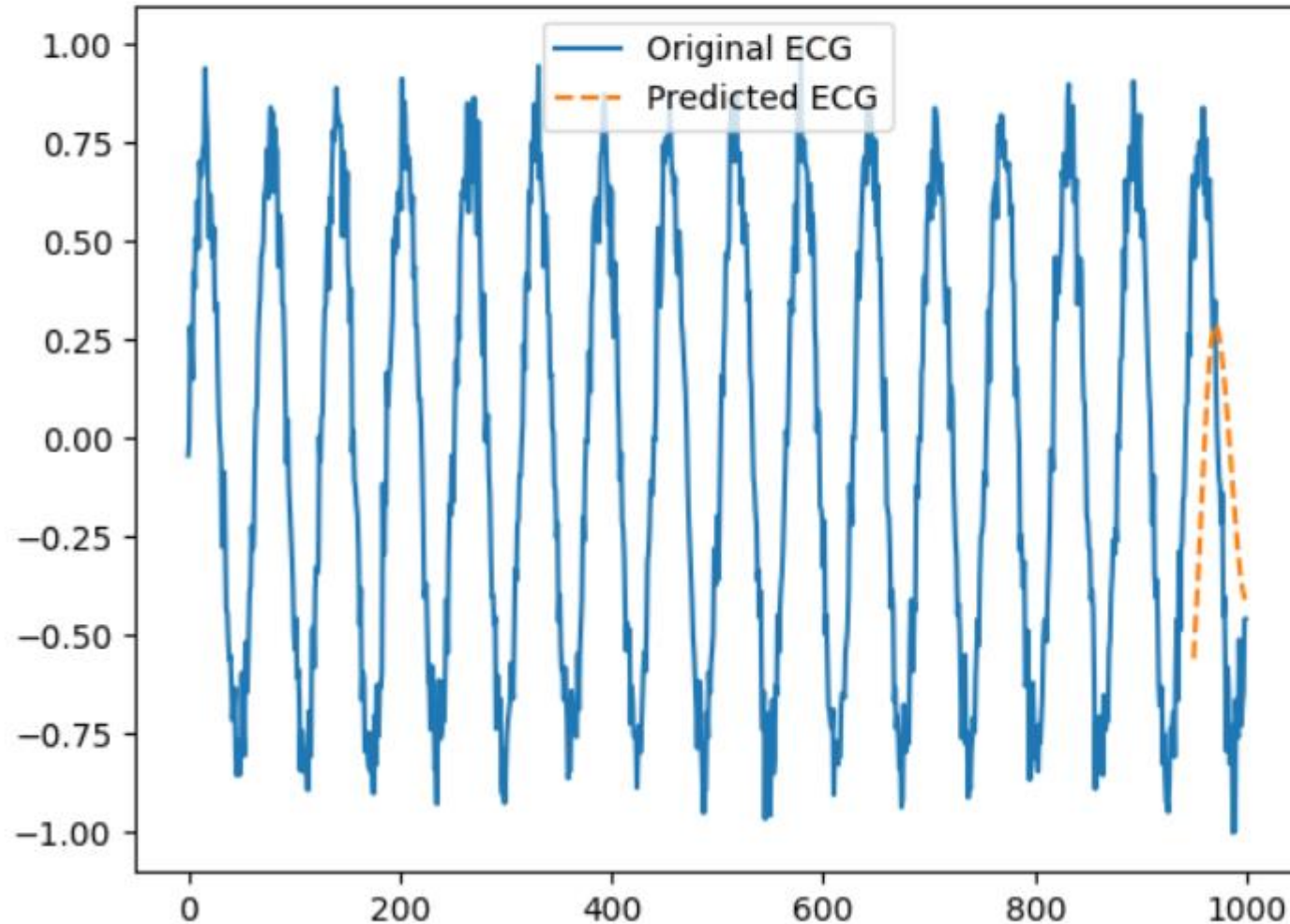
## Generated Predictions:

Step 1: Predicted Value: -0.52234  
Step 2: Predicted Value: -0.46358  
Step 3: Predicted Value: -0.40732  
Step 4: Predicted Value: -0.34827  
Step 5: Predicted Value: -0.28727  
Step 6: Predicted Value: -0.22463  
Step 7: Predicted Value: -0.15926  
Step 8: Predicted Value: -0.09097  
Step 9: Predicted Value: -0.02128  
Step 10: Predicted Value: 0.04991  
Step 11: Predicted Value: 0.12135  
Step 12: Predicted Value: 0.19269  
Step 13: Predicted Value: 0.26191  
Step 14: Predicted Value: 0.32845  
Step 15: Predicted Value: 0.39026  
Step 16: Predicted Value: 0.44613  
Step 17: Predicted Value: 0.49456  
Step 18: Predicted Value: 0.53387  
Step 19: Predicted Value: 0.56354  
Step 20: Predicted Value: 0.58294  
Step 21: Predicted Value: 0.59135  
Step 22: Predicted Value: 0.58880  
Step 23: Predicted Value: 0.57574  
Step 24: Predicted Value: 0.55260

Step 25: Predicted Value: 0.52005  
Step 26: Predicted Value: 0.47881  
Step 27: Predicted Value: 0.42968  
Step 28: Predicted Value: 0.37346  
Step 29: Predicted Value: 0.31096  
Step 30: Predicted Value: 0.24299  
Step 31: Predicted Value: 0.17038  
Step 32: Predicted Value: 0.09404  
Step 33: Predicted Value: 0.01496  
Step 34: Predicted Value: -0.06576  
Step 35: Predicted Value: -0.14685  
Step 36: Predicted Value: -0.22691  
Step 37: Predicted Value: -0.30440  
Step 38: Predicted Value: -0.37775  
Step 39: Predicted Value: -0.44539  
Step 40: Predicted Value: -0.50588  
Step 41: Predicted Value: -0.55801  
Step 42: Predicted Value: -0.60091  
Step 43: Predicted Value: -0.63402  
Step 44: Predicted Value: -0.65719  
Step 45: Predicted Value: -0.67057  
Step 46: Predicted Value: -0.67459  
Step 47: Predicted Value: -0.66986  
Step 48: Predicted Value: -0.65709  
Step 49: Predicted Value: -0.63705  
Step 50: Predicted Value: -0.61049



```
plt.plot(df['ECG'].values, label="Original ECG")  
plt.plot(range(len(df)-50, len(df)), predictions, label="Predicted ECG", linestyle="dashed")  
plt.legend()  
plt.show()
```



The plot will show the **original ECG** signal as a solid line, and the **predicted ECG** values as a dashed line for the last 50 data points.