A Technical Exposition of an Ensemble Deep Learning Framework for Solana (SOL-USD) Hourly Price Forecasting

Abstract

Forecasting the price movements of highly volatile assets like cryptocurrencies presents a significant challenge for quantitative finance and machine learning practitioners. This paper provides a detailed technical exposition of a sophisticated deep learning framework, implemented in Python, designed specifically for predicting the hourly price of Solana (SOL-USD). The core of the framework consists of an ensemble of hybrid Convolutional Neural Network-Long Short-Term Memory (CNN-LSTM) models, termed SolanaPredictor. This architecture integrates 1D CNN layers for local feature extraction, parallel LSTM pathways for capturing distinct temporal dynamics related to price value and direction, and incorporates both learnable time-weighted attention and fixed sinusoidal positional encoding to enhance temporal focus. A key component is the custom DirectionalLoss function, which strategically balances the prediction of price magnitude (using robust Huber loss) and price direction, incorporating class weighting to address potential imbalances in market movements. The framework employs a comprehensive data preprocessing pipeline, including outlier mitigation, log transformation, extensive feature engineering encompassing standard technical indicators and custom directional signals, stationarity assessment via the Augmented Dickey-Fuller test with conditional differencing, and robust scaling techniques. Model training utilizes performance-driven learning rate scheduling and early stopping based on validation directional accuracy. Finally, an ensemble strategy averages predictions from multiple model instances trained with different initializations to enhance stability and generalization. This paper meticulously dissects each component of the implementation, elucidating the design choices and underlying rationale, thereby offering a comprehensive case study on applying advanced deep learning methodologies to the complex task of cryptocurrency price forecasting.

1. Introduction

1.1. Context

The proliferation of digital assets has introduced a new frontier in financial markets, characterized by unprecedented volatility, rapid innovation, and distinct market dynamics compared to traditional equities or commodities. Cryptocurrencies such as Solana (SOL-USD) often exhibit sharp price fluctuations, non-stationary behavior, and complex dependencies influenced by factors ranging from technological developments to market sentiment and macroeconomic trends. Accurately forecasting the price movements of these assets is consequently a highly challenging research problem, yet one with significant practical implications.

1.2. Problem Statement

The primary objective addressed by the framework under analysis is the prediction of the next hour's price movement for Solana (SOL-USD), based on historical hourly market data. This involves forecasting not only the magnitude of the potential price change but also, crucially, its direction (upward or

downward). Reliable short-term forecasts can inform algorithmic trading strategies, assist in dynamic risk management, provide insights for market analysis, and potentially contribute to portfolio optimization within the volatile cryptocurrency space. The inherent noise, non-stationarity, and potential for abrupt shifts in cryptocurrency time series necessitate sophisticated modeling approaches that can capture complex underlying patterns.

1.3. Proposed Solution Overview

This paper examines a specific Python implementation designed to tackle the SOL-USD hourly prediction task using an advanced deep learning framework. The cornerstone of this framework is the SolanaPredictor model, a hybrid architecture combining Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. This hybridization aims to leverage the strengths of CNNs in extracting salient local features across the input variables at each time step and the power of LSTMs in modeling long-range temporal dependencies within the sequence. The architecture further incorporates attention mechanisms through learnable time weights and fixed positional encodings to selectively focus on relevant parts of the input sequence, particularly recent data. A novel aspect is the custom DirectionalLoss function, explicitly designed to optimize for both the accuracy of the predicted price value (using a robust Huber loss) and the correctness of the predicted direction, incorporating class weights to handle data imbalances. To enhance robustness and mitigate overfitting, the framework employs an ensemble strategy, averaging the outputs of multiple SolanaPredictor instances trained independently.

1.4. Contribution

The contribution of this paper is a rigorous, step-by-step technical dissection and analysis of this specific deep learning implementation for SOL-USD forecasting. It delves into the intricacies of the code, explaining the rationale behind each design choice, from the initial data acquisition and multi-stage preprocessing pipeline to the detailed architecture of the SolanaPredictor model, the formulation of the custom loss function, the training methodology including optimization and regularization techniques, and the final prediction generation process. By providing this in-depth exposition, the paper serves as a detailed case study illustrating the application of contemporary deep learning techniques to the challenging domain of cryptocurrency price prediction, highlighting practical considerations and sophisticated solutions implemented within the code.

