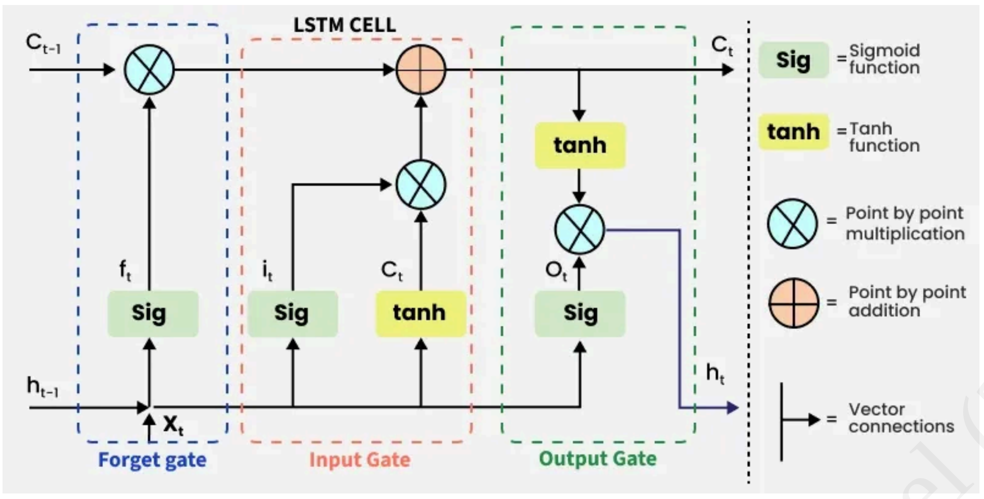
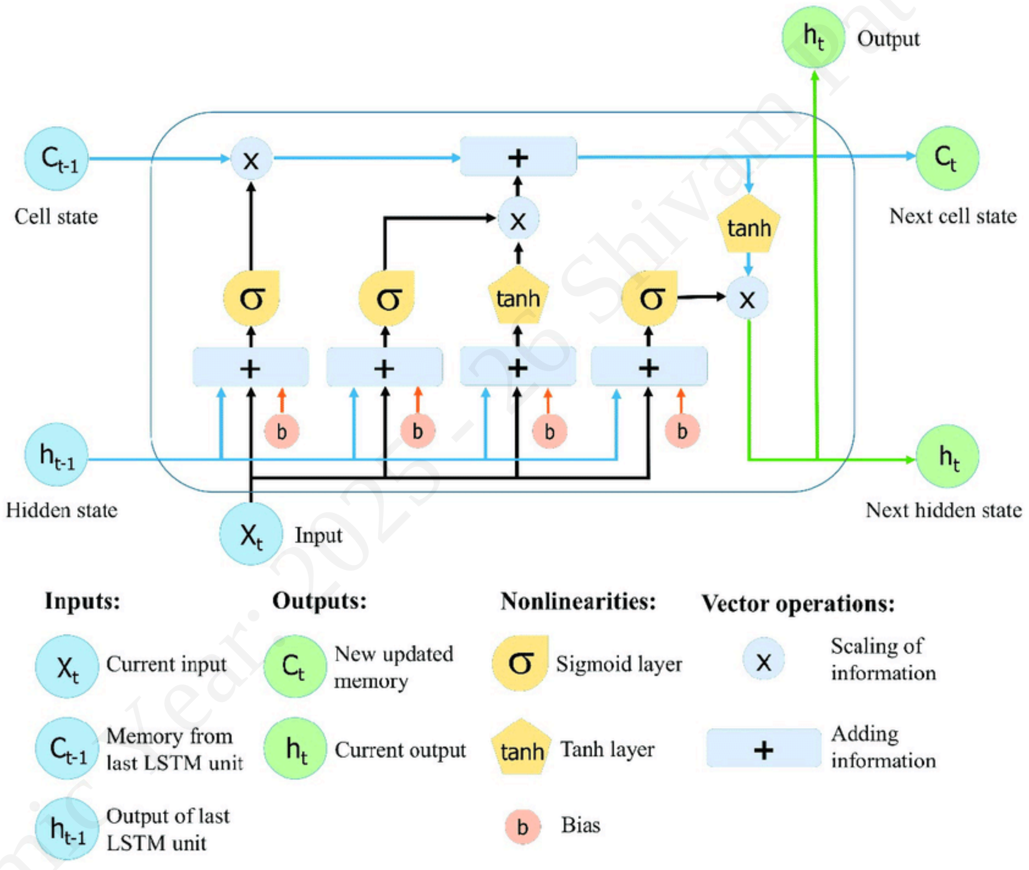
Practical 11 & 12 Implementing & Comparing : LSTM & Bi-LSTM

Theory for LSTM:-





LSTM – Long Short-Term Memory

1. What is LSTM?

LSTM (Long Short-Term Memory) is an advanced type of Recurrent Neural Network (RNN) designed to **remember information for long durations**. It solves the **vanishing gradient problem** found in simple RNNs by using a **memory cell** and **three gates** that control flow of information.

