Market Mood & Moves
Sentiment-Driven Stock Prediction

Week 3

Mentors: Meet & Sarthak

January 2026

Welcome to Week 3

In Week 1, we built our data pipelines. In Week 2, we mastered the art of extracting sentiment using FinBERT.
Now, in Week 3, we tackle the dimension of **Time**.
Stock prices are not static values; they are sequences. A price of \$150 means something very different if yesterday was \$100 (uptrend) versus \$200 (crash). To model this, we must move beyond standard Neural Networks and enter the world of **Sequence Modeling**. We will study the Recurrent Neural Network (RNN) and its powerful successor, the Long Short-Term Memory (LSTM) network, directly from the theoretical foundations.
Key Themes:

- **Sequence Probability:** $P(x_t|x_{t-1}, \dots)$.
- **RNNs and LSTMs:** Preserving memory over time.
- **Multimodal Fusion:** Combining FinBERT sentiment with price history.
- **Strategy:** From raw probabilities to Buy/Sell signals.

Contents

1 Probabilistic Modeling of Time Series

Before writing code, we must understand what it means to model a time series mathematically.

1.1 The Sequence Problem

In standard machine learning (like image classification), we assume samples are Independent and Identically Distributed (I.I.D.). In financial markets, this assumption is false. The price at time t is heavily dependent on $t-1$.

We view stock prices as a sequence $x_1, \ldots, x_T$. Our goal is to estimate the joint probability of the entire sequence $P(x_1, \ldots, x_T)$. Using the chain rule of probability, we can factorize this:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}) \tag{1}$$

This equation implies that to predict the current state x_t, we ideally need the *entire* history of the universe up to that point.

1.2 Autoregressive Models

Since we cannot calculate a probability conditioned on infinite history, we use approximation strategies.

1. **Markov Assumption:** We assume the current state depends only on the last τ steps.

   $$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-\tau}, \ldots, x_{t-1})$$

   This effectively uses a fixed-length window (e.g., the last 30 days) as input to a standard regression model (MLP).

2. **Latent Variable Models (RNNs):** instead of looking only at a fixed window, we maintain a summary of the past, called the **Hidden State** (h_t).

   $$h_t = f(x_t, h_{t-1})$$

   $$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1})$$

Insight: Why "Hidden"?

The state h_t is called "hidden" not because it's invisible to the programmer, but because there is no ground truth label for it in the dataset. The network must *learn* what information to store in h_t (e.g., "market is volatile", "uptrend detected") to minimize the prediction error of the observed price x_t.

2 Recurrent Neural Networks (RNNs)

The Recurrent Neural Network is the direct implementation of the Latent Variable model described above.

2.1 The Architecture

In a standard Multilayer Perceptron (MLP) with a single hidden layer, the hidden state H is calculated as:

$$H = \phi(XW_{xh} + b_h)$$

where ϕ is an activation function, X is the input, and W_{xh} are the weights.

In an RNN, we introduce a new weight matrix W_{hh} that connects the previous hidden state to the current one. The calculation for time step t becomes:

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \tag{2}$$

- $X_t \in \mathbb{R}^{n \times d}$: Input at time t (batch size n, features d).
- $H_{t-1} \in \mathbb{R}^{n \times h}$: Hidden state from previous step.
- $W_{xh} \in \mathbb{R}^{d \times h}$: Weights for input.
- $W_{hh} \in \mathbb{R}^{h \times h}$: Weights for recurrent connection.

The output O_t is then computed solely from the hidden state:

$$O_t = H_t W_{hq} + b_q$$

2.2 Backpropagation Through Time

Training an RNN is difficult because the gradient of the loss at time T depends on the computations at time $T-1, T-2, \ldots, 1$.

Theoretical Detail: The Vanishing Gradient Problem

Computing gradients involves the chain rule through the sequence. This results in a product of T matrices (essentially W_{hh}^T).

- If the eigenvalues of W_{hh} are < 1, the gradients decay exponentially to zero (Vanishing Gradient). The model forgets early history.
- If the eigenvalues are > 1, the gradients grow exponentially (Exploding Gradient).

For financial data, where a news event 30 days ago might impact today's trend, the "memory" of a standard RNN is typically too short.

3 Modern RNNs: The LSTM

To solve the memory issue, Hochreiter and Schmidhuber (1997) introduced the **Long Short-Term Memory (LSTM)** network. It is designed to allow gradients to flow unchanged for long durations.

3.1 The Memory Cell vs. Hidden State

The key innovation of the LSTM is separating the internal memory from the output.