1.5. Structure

The remainder of this paper is organized as follows: Section 2 details the Data Acquisition and Preparation Framework, covering data retrieval, cleaning, transformation, feature engineering, stationarity handling, scaling, and sequentialization. Section 3 describes The SolanaPredictor Modeling Approach, dissecting the hybrid CNN-LSTM architecture, the custom DirectionalLoss function, and the ensemble strategy. Section 4 outlines the Model Training Protocol, including optimization, learning rate scheduling, and overfitting mitigation. Section 5 explains the process of Forecasting Future Price Movements using the trained ensemble. Section 6 provides a Discussion, synthesizing the strengths and limitations of the framework and suggesting directions for future work. Finally, Section 7 concludes the paper with a summary of the key findings and the significance of the analyzed approach.

2. Data Acquisition and Preparation Framework

The foundation of any data-driven forecasting model lies in the careful acquisition, cleaning, transformation, and structuring of input data. This section details the systematic pipeline implemented in the provided code to prepare historical SOL-USD market data for consumption

by the deep learning model.

2.1. Data Retrieval and Initial Cleansing

- **Data Source:** The process commences by retrieving historical market data using the yfinance library. Specifically, it fetches data for the ticker 'SOL-USD' over a defined period ("730d", representing approximately two years) at a specified interval ('1h'). This yields a time series dataset containing the standard Open, High, Low, Close prices, and Volume (OHLCV) for each hour within the requested period.
- **Outlier Handling (clean_extreme_values function):** Recognizing that raw financial data can contain erroneous entries or genuine but extreme price spikes that might destabilize model training, a dedicated function clean_extreme_values is applied early in the pipeline. This function targets specified columns ('Volume', 'Open', 'High', 'Low', 'Close'). For each column, it calculates the first quartile (Q1) and third quartile (Q3), and subsequently the Interquartile Range (IQR = Q3 - Q1). Values falling below Q1 - threshold * IQR or above Q3 + threshold * IQR are considered outliers. The implementation uses a threshold of 5 (adjusted from a code comment suggesting 10), indicating a reasonably strict criterion for outlier detection. These identified outliers are then clipped, meaning they are replaced with the calculated lower or upper bound, respectively. This proactive outlier mitigation is performed *before* any further transformations or feature calculations. This sequence is deliberate: extreme values can significantly distort statistical measures (like mean and standard deviation) used in subsequent steps such as feature scaling or technical indicator calculation. Furthermore, transformations like the logarithm are undefined for non-positive values, which could arise from data errors; early clipping helps prevent such issues. Applying this cleaning step at the outset ensures that derived features are computed on a more numerically stable foundation, preventing the propagation of potentially disruptive outliers through the pipeline and reflecting an understanding of the noisy characteristics often present in financial market data.

2.2. Data Transformation and Feature Engineering

Following initial cleaning, the data undergoes significant transformation and enrichment through feature engineering to provide a more informative representation for the model.

- **Log Transformation:** The 'Close' price is transformed using the natural logarithm (np.log(df['Close'])). This is a standard practice in financial time series analysis for several reasons: it helps stabilize the variance of the series, making it more homoscedastic; it can linearize exponential growth trends often observed in asset prices; and it tends to make the distribution of price returns closer to a normal distribution, which can be beneficial for statistical modeling and neural network training. The original log-transformed closing price is explicitly preserved in a separate column (Original_Log_Close). This preservation is crucial for correctly reconstructing the price prediction later, especially if differencing is applied (as discussed in Section 2.3). The primary 'Close' column is then updated to contain

these log prices, serving as the basis for subsequent indicator calculations and potentially as the direct target variable if differencing is not required.