LSTM can learn patterns from **sequential data**, such as:

- text
- speech
- sensor readings
- time-series
- video frames

2. Why LSTM is Needed?

Standard RNNs forget older information because gradients shrink as they move backward in time. This makes them poor at learning long-term dependencies.

LSTM solves this by:

- using a **cell state** that can carry information for long range
- using **gates** to decide what to keep, forget, or output

Thus, LSTM can learn long sequences such as sentences, long tasks, multi-step prediction problems, etc.

---

3. LSTM Architecture (Simple Explanation)

Each LSTM cell has four main components:

1. Forget Gate

- Decides **what part of past memory to erase**.
- Removes unwanted or outdated information.

2. Input Gate

- Decides **what new information should be stored**.

3. Candidate Memory

- Creates a new possible memory value.

4. Output Gate

- Decides **what part of the cell state becomes output** (hidden state).

Cell State

- A long-term memory pathway that flows through the entire sequence.
- Helps LSTM remember information for long periods.

---

4. Variants of LSTM

1. Vanilla LSTM

Standard single-direction LSTM.

2. Bi-directional LSTM (BiLSTM)

Reads sequence from both forward and backward directions.

Use Cases: NLP, speech processing, text tagging.

---

3. Stacked / Deep LSTM

Multiple LSTM layers stacked on each other.

Use Cases: Complex sequence modeling, deep learning tasks.

---

4. Peephole LSTM

Gates also see the previous cell state.

Use Cases: Fine-grained temporal control tasks.

---

5. ConvLSTM

Uses convolution instead of matrix multiplication.

Use Cases: Videos, images, spatiotemporal data.

---

6. Attention + LSTM

LSTM + attention mechanism to improve long-term dependencies.

Use Cases: Translation, speech models, summarization.

---

5. Applications / Use Cases of LSTM

Natural Language Processing (NLP)

- Next word prediction
- Text generation
- Machine translation
- Sentiment analysis
- Chatbots

Speech Processing

- Speech recognition
- Speaker identification
- Emotion detection

Time Series Prediction

- Weather forecasting
- Stock price prediction
- Energy load forecasting
- Medical signals (ECG, EEG)

Video and Image Tasks

- Action recognition
- Video captioning
- Frame prediction (ConvLSTM)

Anomaly Detection

- Network logs
- Sensor faults
- Fraud detection

---

6. Advantages of LSTM

✓ Handles long-term dependencies

Can remember information over long sequences better than RNNs.

✓ Solves vanishing gradient

Due to cell state and gated mechanism.

✓ Works well for sequential and time-dependent data

Text, audio, time-series, etc.

✓ Flexible architecture

Can be stacked, bi-directional, attention-based.

✓ Good performance on many real-world problems

Before transformers, LSTMs dominated NLP and speech tasks.

---

7. Limitations of LSTM

✗ Slow training

More gates = more parameters.

✗ High computational cost

Compared to GRU, RNN, and Transformers.

✗ Cannot parallelize across time

Each step depends on previous step → slow on long sequences.

✗ Sometimes outperformed by Transformers

Transformers handle longer context better.

✗ Requires large datasets

To perform well in complex tasks.

---

8. Summary

- LSTM is an RNN architecture designed to store long-range information.
- It uses gates (Forget, Input, Output) and a cell state to control memory flow.
- Works in different formats: 1→1, 1→T, T→1, T→T, seq2seq.
- Highly useful in NLP, speech, time-series, and video tasks.
- Advantages: long memory, stable gradients.
- Limitations: slow, heavy, replaced by transformers in many areas.

LSTM – Sequence Mapping Types, Forward & Backprop

---

1. LSTM Forward Equations

Let:

- x_t = input at time t
- h_{t-1} = hidden state from previous step
- c_{t-1} = previous cell state
- $z_t = [h_{t-1} ; x_t]$ (concatenation)

Gates:

Forget gate:

$f_t = \text{sigmoid}(W_f * z_t + b_f)$

Input gate:

$i_t = \text{sigmoid}(W_i * z_t + b_i)$

Candidate memory:

$c\sim_t = \tanh(W_c * z_t + b_c)$

Cell state update:

$c_t = f_t * c_{t-1} + i_t * c\sim_t$

Output gate:

$o_t = \text{sigmoid}(W_o * z_t + b_o)$

Hidden state:

$h_t = o_t * \tanh(c_t)$

---

2. LSTM Sequence Mapping Types + Loss Functions

A. One-to-One (1 → 1)

Loss (classification):

