SEN605 · WEEK 6

Deep Learning for Financial Time Series:

LSTM and GRU

RNN · LSTM cell · GRU · Sequence preparation · Training with PyTorch · Attention mechanism

Instructor: Assoc. Prof. Dr. Onur Polat

Course: SEN605 — Elective

Graduate School of Informatics · Systems Engineering Program · Hacettepe University

Week 6 — Agenda

What we will cover today

01 **Limits of Classical ML and Motivation for RNN**

02 **LSTM Cell — Gates and Memory**

03 **GRU (Gated Recurrent Unit)**

04 **Sequence Data Preparation (Sliding Window)**

05 **LSTM Model with PyTorch**

06 **Training Loop and Early Stopping**

07 **Attention Mechanism (Introduction)**

08 **Application: USD/TRY LSTM Forecast**

From Week 5 to Week 6

Why LSTM?

✅ Week 5 Completed

XGBoost direction forecast for BIST-100 achieved AUC ≈ 0.60–0.65 — beating the market is hard but not impossible.

🧠 How LSTM Differs

Consumes sequence data in raw form and learns to filter its own history through gates — feature engineering shrinks dramatically.

⚠️ Classical ML Limitation

XGBoost requires manual lag features; it cannot learn the internal structure (long-term dependencies) of the time series.

🎯 Today's Goal

Build an LSTM model in PyTorch; forecast USD/TRY log-return using a 5-day window.

Deep Learning ≠ always better: LSTM's success depends on the length and quality of the data. For daily financial series, LSTM is not always superior to XGBoost. However, it provides a clear advantage for intraday tick data, combinations of text + price, or very long sequences.

RNN and the Vanishing-Gradient Problem

Why LSTM was invented

Vanilla RNN: $h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$ **Output:** $y_t = W_y h_t + b_y$

Gradient flow: as we step back from t to t−3, the signal weakens (×W_h each step) → vanishing gradient



Vanishing Gradient

During BPTT (Backprop Through Time), the gradient shrinks exponentially over long sequences. tanh derivative ≤ 1; product over many steps → 0. Long-term dependencies cannot be learned.

Exploding Gradient

Controlled with gradient clipping. The primary motivation for LSTM is the vanishing-gradient problem — the gating mechanism lets gradients flow more freely.

LSTM Solution

The cell state C_t provides an additional information pathway. Gates (input, forget, output) learn how much information passes — gradient flows without being cut off.

SECTION 1

LSTM Cell — Gates and Memory

Hochreiter & Schmidhuber (1997) — long short-term memory

LSTM Cell — 4 Gates, 2 States

At each step: what to forget, what to learn, what to output

← Cell State C_t (long-term memory)

f_t Forget Gate

What to forget?
f_t=0: forget all
f_t=1: keep all

$\sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

i_t Input Gate

What to learn?
$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

$\sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

C_t Cell Update

Keep old +
add new

$f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$

o_t Output Gate

What to output?
$h_t = o_t \odot \tanh(C_t)$

$\sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

LSTM Equations — Summary

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{Forget}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{Input}$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \quad \text{Candidate}$$
$$\mathbf{C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t} \quad \textbf{Cell}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad \text{Output}$$
$$\mathbf{h_t = o_t \odot \tanh(C_t)} \quad \textbf{Hidden}$$

$\odot$: element-wise (Hadamard) product
σ: sigmoid $[0,1]$ tanh: $[-1,1]$
Parameters: $4 \times (n_h^2 + n_x n_h + n_h)$

SECTION 2

GRU (Gated Recurrent Unit)

Cho et al. (2014) — a simplified version of LSTM

GRU — Gated Recurrent Unit

2 gates, 1 state — faster than LSTM, similar performance

GRU equations: $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$ $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$ $\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t])$ $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