- **Technical Indicators:** A suite of standard technical indicators is calculated based on the log-transformed prices and volume. These indicators aim to capture various aspects of market dynamics:
    - *Trend Following:* Simple Moving Average (SMA_10) and Exponential Moving Average (EMA_10) provide smoothed versions of the price trend over a 10-hour window. Moving Average Convergence Divergence (MACD), calculated as the difference between 12-period and 26-period EMAs, along with its 9-period EMA signal line (MACD_signal), helps identify trend direction and momentum shifts. The Awesome Oscillator (AO), computed as the difference between 5-period and 34-period SMAs of the midpoint price, also signals market momentum.
    - *Momentum:* The Relative Strength Index (RSI_14) measures the magnitude of recent price changes to evaluate overbought or oversold conditions over a 14-hour lookback.
    - *Volatility/Trend Strength:* The Average Directional Index (ADX), derived from the True Range (TR) and Directional Movement Index (DX) over a 14-hour window, quantifies the strength of a prevailing trend (regardless of direction).
    - *Volume-Based:* On-Balance Volume (OBV) accumulates volume on up-days and subtracts it on down-days, relating price and volume. The Volume Rate of Change (VROC) measures the rate at which volume is changing over a 14-hour period.
    - After calculating these indicators, the clean_extreme_values function is called again, specifically targeting potentially volatile indicators like OBV, VROC, DX, and ADX with a higher threshold (10), further ensuring numerical stability.
- **Custom Directional/Volatility Features (add_directional_features function):** Beyond standard indicators, the framework incorporates custom features designed to capture signals particularly relevant for short-term directional prediction:
    - *Price Momentum (Price_Momentum_5, Price_Momentum_10):* Direct calculation of the price change over the last 5 and 10 hours (based on log prices).
    - *Directional Change (Dir_Change_5, Dir_Change_10):* Explicitly encodes the sign (+1, -1, or 0) of the price change over the last 5 and 10 hours.
    - *Volatility (Volatility_5, Volatility_10):* Rolling standard deviation of log prices over 5 and 10 hours, measuring recent price fluctuation.
    - *RSI Signals (RSI_Oversold, RSI_Overbought):* Binary flags triggered when RSI_14 crosses below 30 or above 70, respectively, potentially indicating reversal points.
    - *Volume Surge (Volume_Surge):* A binary flag indicating if the current volume is significantly higher (1.5 times) than the 10-hour rolling average volume.
    - *Breakout Signals (Breakout_Up, Breakout_Down):* Binary flags indicating if the closing price crosses above the upper or below the lower Bollinger Band (calculated within the function using a 20-period SMA and 2 standard deviations), signaling potential volatility increases or trend continuations.
    - *Crypto-Specific (VWAP, Close_VWAP_Ratio):* Calculation of the Volume-Weighted

Average Price (VWAP) and the ratio of the closing price to the VWAP, metrics often monitored in cryptocurrency trading.

This combination of standard technical indicators and custom-engineered features provides the model with a rich, multi-faceted representation of the market state. Standard indicators offer established perspectives on trend, momentum, and volume, while the custom features directly target recent price dynamics, volatility shifts, and potential event triggers (reversals, breakouts, volume spikes) that could be predictive of near-term (hourly) movements. The inclusion of VWAP acknowledges specific metrics commonly analyzed within the cryptocurrency domain.

- **Handling NaNs:** Technical indicators calculated using rolling windows inevitably produce Not-a-Number (NaN) values at the beginning of the series. The line df.dropna(subset=features).reset_index(drop=True) removes any rows containing NaNs in the columns designated as features, ensuring that only complete data records are fed into the model.
- **Feature Selection:** The code defines all_features to encompass potentially all calculated numerical columns (excluding date/time) intended for scaling, and features which seems initially intended as the final model input set. However, the implementation ultimately uses all_features when creating the datasets and configuring the model's input size (input_size=len(all_features)). This indicates that all generated, cleaned, and non-NaN features are utilized as inputs to the SolanaPredictor model.

Table 1: Engineered Features

Feature Name	Calculation/Brief Description	Category
SMA_10	10-hour Simple Moving Average of log 'Close'	Trend
EMA_10	10-hour Exponential Moving Average of log 'Close'	Trend
RSI_14	14-hour Relative Strength Index based on log 'Close'	Momentum
MACD	(12-hour EMA - 26-hour EMA) of log 'Close'	Trend/Momentum
MACD_signal	9-hour EMA of MACD	Trend/Momentum

ADX	14-hour Average Directional Index (derived from TR, DX)	Trend Strength
AO	(5-hour SMA - 34-hour SMA) of midpoint price	Momentum
OBV	On-Balance Volume (cumulative volume based on price direction)	Volume
VROC	14-hour Volume Rate of Change	Volume
Price_Momentum_5	Log 'Close' difference over 5 hours	Custom Signal
Price_Momentum_10	Log 'Close' difference over 10 hours	Custom Signal
Dir_Change_5	Sign of log 'Close' difference over 5 hours	Custom Signal
Dir_Change_10	Sign of log 'Close' difference over 10 hours	Custom Signal
Volatility_5	5-hour rolling standard deviation of log 'Close'	Volatility
Volatility_10	10-hour rolling standard deviation of log 'Close'	Volatility
RSI_Oversold	Binary flag (1 if RSI_14 < 30, else 0)	Custom Signal
RSI_Overbought	Binary flag (1 if RSI_14 > 70, else 0)	Custom Signal
Volume_Surge	Binary flag (1 if Volume > 1.5 *	Custom Signal