$L = - \text{sum}(y_k * \log(yhat_k))$

B. One-to-Many (1 → T)

$L = - \text{sum over t} (\log(p(y_t | \text{inputs})))$

C. Many-to-One (T → 1)

$L = - \text{sum}(y_k * \log(yhat_k))$

D. Many-to-Many (T → T)

$L = - \text{sum over t,k} (y_{t,k} * \log(yhat_{t,k}))$

E. Seq2Seq (Encoder–Decoder)

$L = - \text{sum over t} (\log(p(y_t | \text{encoder\_output})))$

---

3. Backpropagation Through Time (BPTT)

Total loss:

L = sum over t (l_t)

General gradient flow:

dL/dθ = sum over t (sum over k>=t (d(l_k)/d(h_k) * d(h_k)/d(h_t) * d(h_t)/dθ))

---

4. LSTM Backpropagation

Step 1: Errors wrt outputs

delta_o_t = delta_h_t * tanh(c_t)

delta_c_t = delta_h_t * o_t * (1 - tanh(c_t)²) + delta_c_{t+1} * f_{t+1}

Step 2: Errors wrt gates

delta_c~$_t$ = delta_c_t * i_t

delta_i_t = delta_c_t * c~$_t$

delta_f_t = delta_c_t * c_{t-1}

Step 3: Gate pre-activations

delta_a^i_t = delta_i_t * i_t * (1 - i_t)

delta_a^f_t = delta_f_t * f_t * (1 - f_t)

delta_a^o_t = delta_o_t * o_t * (1 - o_t)

delta_a^c_t = delta_c~$_t$ * (1 - (c~$_t$)²)

Step 4: Parameter Gradients

dL/dW_i = sum(delta_a^i_t * z_t^T)

dL/db_i = sum(delta_a^i_t)

Similarly for W_f, W_o, W_c.

Step 5: Gradient to previous timestep

delta_z_t = W_i^T * delta_a^i_t + W_f^T * delta_a^f_t + W_o^T * delta_a^o_t + W_c^T * delta_a^c_t

Split into:

delta_h_{t-1} and delta_x_t

---

5. Backprop for Each Mapping Type

Many-to-One (T → 1)

delta_h_T = dL/dh_T

Many-to-Many (aligned)

delta_h_t = d(l_t)/d(h_t) + delta_h_{t+1}

Seq2Seq

Decoder gradients flow back to encoder final state:

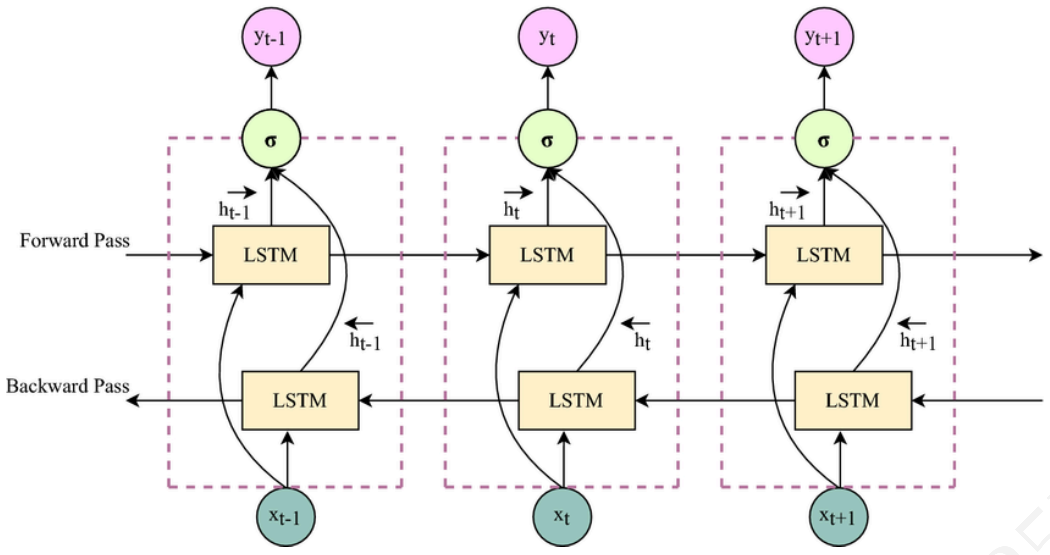
delta_h(enc) = sum_t (d(l_t)/d(h(enc)))

---

6. Practical Notes

- LSTM avoids vanishing gradient using c_t additive path.
- Use gradient clipping to avoid exploding gradients.
- Teacher forcing improves seq2seq training.

Bi-LSTM Theory:-



Bi-LSTM – Bidirectional Long Short-Term Memory

1. What is Bi-LSTM?

A Bidirectional LSTM (Bi-LSTM) is an extension of the standard LSTM that processes a sequence **in both forward and backward directions**.

It uses:

- one LSTM reading from **left** → **right**
- another LSTM reading from **right** → **left**

At each time step, outputs from both directions are combined.

This allows the network to learn **past and future context** for every position.

---

2. Why Bi-LSTM is Needed?

Standard LSTM only uses **past** information (previous tokens).