Criterion	GRU	LSTM
Number of gates	2 (Update + Reset)	4 (Forget+Input+Cell+Output)
Number of states	1 (h_t)	2 (h_t + C_t)
Parameters	$3 \times (n_h^2 + n_x n_h + n_h)$	$4 \times (n_h^2 + n_x n_h + n_h)$
Training speed	Faster (~25-30%)	Slower
Long-term dependency	Usually as good as LSTM	Theoretical advantage
Preferred for	Small-to-medium datasets	Large data, deep sequences

SECTION 3

Sequence Data Preparation

Sliding window · Normalization · Train/Val/Test split

Sequence Data Preparation

LSTM input shape: [batch, seq_len, features]

Sliding Window — building sequences with seq_len=5

Raw data:

0.8	-1.2	0.5	1.1	-0.7	0.9	-0.4	1.3	0.2	-0.8	0.6	1.0
t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12

Window 1:	0.8	-1.2	0.5	1.1	-0.7	→ **y = 0.9**
Window 2:	-1.2	0.5	1.1	-0.7	0.9	→ **y = -0.4**
Window 3:	0.5	1.1	-0.7	0.9	-0.4	→ **y = 1.3**

```
# Sliding window code


def create_sequences(data, seq_len, horizon=1):
    X, y = [], []
    for i in range(len(data)-seq_len-horizon+1):
        X.append(data[i:i+seq_len])      # window
        y.append(data[i+seq_len:i+seq_len+horizon])
    return np.array(X), np.array(y)

X_seq, y_seq = create_sequences(scaled, seq_len=20)
```

Tensor Shapes

X: (N, seq_len, n_features)
y: (N,) or (N, horizon)

Example: 1000 days, seq=20,
5 features → (980, 20, 5)

DataLoader: batch_size=32

Normalization and Time-Based Split

Critical preprocessing steps for LSTM

Normalization Strategies

- MinMaxScaler: [0,1] range — for price forecasting
- StandardScaler: μ=0, σ=1 — for return forecasting
- Critical: **fit the scaler ONLY on the training set**
- Inverse transform: predict → inverse_transform
- Each feature must be scaled separately (multivariate)
- Batch normalization: inside the model — speed boost

Train / Val / Test Split

- 70% train / 15% validation / 15% test (chronological)
- Val set: for early stopping
- Test set: final evaluation only — never optimize on it
- Purging: leave a gap (e.g. 5 days) before validation starts

```
# Normalize and split
```

```
import yfinance as yf
import numpy as np, pandas as pd
from sklearn.preprocessing import MinMaxScaler

px = yf.download("USDTRY=X",
    start="2018-01-01", end="2024-12-31",
    auto_adjust=True)["Close"].dropna()
r = np.log(px / px.shift(1))

# Chronological split
n = len(df)
n_tr, n_val = int(n*0.70), int(n*0.15)
train, val, test = df.iloc[:n_tr], \
    df.iloc[n_tr:n_tr+n_val], df.iloc[n_tr+n_val:]

scaler = MinMaxScaler()
train_s = scaler.fit_transform(train)
val_s = scaler.transform(val)
test_s = scaler.transform(test)
```

SECTION 4

LSTM Model with PyTorch

nn.LSTM · Model architecture · DataLoader · Training loop

PyTorch LSTM Model Definition

Flexible architecture with nn.Module


```python
# PyTorch LSTM model

import torch
import torch.nn as nn

class FinancialLSTM(nn.Module):
    def __init__(self, input_size, hidden_size,
                 num_layers, dropout=0.2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers>1 else 0,
        )
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.dropout(out[:, -1, :])
        return self.fc(out).squeeze(-1)
```

Key nn.LSTM Parameters

- input_size: number of features
- hidden_size: 32–256
- num_layers: 2 typical
- batch_first: (batch, seq, feat)
- dropout: 0.1–0.5
- bidirectional: for classification

Architecture Choices

- Vanilla LSTM: single layer + FC
- Stacked LSTM: 2–3 layers
- LSTM + Dropout
- LSTM + BatchNorm
- Bidirectional (classification)
- LSTM + Attention (long seq.)