- $\mathbf{C}_t$: The **Cell State** (Long-term memory). Flows like a conveyor belt.
- $\mathbf{H}_t$: The **Hidden State** (Short-term output). Derived from the cell state.

3.2 The Gating Mechanism

LSTMs use "gates"—neural networks with Sigmoid activations (σ) that output values between 0 (closed) and 1 (open)—to control information flow.

3.2.1 Input Gate (I_t)

Decides how much of the *new* information should be stored in the cell state.

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i)$$

3.2.2 Forget Gate (F_t)

Decides how much of the *old* cell state to discard (crucial for changing market regimes).

$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f)$$

3.2.3 Output Gate (O_t)

Decides what part of the cell state to output as the hidden state.

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o)$$

3.2.4 Candidate Memory Cell ($\tilde{\mathbf{C}}_t$)

A temporary proposal for new memory (using Tanh activation).

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c)$$

3.3 The Flow Equations

The update logic combines these gates:

Step 1: Update Long-Term Memory

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

Interpretation: Forget some old history, add some new history.

Step 2: Generate Hidden State

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

Interpretation: The hidden state is a filtered version of the long-term memory.

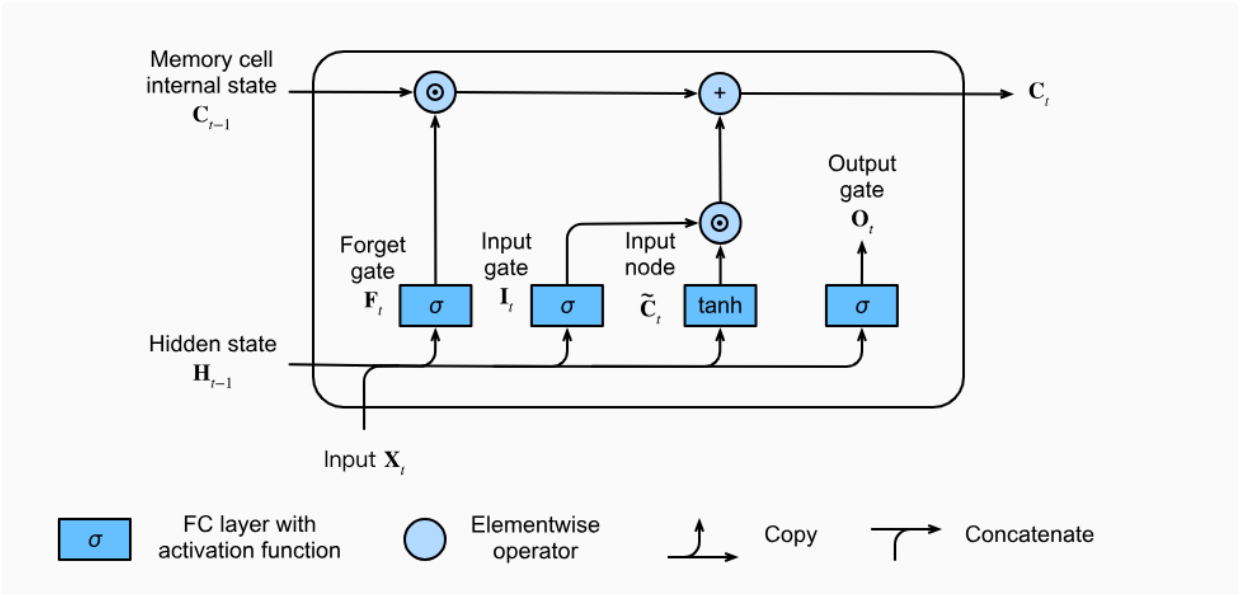

Figure 1: LSTM Architecture

4 Design of the Market Model

We will now design a multimodal time series model. It will consume two types of data:

1. **Numerical Data:** Historical Price and Volume (OHLCV).
2. **Semantic Data:** Sentiment Scores derived from FinBERT (from Week 2).

4.1 Feature Engineering

Before feeding data into the LSTM, we must construct our feature vector x_t.

1. Raw Features:

- Open, High, Low, Close, Volume.
- **FinBERT Sentiment Score (S_t):** A value $\in [-1, 1]$ representing the aggregate news sentiment for that day.

2. Technical Indicators (Derived): LSTMs are powerful, but helping them with engineered features improves convergence.

- **RSI (Relative Strength Index):** Momentum indicator.
- **MACD (Moving Average Convergence Divergence):** Trend-following momentum.
- **Rolling Volatility:** Standard deviation of returns over the last N days.