But many tasks require **future** information too. For example:

- In the sentence: "*The bank will **loan** money*" the word “loan” helps interpret “bank”.

Bi-LSTM solves this by using **context from both sides**, improving accuracy in sequence tasks.

---

3. Bi-LSTM Architecture (Simple Explanation)

A Bi-LSTM has two LSTMs:

1. **Forward LSTM** Processes sequence from t = 1 → T Produces forward hidden states: ($h^{forward}_1, h^{forward}_2, \ldots, h^{forward}_T$)
2. **Backward LSTM** Processes sequence from t = T → 1 Produces backward hidden states: ($h^{backward}_T, h^{backward}_{T-1}, \ldots, h^{backward}_1$)

Combined Output at each time step:

- Concatenation ($h_t = [\, h^{forward}_t, h^{backward}_t \,]$)

or

- Sum ($h_t = h^{forward}_t + h^{backward}_t$)

---

4. Types of Bi-LSTM (based on task)

Same mapping types as LSTM but using a bidirectional layer:

A. Bi-LSTM Many-to-Many (aligned)

Input & output have same length.

Use cases:

- POS Tagging
- NER
- Speech recognition
- Text tagging

B. Bi-LSTM Many-to-One

Sequence input → single output.

Use cases:

- Sentiment analysis
- Document classification
- Emotion recognition

C. Bi-LSTM in Seq2Seq Encoder

Encoder is often bidirectional.

Use cases:

- Machine translation
- Summarization
- Chatbots

D. Stacked Bi-LSTM

Multiple layers stacked.

Use cases:

- Deeper NLP systems
- Speech-to-text
- Video tasks

---

5. Use Cases of Bi-LSTM

Natural Language Processing

- POS tagging
- Named Entity Recognition (NER)
- Chunking
- Relation extraction
- Text classification
- Sentence boundary detection

Speech Processing

- Speech recognition
- Speaker identification
- Emotion detection

Time-Series

- Activity recognition
- Human motion prediction
- Fault detection

Biomedical

- ECG classification
- DNA sequence analysis

---

6. Advantages of Bi-LSTM

✓ Uses both past and future context

Improves prediction accuracy significantly.

✓ Better for NLP

Language meaning often depends on BOTH previous and next words.

✓ Works well on tagging and label-per-token tasks

Like NER, POS, and speech labeling.

✓ Can be combined with attention

Even stronger performance.

✓ Handles long-term dependencies (inherits LSTM advantages)

---

7. Limitations of Bi-LSTM

✗ Cannot be used for online/real-time prediction

Needs full sequence (cannot run backward without complete data).

✗ Double computation

Two LSTMs = nearly 2× parameters.

✗ Slower training

More memory and processing power required.

✗ Still sequential → cannot parallelize over time

Transformers solve this.

✗ Sometimes outperformed by Transformers in NLP

---

8. Summary

- Bi-LSTM is an LSTM that reads data in **both forward and backward directions**.
- Output is formed by **combining forward and backward hidden states**.
- Useful for tasks where context from both sides matters.
- Types include many-to-many, many-to-one, seq2seq encoder, stacked Bi-LSTM.
- Advantages: better context understanding.
- Limitations: not suitable for real-time, heavier computation.

Bi-LSTM – Forward and Backward Propagation

1. What is Bi-LSTM?

A Bi-LSTM uses two LSTMs:
1. Forward LSTM (reads from left to right: t = 1 → T)
2. Backward LSTM (reads from right to left: t = T → 1)

At each time step t, the outputs of both directions are combined.

---

2. Bi-LSTM Output

Final output at time t:
$h_t = [h_t^f ; h_t^b]$
(Concatenation of forward and backward hidden states)

---

3. Forward LSTM (Left → Right)

Let $z_t^f = [h_{t-1}^f ; x_t]$

Forget gate:
$f_t^f = \text{sigmoid}(W_f^f * z_t^f + b_f^f)$

Input gate:
$i_t^f = \text{sigmoid}(W_i^f * z_t^f + b_i^f)$

Candidate memory:
$c\sim_t^f = \tanh(W_c^f * z_t^f + b_c^f)$

Cell state update:
$c_t^f = f_t^f * c_{t-1}^f + i_t^f * c\sim_t^f$

Output gate:
$o_t^f = \text{sigmoid}(W_o^f * z_t^f + b_o^f)$

Hidden output:
$h_t^f = o_t^f * \tanh(c_t^f)$

---

4. Backward LSTM (Right → Left)

Let $z_t^b = [h_{t+1}^b ; x_t]$

Forget gate:
$f_t^b = \text{sigmoid}(W_f^b * z_t^b + b_f^b)$

Input gate:
$i_t^b = \text{sigmoid}(W_i^b * z_t^b + b_i^b)$

Candidate memory:
$c\sim_t^b = \tanh(W_c^b * z_t^b + b_c^b)$

Cell state update:
$c_t^b = f_t^b * c_{t+1}^b + i_t^b * c\sim_t^b$

Output gate:
$o_t^b = \text{sigmoid}(W_o^b * z_t^b + b_o^b)$

Hidden output:
$h_t^b = o_t^b * \tanh(c_t^b)$

---

5. Combining Both Directions

Final output at time t:

$h_t = [h_t^f ; h_t^b]$

---

6. Backpropagation in Bi-LSTM (Simple Explanation)

Step 1: Gradient from the loss

The loss gives a gradient dL/dh_t for the combined output.