Training Loop and Early Stopping

DataLoader · Optimizer · Loss · Early Stopping


```
# Training loop



optimizer = torch.optim.Adam(model.parameters(),
    lr=1e-3, weight_decay=1e-5)
criterion = nn.MSELoss()
scheduler = torch.optim.lr_scheduler.\
    ReduceLROnPlateau(optimizer, patience=5)

best_val, patience_cnt, PATIENCE = 1e9, 0, 15
for epoch in range(200):
    model.train()
    for xb, yb in tr_loader:
        optimizer.zero_grad()
        loss = criterion(model(xb), yb)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(
            model.parameters(), max_norm=1.0)
        optimizer.step()

    model.eval()
    with torch.no_grad():
        val_loss = criterion(model(X_val), y_val).item()
    if val_loss < best_val - 1e-5:
        best_val = val_loss; patience_cnt = 0
        torch.save(model.state_dict(), "best.pt")
    else:
        patience_cnt += 1
        if patience_cnt >= PATIENCE: break
```

Optimizer & Loss

- Regression: MSE / HuberLoss
- Classification (direction): BCEWithLogits
- Optimizer: Adam (lr=1e-3) / AdamW
- Scheduler: ReduceLROnPlateau
- Gradient clipping max_norm=1.0

Early Stopping

- Monitor val loss — stop when no improvement in `patience` steps
- Save best model weights
- patience=10-20 typical
- min_delta=1e-5
- NEVER use test set for ES

SECTION 5

Attention Mechanism

Which time step to pay attention to? — Gateway to Transformers

Attention Mechanism — Add-on to LSTM

For long sequences, which past step matters?

Core idea:

Over LSTM time-step outputs H = [h_1, h_2, …, h_n], compute learned weights (attention scores). Produce the final prediction as the weighted sum of these hidden states.

```
# Attention LSTM


class AttentionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size,
                 num_layers=2, dropout=0.2):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size,
            num_layers, batch_first=True,
            dropout=dropout if num_layers>1 else 0)
        self.attention = nn.Linear(hidden_size, 1)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        h_all, _ = self.lstm(x)
        e = self.attention(h_all)     # (B,T,1)
        alpha = torch.softmax(e, dim=1)
        context = (alpha * h_all).sum(dim=1)
        out = self.dropout(context)
        return self.fc(out).squeeze(-1)
```

Attention Steps

1. LSTM → $H \in \mathbb{R}(T \times n_h)$
2. Score: $e_t = \tanh(W_a h_t + b_a)$
3. Normalize: $\alpha_t = \text{softmax}(e_t)$
4. Context: $c = \Sigma\, \alpha_t h_t$
5. Output: $\hat{y} = FC(c)$

Financial Interpretation

- Model learns which past days matter
- Crisis periods / major news → high attention
- α_t can be visualized — interpretability restored

SECTION 6

Application: USD/TRY LSTM Forecast

Full deep-learning pipeline, step by step

USD/TRY Pipeline — Steps 1–2: Data & Sequences

Lab application

Data Download and Feature Preparation

```
import yfinance as yf, numpy as np, pandas as pd
from sklearn.preprocessing import MinMaxScaler
import torch, torch.nn as nn

usdtry = yf.download("USDTRY=X", start="2018-01-01",
    end="2024-12-31", auto_adjust=True)["Close"].dropna()
r = np.log(usdtry / usdtry.shift(1))

df = pd.DataFrame({
    "ret":     r,
    "ret_5d":  np.log(usdtry/usdtry.shift(5)),
    "vol_10d": r.rolling(10).std(),
    "ma_ratio": usdtry/usdtry.rolling(20).mean()-1,
}).dropna()
```