	10-hour rolling mean Volume)	
Breakout_Up	Binary flag (1 if log 'Close' > previous Upper Bollinger Band)	Custom Signal
Breakout_Down	Binary flag (1 if log 'Close' < previous Lower Bollinger Band)	Custom Signal
VWAP	Volume Weighted Average Price (cumulative)	Crypto-Specific
Close_VWAP_Ratio	Ratio of log 'Close' to log VWAP	Crypto-Specific

2.3. Stationarity Assessment and Treatment

- **Concept of Stationarity:** Many time series models, including LSTMs, perform better or rely on the assumption that the statistical properties (mean, variance, autocorrelation) of the series are constant over time, a property known as stationarity. Non-stationary series, common in financial markets, can lead to spurious correlations and models that fail to generalize outside the training period.
- **ADF Test:** To formally assess stationarity, the framework employs the Augmented Dickey-Fuller (ADF) test (adfuller from the statsmodels.tsa.stattools library). This test is applied to the log-transformed 'Close' price series. The null hypothesis of the ADF test is that the time series possesses a unit root, indicating non-stationarity.
- **Conditional Differencing:** The decision to difference the series is made adaptively based on the ADF test result. If the p-value returned by adfuller is greater than a chosen significance level (0.05), the null hypothesis (non-stationarity) is not rejected. In this scenario, the 'Close' series (containing log prices) is differenced by calculating the change between consecutive time steps (df['Close'].diff()). This transformation often helps to induce stationarity by removing trends or unit roots. The dropna() method is subsequently called to remove the initial NaN value created by the differencing operation. A boolean flag, differenced, is set to True to record that this transformation has occurred. This flag is essential for ensuring the correct inverse transformation is applied during the prediction phase (see Section 5.2). This conditional application of differencing represents a statistically informed preprocessing step. Rather than universally applying differencing, which could remove valuable information if the series were already stationary, the framework adapts based on the observed properties of the specific data segment being analyzed. This aims to satisfy model assumptions regarding stationarity only when statistically warranted.

2.4. Data Scaling and Sequentialization

The final preprocessing steps involve scaling the features and the target variable, splitting the data, and structuring it into sequences suitable for the deep learning model.

- **Scaling Rationale:** Neural networks typically require numerical features to be scaled. Scaling prevents features with inherently larger values (e.g., volume) from disproportionately influencing the model's weights and loss function compared to features with smaller values. It also aids the convergence of gradient-based optimization algorithms and makes the model less sensitive to the absolute scale of inputs.
- **Scalers:** Two different scaling techniques are employed:
    - RobustScaler for all_features: This scaler centers the data using the median and scales it according to the Interquartile Range (IQR). The formula is X_scaled = (X - median) / IQR. RobustScaler is chosen because it is less sensitive to outliers compared to StandardScaler (which uses mean and standard deviation) or MinMaxScaler. Given the potential for extreme values in financial data, even after the initial clipping step, this choice enhances the robustness of the scaling process. Crucially, the RobustScaler is fitted *only* on the training portion of the data (df.iloc[:train_end][all_features]).
    - MinMaxScaler for the target ('Close'): This scaler transforms the target variable (which is either log prices or differenced log prices) into the range using the formula X_scaled = (X - min) / (max - min). This is a common choice for target variables in regression tasks with neural networks, particularly if output activation functions might implicitly operate within this range. Similar to the feature scaler, the MinMaxScaler is fitted *only* on the target values corresponding to the training set (df.iloc[['Close']]). Fitting both scalers exclusively on the training data is paramount to prevent data leakage. Information from the validation and test sets (representing future data relative to training) must not influence the scaling parameters (median, IQR, min, max) used during training, as this would lead to an overly optimistic evaluation of the model's generalization performance.
- **Data Splitting:** The dataset is split chronologically into training, validation, and test sets. The split points are determined by train_ratio (0.7) and val_ratio (0.15). The training set comprises the first 70% of the available sequences, the validation set the next 15%, and the test set the remaining ~15%. This chronological split is essential for time series forecasting, as it mimics the real-world scenario where a model is trained on past data and evaluated on its ability to predict subsequent, unseen data. The lookback window size T (24 hours) determines the starting point for constructing sequences.
- **Sequentialization (TimeSeriesDataset, DataLoader):** The TimeSeriesDataset class is responsible for transforming the scaled DataFrame into input-output pairs suitable for sequence modeling. For a given index idx, its __getitem__ method extracts a sequence of length T (from time t-T to t-1, where t = start_idx + idx) containing all scaled features (all_features) as the input tensor (seq). It also retrieves the corresponding scaled target value