Step 2: Split gradient into forward & backward parts

If output is concatenation:

dL/dh_t^f = left half of dL/dh_t
dL/dh_t^b = right half of dL/dh_t

Step 3: Backprop inside forward LSTM

Run normal LSTM backprop from t = T → 1 using dL/dh_t^f.

Step 4: Backprop inside backward LSTM

Run normal LSTM backprop from t = 1 → T using dL/dh_t^b.

Step 5: Compute parameter gradients

Each direction has its own parameters:
W^f updated using forward gradients
W^b updated using backward gradients

---

7. LSTM Cell Derivatives (Used for Both Directions)

$\text{delta}_{h,t} = dL/dh_t$

$\text{delta}_{o,t} = \text{delta}_{h,t} * \tanh(c_t)$

$\text{delta}_{c,t} = \text{delta}_{h,t} * o_t * (1 - \tanh(c_t)^2) + \text{delta}_{c,t+1} * f_{t+1}$

$delta_{i,t}$ = $delta_{c,t}$ * $c\sim_t$

$delta_{f,t}$ = $delta_{c,t}$ * c_{t-1}

$delta_{c\sim,t}$ = $delta_{c,t}$ * i_t

Activation derivatives:

delta_$a_{i,t}$ = $delta_{i,t}$ * i_t * (1 - i_t)
delta_$a_{f,t}$ = $delta_{f,t}$ * f_t * (1 - f_t)
delta_$a_{o,t}$ = $delta_{o,t}$ * o_t * (1 - o_t)
delta_$a_{c,t}$ = $delta_{c\sim,t}$ * (1 - $(c\sim_t)^2$)

✅ Difference Between LSTM and Bi-LSTM

Feature	LSTM	Bi-LSTM
Full Form	Long Short-Term Memory	Bidirectional Long Short-Term Memory
Direction of Processing	Processes sequence **in one direction** (forward)	Processes sequence in **two directions** (forward + backward)
Context Used	Uses **past context only**	Uses **past + future context**
Architecture	Single LSTM layer	Two LSTM layers: Forward + Backward
Hidden State Output	Only one hidden state: (h_t)	Two hidden states: (h_t^{fwd}), (h_t^{bwd}) combined
Accuracy	Lower for tasks needing full context	Higher because it sees entire context
Computational Cost	Lower (single LSTM)	Higher (almost 2× parameters & computation)
Memory Requirement	Less	More
Use in Real-Time Applications	Suitable (predicts step-by-step)	Not suitable (needs full sequence to run backward pass)
Best For	Time-series prediction, forecasting, signal modeling	NLP tasks like NER, POS, sentiment, speech recognition
Output Shape	Same as input direction	Usually concatenation: ([h^{fwd}, h^{bwd}])
Training Speed	Faster	Slower
Parallelization	Limited (like all RNNs)	Even more limited (two passes)

Summary

✔ LSTM

Reads sequence in *one direction* → good for tasks needing only past information.

✔ Bi-LSTM

Reads sequence in *two directions* → uses past + future context → better for NLP and tagging tasks.

When to Use What?

Use **LSTM** when:

- Real-time prediction is required
- You only have past data (forecasting)
- Lightweight model needed

Use **Bi-LSTM** when:

- Entire sequence is available
- Task needs context from both sides
- You want higher accuracy in NLP