Normalization and Sequence Creation

```
SEQ_LEN = 20                                # 20-day window
n = len(df); n_tr, n_val = int(n*0.70), int(n*0.15)
train, val, test = df.iloc[:n_tr], df.iloc[n_tr:n_tr+n_val], df.iloc[n_tr+n_val:]

scaler = MinMaxScaler()
train_s = scaler.fit_transform(train)    # fit ONLY on train
val_s = scaler.transform(val); test_s = scaler.transform(test)

def create_seq(data, sl):
    X, y = [], []
    for i in range(len(data)-sl):
        X.append(data[i:i+sl]); y.append(data[i+sl, 0])
    return np.array(X), np.array(y)
```

USD/TRY Pipeline — Steps 3–4: Train & Evaluate

Lab application

Build Model and Train

```
model = AttentionLSTM(input_size=4, hidden_size=64, num_layers=2, dropout=0.2)
loader = DataLoader(TensorDataset(Xtr_t, ytr_t), batch_size=32, shuffle=False)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
criterion = nn.MSELoss()

best_val, patience_cnt, PATIENCE = 1e9, 0, 20
for epoch in range(300):
    model.train()
    for xb, yb in loader:
        optimizer.zero_grad(); criterion(model(xb), yb).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0); optimizer.step()
    model.eval()
    with torch.no_grad():
        vl = criterion(model(Xval_t), yval_t).item()
```

Test Evaluation and Visualization

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
model.eval()
with torch.no_grad():
    y_hat = model(Xte_t).numpy()

# Inverse-transform only the target column
pad = np.zeros((len(y_hat), 4)); pad[:, 0] = y_hat
y_pred_inv = scaler.inverse_transform(pad)[:, 0]

mae = mean_absolute_error(y_true_inv, y_pred_inv)
rmse = np.sqrt(mean_squared_error(y_true_inv, y_pred_inv))
print(f"MAE: {mae:.6f}  RMSE: {rmse:.6f}")
```

LSTM Hyperparameter Guide

Practical starting values for financial time series

Hyperparameter	Small (<500)	Medium (500–2000)	Large (>2000)	Description
seq_len	10-15	15-30	30-60	Window — larger = more context
hidden_size	32	64-128	128-256	Hidden size — larger = more capacity
num_layers	1	2	2-3	Number of layers — 3+ rare
dropout	0.1	0.2-0.3	0.3-0.5	Between stacked layers
learning_rate	1e-3	1e-3 – 5e-4	1e-4 – 5e-4	Adam — use scheduler
batch_size	16	32	32-64	Smaller = better generalization
patience	10	15	20	Early stopping tolerance

All Models Comparison — From ARIMA to LSTM

Weeks 3–6 summary: which model for when?

Model	Speed	Data Need	Interpretable	Sequential	Nonlinear	Strength
ARIMA	⚡⚡⚡	Low	High	✗	✗	Explainable baseline
GARCH	⚡⚡⚡	Low	High	✗	✗	Standard volatility model
XGBoost	⚡⚡	Medium	Medium	⚠	✓	Tabular, feature importance
LSTM	⚡	High	Low	✓	✓	Long dependency, raw series
GRU	⚡⚡	Medium	Low	✓	✓	Fast LSTM alternative
Att-LSTM	⚡	High	Medium	✓	✓	Interpretable attention weights

Overfitting and Regularization in LSTM

Deep models — particularly dangerous

Dropout

```
# only works for num_layers > 1
self.lstm = nn.LSTM(..., dropout=0.3)
# extra dropout before FC
self.dropout = nn.Dropout(0.3)
out = self.dropout(lstm_out[:,-1,:])
```

Weight Decay (L2)

```
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=1e-3,
    weight_decay=1e-5  # L2 coefficient
)
```

Early Stopping

```
if val_loss < best_val - 1e-5:
    best_val = val_loss
    torch.save(model.state_dict(), "best.pt")
    patience_cnt = 0
else:
    patience_cnt += 1
    if patience_cnt >= 15: break
```

Gradient Clipping