('Close') at time t as the output tensor (target). This implements a sliding window approach over the time series. The DataLoader utility then takes these datasets and creates iterable batches of sequences and targets. It shuffles the training data (shuffle=True) in each epoch to introduce stochasticity and potentially improve generalization, while keeping the validation and test data unshuffled (shuffle=False) for consistent evaluation. This careful structuring ensures the model receives data in the required temporal sequence format while adhering to sound validation practices by preventing leakage and maintaining chronological order for evaluation.

3. The SolanaPredictor Modeling Approach

This section details the architecture of the core predictive model (SolanaPredictor), the custom loss function designed to guide its optimization (DirectionalLoss), and the ensemble strategy used to combine multiple model instances.

3.1. Hybrid CNN-LSTM Architecture Design (SolanaPredictor class)

- **Overall Rationale:** The SolanaPredictor employs a hybrid architecture combining Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. This design choice is motivated by the complementary strengths of these components for time series analysis. 1D CNNs are effective at automatically extracting hierarchical, local patterns across the input features at different time steps (akin to spatial feature extraction in the feature dimension). LSTMs, a type of Recurrent Neural Network (RNN), are specifically designed to capture temporal dependencies and long-range patterns within sequential data, mitigating the vanishing gradient problem common in simple RNNs. By integrating them, the model aims to learn complex spatio-temporal representations from the multivariate financial time series.
- **CNN Layers:** The initial processing block consists of a sequence (nn.Sequential) of 1D convolutional layers (nn.Conv1d).
    - The first nn.Conv1d takes the input sequence with in_channels equal to the number of features (len(all_features)) and applies cnn_channels (64) filters. The kernel_size=3 means each filter looks at a local window of 3 consecutive time steps across all features. padding=1 ensures the output sequence length remains the same as the input.
    - A LeakyReLU(0.2) activation function follows. Unlike standard ReLU, LeakyReLU allows a small, non-zero gradient for negative inputs, which can help prevent "dying ReLU" issues where neurons become inactive during training.
    - BatchNorm1d(cnn_channels) is applied for normalization across the feature channels. Batch normalization helps stabilize training, allows for potentially higher learning rates, and acts as a form of regularization.
    - A second identical block (Conv1D, LeakyReLU, BatchNorm1D) follows, allowing the network to learn more complex feature representations.
    - Finally, Dropout(0.2) is applied. Dropout is a regularization technique that randomly

sets a fraction (20%) of the activations to zero during training, preventing the model from becoming overly reliant on specific neurons and improving generalization. The output of the CNN block is then permuted to be suitable for the LSTM layers (batch_first=True format).