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, PowerTransformer
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Bidirectional, Dense, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow import keras
# ----------------------------
# 1) Load data (Local CSV)
# ----------------------------
CSV_PATHS = ["AirPassengers.csv","AirPassengers.csv", "airline-passengers.csv"]  # try in this order
TARGET = "#Passengers"
DATE_COL = "Month"
```

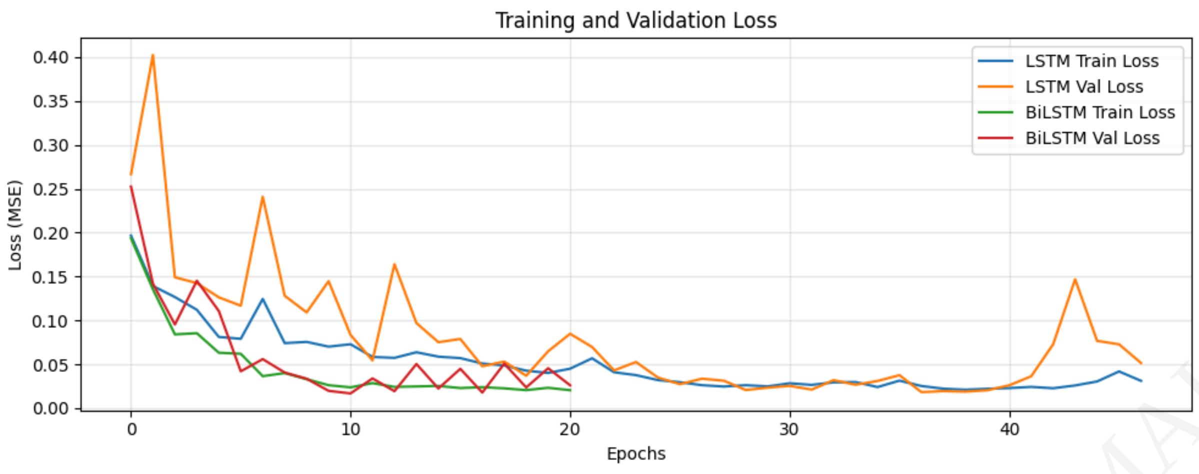
```
df = None
for p in CSV_PATHS:
    try:
        df = pd.read_csv(p)
        print(f"Loaded: {p}")
        break
    except Exception:
        pass

if df is None:
    raise FileNotFoundError(
        "Could not find CSV. Place 'AirPassengers.csv' or 'airline-passengers.csv' in the working directory.\n"
        "Expected columns: Month, Passengers"
    )

# normalize columns
df.columns = [c.strip() for c in df.columns]

# Basic checks / coercions
if DATE_COL not in df.columns:
    # try fallback common variants
    for c in df.columns:
        if c.lower() in ["month", "date", "time"]:
            DATE_COL = c
            break

if TARGET not in df.columns:
    # try fallback common variants
    for c in df.columns:
        if c.lower() in ["passengers", "passenger", "value"]:
            TARGET = c
            break

# Parse datetime and sort
df[DATE_COL] = pd.to_datetime(df[DATE_COL], errors="coerce")
df = df.dropna(subset=[DATE_COL, TARGET]).sort_values(DATE_COL).reset_index(drop=True)

# Keep only date + target
df_small = df[[DATE_COL, TARGET]].copy()
df_small[TARGET] = df_small[TARGET].astype("float32")

print(df_small.head())

# ----------------------------
# 2) Scale & window (12 months)
# ----------------------------
values = df_small[[TARGET]].values
#scaler = StandardScaler(with_std=True)
#scaler = RobustScaler()
scaler = PowerTransformer(method='yeo-johnson')
#scaler = MinMaxScaler()
scaled = scaler.fit_transform(values)

def make_windows(series, window=12):
    X, y = [], []
    for i in range(window, len(series)):
        X.append(series[i-window:i, 0])
        y.append(series[i, 0])
    X = np.array(X)
    y = np.array(y)
    return X.reshape((X.shape[0], X.shape[1], 1)), y

WINDOW = 12*3  # one year of monthly lags
EPOCHS = 1000
X, y = make_windows(scaled, window=WINDOW)
time_index = df_small[DATE_COL].iloc[WINDOW:].reset_index(drop=True)

# Chronological split: 70/15/15 (small dataset)
n = len(X)
n_train = int(n * 0.7)
n_val   = int(n * 0.15)
n_test  = n - n_train - n_val

X_train = X[:n_train]
y_train = y[:n_train]
X_val   = X[n_train:n_train+n_val]
y_val   = y[n_train:n_train+n_val]
X_test  = X[n_train+n_val:]
y_test  = y[n_train+n_val:]

time_test = time_index[n_train+n_val:].reset_index(drop=True)

# ----------------------------
# 3) Models
# ----------------------------
# Hypermarams: activation, recurrent_activation, use_bias, kernel_initializer, recurrent_initializer, bias_initializer, unit_forget_bias, kerne
def build_lstm(window,
               units=1024,
               activation="tanh",
               recurrent_activation="hard_sigmoid",
               use_bias=True,
               kernel_initializer="glorot_uniform",
               recurrent_initializer="orthogonal",
               bias_initializer="zeros",
               unit_forget_bias=True,
               dropout=0.0,
               recurrent_dropout=0.0,
               optimizer=keras.optimizers.AdamW(learning_rate=0.001),
               loss="mse"):
```