```
# after backward, before step
torch.nn.utils.clip_grad_norm_(
    model.parameters(),
    max_norm=1.0
)
```

Multi-Output and Hybrid Models

Return + direction · LSTM + GARCH


```python
# Multi-output LSTM


class MultiOutputLSTM(nn.Module):
    def __init__(self, input_size, hidden_size,
                 num_layers=2, dropout=0.2):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size,
            num_layers, batch_first=True,
            dropout=dropout if num_layers>1 else 0)
        self.dropout = nn.Dropout(dropout)
        self.reg_head = nn.Linear(hidden_size, 1)  # return
        self.cls_head = nn.Linear(hidden_size, 1)  # direction

    def forward(self, x):
        out, _ = self.lstm(x)
        h = self.dropout(out[:, -1, :])
        return self.reg_head(h).squeeze(-1), \
               self.cls_head(h).squeeze(-1)

# combined loss
def combined_loss(rp, dp, rt, dt, alpha=0.5, beta=0.5):
    return alpha*mse_loss(rp,rt) + beta*bce_loss(dp,dt)
```

Multi-Output

- Predict return and direction from one model
- Shared backbone + two heads
- Loss: α·MSE + β·BCE
- Benefit: shared feature learning
- Use: regression + classification together

LSTM + GARCH Hybrid

- LSTM: conditional mean (return)
- GARCH on LSTM residuals (volatility)
- Two-stage: train LSTM → fit GARCH on residuals
- Can outperform pure GARCH
- Sequential dependency — no parallel training

Transformer and Financial LLMs (Overview)

After Week 6 — current research frontiers

2021

Temporal Fusion Transformer (TFT)

Lim et al. — Transformer specialized for multivariate time series. Combines static and dynamic variables. Powerful for covariate-rich features.

AAAI 2021 Best Paper candidate

2021

Informer (long sequences)

Zhou et al. — O(T log T) instead of standard Transformer's O(T^2). Enables very long-horizon (500+ days) forecasts. Self-attention applied to finance.

AAAI 2021 Best Paper

2023

FinBERT / FinGPT

LLMs pre-trained on financial text (news, analyst reports). Combine news sentiment with price prediction.

Finance NLP: text + price

2024

Time-Series Foundation Models

TimeGPT, Moirai, Chronos: large-scale pre-training enables zero-shot time-series forecasting. Fine-tuning for domain tasks.

2024 research focus

GPU Acceleration and Practical Tools

Efficient training for large models


```
# GPU training
```

```
import torch

device = torch.device(
    "cuda" if torch.cuda.is_available() else
    "mps"  if torch.backends.mps.is_available() else
    "cpu")
print(f"Using device: {device}")

model = FinancialLSTM(4, 64, 2).to(device)
X_tr_d = X_tr_t.to(device); y_tr_d = y_tr_t.to(device)

# Mixed precision (CUDA only)
scaler_amp = torch.cuda.amp.GradScaler()
with torch.cuda.amp.autocast():
    pred = model(xb); loss = criterion(pred, yb)
scaler_amp.scale(loss).backward()
scaler_amp.step(optimizer); scaler_amp.update()
```

GPU / Device

- Install: pip install torch --index-url ...cu121
- .to(device) for model + tensors
- Google Colab: free T4 GPU
- Kaggle Notebooks: P100, 30h/week
- Apple Silicon: device = "mps"

Performance Tools

- torch.compile() — auto optimization
- Mixed precision → memory ~50% less
- TensorBoard: loss/metric viz
- Weights & Biases (wandb)

Time-Series Decomposition with Deep Learning

Trend · Seasonality · Residual components via LSTM

N-BEATS and N-HiTS (overview)

N-BEATS (Oreshkin et al., 2020): interpretable time-series forecasting. Learns trend + seasonal components in separate blocks. Both ML and statistical interpretation possible.
N-HiTS (Challu et al., 2023): hierarchical sampling for long-horizon forecasting. Faster than Transformer, fewer parameters.
Financial use: macro series, seasonal commodities, implied-volatility surfaces. For daily equities, LSTM/GRU is usually sufficient.