- **Time-Weighted Attention & Positional Encoding:** Before and after the CNN block, mechanisms are introduced to enhance the model's awareness of temporal positioning and importance:
  - *Learnable Time Weights:* A nn.Parameter named time_weights is initialized as a tensor of ones with size T (24), divided by T for initial uniform weighting. These weights are applied to the input sequence *before* the CNN layers (weighted_input = x * time_weights.view(1, -1, 1)), after being passed through a softmax function. This effectively implements a learnable attention mechanism where the model can assign different importance scores to each of the 24 time steps in the input window, potentially learning to focus more on recent data points if predictive.
  - *Sinusoidal Positional Encoding:* After the CNN layers, fixed sinusoidal positional encodings (pos_enc) are added to the CNN output features. This technique, popularized by the Transformer architecture, injects explicit information about the absolute or relative position of each time step within the sequence. It uses sine and cosine functions of varying frequencies applied to the position index. Since the CNN operations might obscure the original temporal order, adding positional encoding ensures this information is available to the subsequent LSTM layers. The implementation carefully handles potential dimension mismatches between the fixed encoding dimension (8) and the number of CNN output channels (64) by adding the encoding only to the first use_dims = min(8, 64) channels (pos_enc_trimmed).
- **Parallel LSTM Pathways:** A key architectural feature is the use of two separate LSTM layers (lstm_value and lstm_dir). Both LSTMs receive the *same* input: the output of the CNN block enhanced with positional encoding. Each LSTM has lstm_hidden_size (128) units and operates with batch_first=True. The rationale for parallel pathways is to allow the model to specialize in learning the temporal patterns most relevant for predicting two distinct, albeit related, aspects of the target: its magnitude (value) and its direction. This constitutes a form of multi-task learning, where shared initial representations (from the CNN) are processed by specialized recurrent layers.
- **Output Heads:** The final hidden state from the last time step of each LSTM (value_out[:, -1, :] and dir_out[:, -1, :]) is fed into a separate fully connected layer (nn.Linear).
  - fc_value maps the lstm_value output to a single scalar, representing the predicted scaled value (log price or differenced log price).
  - fc_dir maps the lstm_dir output to a single scalar, representing a logit associated with the predicted direction.
- **Prediction Combination:** The outputs from the two heads are combined using a specific heuristic formula:
  1. The direction logit is passed through a tanh activation function (dir_sign =

torch.tanh(dir_logit)). Tanh squashes the output to the range [-1, 1], providing a "soft" sign indicating the predicted direction and confidence (values closer to +/- 1 indicate stronger confidence).

2. The final output is calculated as value_adjusted = value_pred * (0.8 + 0.2 * dir_sign). This formula uses the predicted direction sign to modulate the predicted value. If the predicted direction is strongly positive (dir_sign ≈ 1), the value prediction is scaled by 0.8 + 0.2*1 = 1.0. If strongly negative (dir_sign ≈ -1), it's scaled by 0.8 + 0.2*(-1) = 0.6. If the direction prediction is uncertain (dir_sign ≈ 0), it's scaled by approximately 0.8. This suggests an attempt to temper the magnitude prediction based on the directional signal, potentially reducing the magnitude if the direction is predicted to be negative. The constants 0.8 and 0.2 define the base scaling and the modulation strength.

This architecture represents a sophisticated fusion of techniques. The parallel LSTM pathways facilitate multi-task learning for value and direction. The placement of learnable attention *before* the CNN focuses feature extraction, while adding positional encoding *after* the CNN ensures temporal context reaches the LSTMs. The final combination step introduces a non-linear interaction between the value and direction predictions, although the specific formula appears heuristic rather than derived from first principles.

- Figure 1: SolanaPredictor Model Architecture (Textual Description)
  A diagram illustrating the architecture would show the following flow: An input sequence (Batch Size x T x Num Features) first passes through the Time Weights layer (multiplying each time step by a learned weight). The weighted sequence enters the CNN Block, consisting of two sets of followed by Dropout. Positional Encoding is then added element-wise to the output features of the CNN block. This enhanced feature sequence is fed identically into two parallel LSTM layers: lstm_value and lstm_dir. The final hidden state from lstm_value goes to fc_value to produce value_pred. The final hidden state from lstm_dir goes to fc_dir to produce dir_logit. dir_logit is passed through tanh to get dir_sign. Finally, value_pred and dir_sign are combined using the formula value_pred * (0.8 + 0.2 * dir_sign) to yield the final model output.

3.2. Direction-Focused Optimization: The DirectionalLoss Function

- **Motivation:** Standard regression loss functions like Mean Squared Error (MSE) primarily focus on minimizing the difference between predicted and actual magnitudes. In financial forecasting, however, correctly predicting the *direction* of price movement (up or down) is often paramount for profitability or risk management, even if the exact magnitude is slightly off. Conversely, a model might achieve low MSE by predicting small values close to zero but consistently get the direction wrong. The DirectionalLoss function is custom-designed to address this limitation by explicitly incorporating directional accuracy into the optimization objective.
- **Hybrid Loss Formulation:** The DirectionalLoss class implements a composite loss function, calculated as a weighted sum of a value-based loss and a direction-based loss. The weights are configurable parameters: value_weight (set to 0.6) and direction_weight (set to

0.4).