```
    """
    LSTM with configurable hyperparameters. Keep return_sequences=False to match scalar target.
    """
    m = Sequential([
        Input(shape=(window, 1)),
        LSTM(
            units,
            activation=activation,
            recurrent_activation=recurrent_activation,
            use_bias=use_bias,
            kernel_initializer=kernel_initializer,
            recurrent_initializer=recurrent_initializer,
            bias_initializer=bias_initializer,
            unit_forget_bias=unit_forget_bias,
            dropout=dropout,
            recurrent_dropout=recurrent_dropout,
            return_sequences=False
        ),
        Dense(1)
    ])
    m.compile(optimizer=optimizer, loss=loss)
    return m

def build_bilstm(window,
                 units=1024,
                 activation="tanh",
                 recurrent_activation="hard_sigmoid",
                 use_bias=True,
                 kernel_initializer="glorot_uniform",
                 recurrent_initializer="orthogonal",
                 bias_initializer="zeros",
                 unit_forget_bias=True,
                 dropout=0.0,
                 recurrent_dropout=0.0,
                 optimizer=keras.optimizers.AdamW(learning_rate=0.001),
                 loss="mse"):
    """
    Bidirectional LSTM with the same hyperparameter set. return_sequences=False -> single vector output.
    """
    m = Sequential([
        Input(shape=(window, 1)),
        Bidirectional(
            LSTM(
                units,
                activation=activation,
                recurrent_activation=recurrent_activation,
                use_bias=use_bias,
                kernel_initializer=kernel_initializer,
                recurrent_initializer=recurrent_initializer,
                bias_initializer=bias_initializer,
                unit_forget_bias=unit_forget_bias,
                dropout=dropout,
                recurrent_dropout=recurrent_dropout,
                return_sequences=False
            )
        ),
        Dense(1)
    ])
    m.compile(optimizer=optimizer, loss=loss)
    return m

lstm = build_lstm(WINDOW)
bilstm = build_bilstm(WINDOW)

early = EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True)

print("\nTraining LSTM...")
history_lstm = lstm.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=EPOCHS, batch_size=8, verbose=0,
    callbacks=[early]
)
print(lstm.summary())
print("Training BiLSTM...")
history_bilstm = bilstm.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=EPOCHS, batch_size=8, verbose=0,
    callbacks=[early]
)
print(bilstm.summary())
#   plot history of training/validation loss
plt.figure(figsize=(10, 4))
plt.plot(history_lstm.history['loss'], label='LSTM Train Loss')
plt.plot(history_lstm.history['val_loss'], label='LSTM Val Loss')
plt.plot(history_bilstm.history['loss'], label='BiLSTM Train Loss')
plt.plot(history_bilstm.history['val_loss'], label='BiLSTM Val Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
# ----------------------------
# 4) Predict & invert scaling
# ----------------------------
pred_lstm   = lstm.predict(X_test, verbose=0)
```

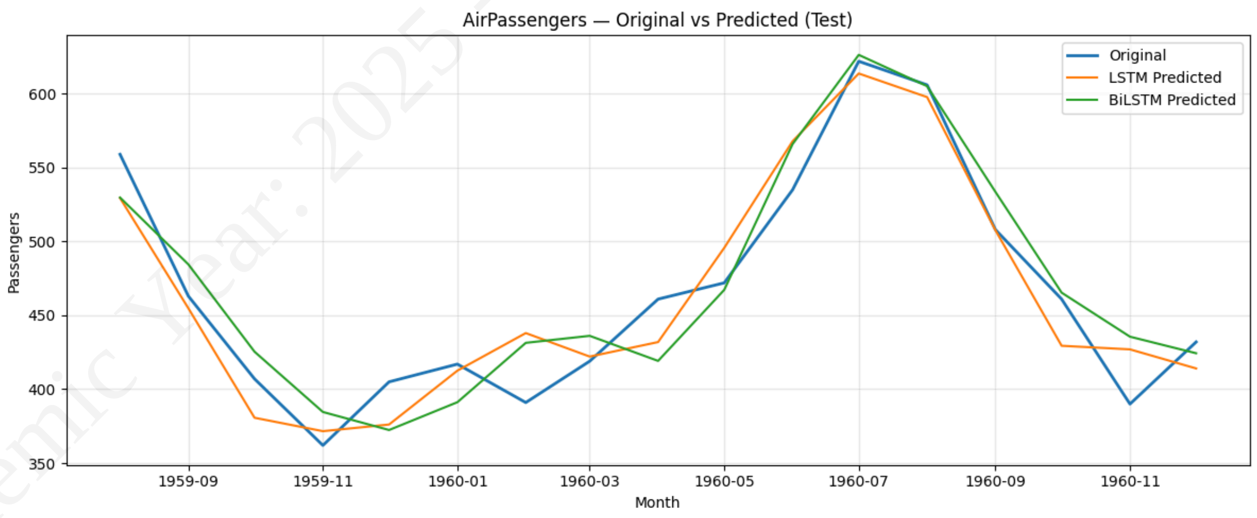
```python
pred_bilstm = bilstm.predict(X_test, verbose=0)

orig_test   = scaler.inverse_transform(y_test.reshape(-1, 1)).flatten()
pred_lstm   = scaler.inverse_transform(pred_lstm).flatten()
pred_bilstm = scaler.inverse_transform(pred_bilstm).flatten()