3. Stationarity Transformation: Neural networks struggle with raw prices (e.g., predicting 2000 when training data was around 100). We train on **Log Returns** or **Percentage Changes**.

$$R_t = \ln\left(\frac{\text{Close}_t}{\text{Close}_{t-1}}\right)$$

4.2 Multimodal Input Structure

Our input tensor for a single batch will have the shape (Batch_Size, Sequence_Length, Num_Features). Let's assume Sequence_Length = 60 (looking back 60 days) and Num_Features = 7 (Returns, Volatility, RSI, MACD, Volume, **Sentiment**).

Insight: Why mix Sentiment with Price?

Price reflects *what* happened. Sentiment reflects *why* it happened or what *might* happen. FinBERT gives the LSTM a "leading indicator". If sentiment drops negative today ($S_t \approx -0.9$) but price hasn't dropped yet, the LSTM can learn that a price drop typically follows such a divergence.

5 Implementation Guide

We will implement the time-series pipeline using pure PyTorch. The implementation is divided into:

1. **Dataset Class:** Handling the sliding window logic.
2. **Model Class:** The LSTM architecture.
3. **Training Pipeline:** Data loading and the training loop.
4. **Visualization:** Plotting predictions vs. actuals.

5.1 The Dataset Class

This class converts our time-series data into the (Input, Target) pairs required for supervised learning.

Time Series Dataset

```python
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np
import matplotlib.pyplot as plt

class MarketDataset(Dataset):
    def __init__(self, features, targets, sequence_length=60):
        """
        features: numpy array of shape (N, num_features)
                  Includes OHLCV + FinBERT Sentiment.
        targets: numpy array of shape (N,)
                 Target values (e.g., next day return).
        sequence_length: Lookback window size.
        """
        self.features = torch.tensor(features, dtype=torch.float32)
        self.targets = torch.tensor(targets, dtype=torch.float32)
        self.seq_len = sequence_length

    def __len__(self):
        # We cannot use the last seq_len items as inputs
        # because they don't have a future target
        return len(self.features) - self.seq_len

    def __getitem__(self, idx):
        # Input: Data from idx to idx + seq_len
        x = self.features[idx : idx + self.seq_len]

        # Target: Data at idx + seq_len (predicting the next step)
        y = self.targets[idx + self.seq_len]

        return x, y
```

5.2 The LSTM Model

This model follows the standard Many-to-One architecture.

Sentiment-Aware LSTM

```python
import torch.nn as nn

class SentimentLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim=1, dropout=0.2):
        super(SentimentLSTM, self).__init__()

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # LSTM Layer
        # batch_first=True expects input (Batch, Seq, Features)
        self.lstm = nn.LSTM(
            input_size=input_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout
        )

        # Fully Connected Output Layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state h0 and cell state c0
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)

        # Forward propagate LSTM
        # out shape: (batch_size, seq_length, hidden_dim)
        out, _ = self.lstm(x, (h0, c0))

        # We only care about the output of the LAST time step
        # out[:, -1, :] fetches the hidden state of the last day in the window
        out = out[:, -1, :]

        # Final prediction
        prediction = self.fc(out)
        return prediction
```

5.3 Training Visualization Pipeline

Here we instantiate the data, the loader, run the training loop, and finally visualize the results on a test set.

Full Training Evaluation Loop

```
import torch.optim as optim

# Hyperparameters
INPUT_DIM = 7       # 6 Technical features and 1 Sentiment score
HIDDEN_DIM = 64
NUM_LAYERS = 2
LR = 0.001
EPOCHS = 50
BATCH_SIZE = 32
SEQ_LEN = 60

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Dummy Data Generation (Replace with your Real Data)
# N=1200 days total (1000 train, 200 test)
total_days = 1200
raw_features = np.random.randn(total_days, INPUT_DIM)
# Create a target that is somewhat predictable for visualization (e.g., sine wave pattern)
time_steps = np.linspace(0, 100, total_days)
raw_targets = np.sin(time_steps) + 0.1 * np.random.randn(total_days)

# Split into Train and Test
train_size = 1000
train_features, test_features = raw_features[:train_size], raw_features[train_size:]
train_targets, test_targets = raw_targets[:train_size], raw_targets[train_size:]

# Data Preparation
train_dataset = MarketDataset(train_features, train_targets, sequence_length=SEQ_LEN)
test_dataset = MarketDataset(test_features, test_targets, sequence_length=SEQ_LEN)

# IMPORTANT: Shuffle=True for Training, False for Testing
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False, drop_last=False)

# Model Setup
model = SentimentLSTM(INPUT_DIM, HIDDEN_DIM, NUM_LAYERS).to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=LR)

# Training Loop
print("Starting Training...")
model.train()

for epoch in range(EPOCHS):
    train_loss = 0
```

```
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

        optimizer.zero_grad()
        output = model(X_batch)

        # Squeeze output to match target shape
        loss = criterion(output.squeeze(-1), y_batch)

        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        train_loss += loss.item()

    if (epoch+1) % 10 == 0:
        print(f'Epoch {epoch+1}/{EPOCHS}: Avg Loss {train_loss/len(
    train_loader):.6f}')

# Evaluation and Visualization
print("Starting Evaluation...")
model.eval()
predictions = []
actuals = []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        X_batch = X_batch.to(device)
        output = model(X_batch)
        predictions.append(output.item())
        actuals.append(y_batch.item())

# Plotting the results
plt.figure(figsize=(12, 6))
plt.plot(actuals, label='Actual Prices', color='blue', alpha=0.6)
plt.plot(predictions, label='LSTM Predictions', color='red', linestyle='
    --')
plt.title('LSTM Model Performance: Actual vs Predicted')
plt.xlabel('Time (Days)')
plt.ylabel('Normalized Value')
plt.legend()
plt.grid(True)
plt.show()
```