- *Value Loss Component:* This component uses nn.HuberLoss(). Huber loss is chosen for its robustness. It behaves like MSE for small errors (providing smooth gradients near the minimum) but transitions to behaving like Mean Absolute Error (MAE) for larger errors. This makes it less sensitive to outliers in the target variable than pure MSE, aligning well with the use of RobustScaler during preprocessing and the often-spiky nature of financial returns.
- *Directional Loss Component:* This component directly penalizes incorrect directional predictions. It first determines the predicted direction (pred_direction = torch.sign(pred)) and the true direction (target_direction = torch.sign(target)). To avoid issues with zero or near-zero actual movements, a valid_mask = (torch.abs(target) > 1e-6) is created to exclude samples where the target change is negligible. The core directional loss is then calculated as the mean of an indicator function (pred_direction!= target_direction) over the valid samples. This represents the proportion of times the model predicted the wrong direction.

- **Class Weighting:** The directional loss component further incorporates optional class_weights. These weights are intended to address potential class imbalance in the directional movements (e.g., markets might trend upwards more often than downwards in certain periods). The analyze_target_distribution function calculates these weights based on the inverse frequency of up (+1), down (-1), and potentially flat (0) movements observed in the training target data. If class_weights are provided to DirectionalLoss, the directional loss term is modified: (((pred_direction!= target_direction) & valid_mask).float() * weights).mean(). Here, weights is a tensor where each element's weight corresponds to the true direction (target_direction) of that sample. This mechanism assigns a higher penalty to misclassifying the direction of less frequent movements, preventing the model from simply optimizing for the majority direction class and encouraging it to learn signals for both upward and downward movements.
- Mathematical Formulation: The complete loss function can be expressed as:
  Loss = value_weight * Huber(pred, target) + direction_weight * Mean(I(pred_dir!= target_dir) * W)
  where Huber is the Huber loss function, I is the indicator function (1 if the condition is true, 0 otherwise), pred_dir and target_dir are the predicted and target signs, W represents the class weight applied based on the target_dir (or 1 if no class weights are used), and the Mean is computed only over samples where target is significantly different from zero (as per valid_mask).
  This custom loss function signifies a crucial design decision to tailor the model's learning objective directly to the dual goals of financial forecasting: achieving reasonable magnitude prediction (robustly handled by Huber loss) while strongly emphasizing directional correctness (via the weighted, class-balanced directional penalty). The specific weighting (60% value, 40% direction) reflects a chosen trade-off, prioritizing value prediction slightly more but allocating substantial importance to getting the direction right, moving beyond

simple statistical error minimization towards optimizing for potentially more actionable forecast characteristics.

3.3. Enhancing Stability via Ensembling: The EnsemblePredictor

- **Rationale:** Deep learning models, particularly complex ones trained on noisy data, can exhibit sensitivity to factors like random weight initialization and the stochastic nature of training algorithms (e.g., data shuffling, dropout). A single trained model might perform well but could represent only one local minimum in the loss landscape or might have overfit slightly differently depending on these random factors. Ensemble methods are a standard technique to address this. By training multiple instances of the same model architecture and combining their predictions, ensembles can often achieve:
  - *Reduced Variance:* Averaging predictions tends to cancel out some of the random errors or noise captured by individual models.
  - *Improved Generalization:* The combined prediction is typically less sensitive to the specific peculiarities of the training data than any single model instance.
  - *Increased Robustness:* The final forecast is less likely to be severely affected if one individual model performs poorly on a particular input.
- **Implementation (EnsemblePredictor class):** The framework implements a straightforward ensemble approach using the EnsemblePredictor class.
  - During training, the main script trains multiple (num_models = 3) instances of the SolanaPredictor model. Each instance is trained independently but uses the same architecture and hyperparameters, differing only in the random seed (torch.manual_seed(42 + i)) set before initialization and training. This ensures diversity stems primarily from different initial weights and potentially different mini-batch sequences due to shuffling.
  - The EnsemblePredictor class stores these trained models in a list (self.models).
  - Its predict method takes an input sequence x. It iterates through each stored model, sets it to evaluation mode (model.eval()) to disable dropout and batch normalization updates, and computes its prediction within a torch.no_grad() context to save memory and computation by disabling gradient tracking.
  - The predictions from all individual models are collected. The implementation then calculates a simple, equally weighted average of these predictions (ensemble_pred += pred * self.weights[i], where weights defaults to [1/len(models)] * len(models)). This averaged value is returned as the final ensemble prediction.