# ----------------------------
# 5) Metrics
# ----------------------------
def mae(a, b): return float(np.mean(np.abs(a - b)))
def rmse(a, b): return float(np.sqrt(np.mean((a - b)**2)))
def mse(a, b): return float(np.mean((a - b)**2))

print("\n=== Test Metrics (Passengers) ===")
print(f"LSTM   → MAE: {mae(orig_test, pred_lstm):.2f} | RMSE: {rmse(orig_test, pred_lstm):.2f} | MSE: {mse(orig_test, pred_lstm):.2f}")
print(f"BiLSTM → MAE: {mae(orig_test, pred_bilstm):.2f} | RMSE: {rmse(orig_test, pred_bilstm):.2f} | MSE: {mse(orig_test, pred_bilstm):.2f}")

# ----------------------------
# 6) Original vs Predicted DataFrame (test)
# ----------------------------
result_df = pd.DataFrame({
    "Month": time_test,
    "Original_Passengers": orig_test,
    "LSTM_Pred": pred_lstm,
    "BiLSTM_Pred": pred_bilstm
})
print("\nHead of Original vs Predicted (test):")
print(result_df.head(12))

# ----------------------------
# 7) Plots
# ----------------------------
plt.figure(figsize=(12, 5))
plt.plot(result_df["Month"], result_df["Original_Passengers"], label="Original", linewidth=2)
plt.plot(result_df["Month"], result_df["LSTM_Pred"], label="LSTM Predicted")
plt.plot(result_df["Month"], result_df["BiLSTM_Pred"], label="BiLSTM Predicted")
plt.title("AirPassengers – Original vs Predicted (Test)")
plt.xlabel("Month")
plt.ylabel("Passengers")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Optional: zoom into last N test points
N = 24
if len(result_df) > N:
    plt.figure(figsize=(12, 5))
    plt.plot(result_df["Month"].iloc[-N:], result_df["Original_Passengers"].iloc[-N:], label="Original", linewidth=2)
    plt.plot(result_df["Month"].iloc[-N:], result_df["LSTM_Pred"].iloc[-N:], label="LSTM Predicted")
    plt.plot(result_df["Month"].iloc[-N:], result_df["BiLSTM_Pred"].iloc[-N:], label="BiLSTM Predicted")
    plt.title(f"Zoom: Last {N} Months – Original vs Predicted")
    plt.xlabel("Month")
    plt.ylabel("Passengers")
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
```

```
Loaded: AirPassengers.csv
       Month  #Passengers
0 1949-01-01        112.0
1 1949-02-01        118.0
2 1949-03-01        132.0
3 1949-04-01        129.0
4 1949-05-01        121.0

Training LSTM...
Model: "sequential_4"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                        ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ lstm_4 (LSTM)                       │ (None, 1024)                │       4,202,496 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_4 (Dense)                     │ (None, 1)                   │           1,025 │
└─────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 12,610,565 (48.11 MB)
 Trainable params: 4,203,521 (16.04 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 8,407,044 (32.07 MB)
None
Training BiLSTM...
Model: "sequential_5"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                        ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ bidirectional_2 (Bidirectional)     │ (None, 2048)                │       8,404,992 │
├─────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_5 (Dense)                     │ (None, 1)                   │           2,049 │
└─────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 25,221,125 (96.21 MB)
 Trainable params: 8,407,041 (32.07 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 16,814,084 (64.14 MB)
```

```
None
```



```
WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x00
0001DAB3790400> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.fun
ction repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define yo
ur @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), plea
se refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  mor
e details.
WARNING:tensorflow:6 out of the last 6 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x00
0001DA6F7E7420> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.fun
ction repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define yo
ur @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), plea
se refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  mor
e details.

=== Test Metrics (Passengers) ===
LSTM   → MAE: 20.37 | RMSE: 24.39 | MSE: 594.85
BiLSTM → MAE: 21.97 | RMSE: 25.87 | MSE: 669.18

Head of Original vs Predicted (test):
        Month  Original_Passengers   LSTM_Pred  BiLSTM_Pred
0  1959-08-01           558.999756  529.414062   529.762024
1  1959-09-01           462.999969  454.774261   484.482880
2  1959-10-01           406.999817  380.696075   425.390594
3  1959-11-01           361.999664  371.607452   384.608032
4  1959-12-01           405.000244  376.116028   372.396027
5  1960-01-01           416.999939  412.594513   391.236084
6  1960-02-01           391.000061  438.019501   431.399139
7  1960-03-01           419.000031  422.100769   436.149323
8  1960-04-01           461.000000  431.930634   419.176605
9  1960-05-01           472.000061  495.714752   467.271088
10 1960-06-01           534.999756  567.873474   565.956482
11 1960-07-01           621.999878  613.729980   626.338806
```


AirPassengers — Original vs Predicted (Test)
Original
LSTM Predicted
BiLSTM Predicted
Passengers
Month