Insight: Interpreting the Plot

When you run the visualization code, you should observe:

- **Lag:** LSTMs often simply predict $P_t \approx P_{t-1}$. If the red line (predictions) looks exactly like the blue line (actuals) but shifted one step to the right, the model hasn't learned the underlying pattern, just the persistence.
- **Noise Smoothing:** A good model will capture the trend (sine wave in our dummy data) while ignoring the high-frequency random noise.

Theoretical Detail: Gradient Clipping

As mentioned in the theoretical section, LSTMs can still suffer from exploding gradients. We implement clip_grad_norm_ to cap the gradient vector norm at 1.0, ensuring stability.

6 Trading Strategy Integration

Having a model that outputs a number is not a trading strategy. We need a decision logic.

6.1 Strategy: Sentiment-Weighted Momentum

We combine the "black box" prediction of the LSTM with the interpretable signal from FinBERT.

Inputs:

- P_{LSTM}: Predicted return for tomorrow (e.g., +0.02%).
- S_{FinBERT}: Average Sentiment Score of today's news (e.g., +0.85).

Logic: We only trade when both the technical prediction (LSTM) and the fundamental sentiment (FinBERT) **agree**. This acts as a confirmation filter.

Buy/Sell Algorithm

```
def get_trade_signal(lstm_prediction, sentiment_score, threshold=0.001):
    """
    Returns: 'BUY', 'SELL', or 'HOLD'
    """

    # CASE 1: Strong Bullish Signal
    # LSTM predicts price UP AND Sentiment is Positive
    if lstm_prediction > threshold and sentiment_score > 0.2:
        return 'BUY'

    # CASE 2: Strong Bearish Signal
    # LSTM predicts price DOWN AND Sentiment is Negative
    elif lstm_prediction < -threshold and sentiment_score < -0.2:
        return 'SELL'

    # CASE 3: Divergence or Weak Signal
    # If LSTM says UP but Sentiment is Negative, we stay out (Risk
    Management)
    else:
        return 'HOLD'
```

6.2 Execution Workflow

- 1. **Pre-Market:** Scrape news from last 24h.
- 2. **Sentiment Analysis:** Run FinBERT to get S_t.
- 3. **Data Update:** Fetch yesterday's OHLCV. Update technical indicators.
- 4. **Forecasting:** Feed sequence window + S_t into LSTM. Get P_{LSTM}.
- 5. **Decision:** Run get_trade_signal.
- 6. **Order:** If 'BUY', execute market order at Open. Set Stop Loss at -2%.

7 Week 3 Completion Checklist

Essential Tasks

Theoretical Mastery:

- ✓ Explain the difference between Markov Models and Recurrent Neural Networks.
- ✓ Draw the internal structure of an LSTM cell (Input, Forget, Output gates).
- ✓ Explain why the Vanishing Gradient problem occurs in standard RNNs.

Implementation:

- ✓ Create a PyTorch 'Dataset' class that implements the Sliding Window technique.
- ✓ Implement the 'SentimentLSTM' class with Dropout.
- ✓ Train the model on at least 1 year of historical data + sentiment features.

Project Integration:

- ✓ Write a script that loads your saved FinBERT model (from Week 2) and your new LSTM model.
- ✓ Generate a mock "Buy/Sell" report for a list of 5 tech stocks (AAPL, MSFT, GOOGL, NVDA, TSLA) based on today's data.

Pro Challenges

- **Attention Mechanism:** Implement an Attention layer on top of the LSTM to let the model "focus" on specific days in the 60-day window (e.g., Earnings calls).