```
# STL + LSTM Hybrid



from statsmodels.tsa.seasonal import STL

# STL decomposition
stl = STL(df["close"], period=252, seasonal=13).fit()
trend    = stl.trend
seasonal = stl.seasonal
resid    = stl.resid

# Model residuals with LSTM
# Forecast: trend + seasonal + LSTM(resid)
```

Practical Notes

- Daily equity prices: weak seasonality — STL poor fit
- Energy prices: strong seasonality — STL ideal
- Inflation / macro: trend-dominant — differencing enough
- After removing deterministic parts, LSTM has cleaner signal
- Rule of thumb: start with ARIMA + GARCH, add LSTM if needed

LSTM Backtesting and Model Deployment

Model serialization for live forecasting


```
# Online retraining



def monthly_retrain(full_data, model, scaler,
                    seq_len=20, n_epochs=50):
    """Update the model as new monthly data arrives"""
    last_2y = full_data.iloc[-504:]  # ~2x252
    train_s = scaler.fit_transform(last_2y)
    X, y = create_sequences(train_s, seq_len)
    Xt, yt = torch.FloatTensor(X), torch.FloatTensor(y[:,0])
    loader = DataLoader(TensorDataset(Xt, yt),
        batch_size=32, shuffle=False)
    model.train()
    optimizer = torch.optim.Adam(
        model.parameters(), lr=1e-4)   # smaller lr
    for _ in range(n_epochs):
        for xb, yb in loader:
            optimizer.zero_grad()
            criterion(model(xb), yb).backward()
            optimizer.step()
    torch.save(model.state_dict(), "updated.pt")
    return model
```

Concept Drift

- Markets change — models decay
- Rolling retrain monthly
- Expanding window
- Sliding-window retrain
- Adaptive: update at every new point

Model Serialization

- torch.save(model.state_dict(), "m.pt")
- ONNX export: cross-platform
- TorchScript: production C++
- FastAPI + PyTorch: REST deploy

Key Publications and Resources

Foundations of the deep-learning × finance literature

1997 — **Hochreiter & Schmidhuber — LSTM**

Long Short-Term Memory. Neural Computation. The original LSTM paper — mathematical foundation of the gating mechanism.

2014 — **Cho et al. — GRU**

Learning Phrase Representations using RNN Encoder–Decoder. EMNLP 2014. Original introduction of GRU.

2017 — **Vaswani et al. — Attention**

Attention Is All You Need. NeurIPS 2017. Foundational Transformer paper — basis of financial LLMs.

2021 — **Lim et al. — TFT**

Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting. Int. J. Forecasting.

2023 — **Challu et al. — N-HiTS**

N-HiTS: Neural Hierarchical Interpolation for Time Series Forecasting. AAAI 2023.

2023 — **FinGPT**

FinGPT: Open-Source Financial Large Language Models. Pre-trained on financial text.

Week 6 Summary

What we learned — end-of-course review

🧠 RNN and LSTM

- Vanishing gradient → LSTM gates
- 4 gates: forget, input, cell, output
- h_t: short memory · C_t: long memory

⚡ GRU vs Comparison

- 2 gates, 1 state — fewer parameters
- Similar performance, faster training
- Preferred for small-medium data

🔧 PyTorch Pipeline

- Sliding window → (N, seq, feat) tensor
- Scaler: fit ONLY on train
- Early stopping + gradient clipping

📊 Attention and Hybrid

- Attention: which step matters → α_t
- LSTM+GARCH: mean + volatility together
- Multi-output: return + direction

🏁 ***Course completed! Weeks 1–6: Data prep → ARIMA → GARCH → ML → LSTM — full financial analytics pipeline***