In the context of volatile and potentially non-stationary financial markets like cryptocurrencies, using an ensemble of models trained with different random seeds provides a practical strategy for enhancing prediction stability. While the underlying market dynamics might shift, the averaged prediction from slightly different model perspectives (learned due to different initializations and training paths) is expected to be more reliable and less prone to extreme errors than relying on a single model instance. It's a computationally feasible way to introduce diversity and improve the robustness of the forecasting system.

4. Model Training Protocol

This section details the procedures used to train the individual SolanaPredictor models that form the ensemble, focusing on the optimization algorithm, learning rate adjustments, and strategies for preventing overfitting and selecting the best model state.

4.1. Optimization and Learning Rate Adjustment

- **Optimizer:** The Adam optimizer (torch.optim.Adam) is employed for training the models. Adam is an adaptive learning rate optimization algorithm that computes individual learning rates for different parameters based on estimates of first-order (mean) and second-order (uncentered variance) moments of the gradients. It is widely used and often effective for training deep neural networks. The implementation specifies an initial learning_rate of 0.0005 and a weight_decay of 1e-4. Weight decay implements L2 regularization by adding a penalty proportional to the squared magnitude of the model weights to the loss function, discouraging overly complex models and helping to prevent overfitting.
- **Learning Rate Scheduling (ReduceLROnPlateau):** Instead of using a fixed learning rate throughout training, the framework utilizes a dynamic learning rate scheduler, specifically torch.optim.lr_scheduler.ReduceLROnPlateau. This scheduler monitors a specified performance metric on the validation set during training. In this implementation, the monitored metric is the validation directional accuracy (mode='max'). If this metric fails to improve for a specified number of epochs (the patience, set to 3), the scheduler reduces the current learning rate by multiplying it by a factor (set to 0.5). This strategy allows the model to make rapid progress early in training when the learning rate is higher and then fine-tune the weights more carefully later in training with smaller learning rates as performance plateaus. Tying the learning rate reduction specifically to *validation directional accuracy*, rather than the overall validation loss, underscores the importance placed on this particular performance aspect. It ensures that the optimization process adapts its step size based on improvements in the model's ability to correctly predict the direction of future price movements on unseen data, directly aligning the learning dynamics with a key objective.

4.2. Overfitting Mitigation and Model Selection

- **Training Loop (train_model function):** The train_model function encapsulates the training process for a single model instance. It follows a standard epoch-based structure. Within each epoch, it iterates through the train_loader, processing batches of sequences and targets. For each batch, it performs a forward pass through the model (outputs = model(seqs)), calculates the loss using the DirectionalLoss criterion (loss = criterion(...)), performs backpropagation to compute gradients (loss.backward()), and updates the model's weights using the optimizer (optimizer.step()). After processing all training batches, the model is switched to evaluation mode (model.eval()), and a similar loop iterates through the val_loader. During validation, predictions are made, and the validation loss is accumulated, but gradient calculations and weight updates are disabled (torch.no_grad()).

- **Early Stopping:** To prevent overfitting and potentially reduce training time, an early stopping mechanism is implemented. This mechanism also monitors the validation directional accuracy (val_dir_acc), calculated after each epoch by comparing the signs of the inverse-transformed predictions and targets on the entire validation set. A variable best_dir_acc tracks the highest validation directional accuracy achieved so far. Whenever a new highest accuracy is reached, the current model's state dictionary (model.state_dict()) is saved to a file (f'best_model_{model_idx}.pth'), and a patience_counter is reset to zero. If the validation directional accuracy does not improve in an epoch, the patience_counter is incremented. If the counter reaches a predefined patience threshold (set to 5 epochs), it signifies that the model's directional performance on the validation set has not improved for several consecutive epochs, potentially indicating the onset of overfitting or that further training is unlikely to yield benefits. At this point, the training loop is terminated (break).
- **Model Selection Criterion:** After the training loop finishes (either by completing all num_epochs or by early stopping), the function explicitly loads the weights from the saved file (model.load_state_dict(torch.load(f'best_model_{model_idx}.pth'))). This ensures that the model returned by the train_model function corresponds to the epoch that achieved the best validation directional accuracy, not necessarily the model from the final epoch.
  The consistent use of validation directional accuracy as the primary metric for both the learning rate scheduler and the early stopping/model selection criterion is highly significant. It demonstrates a clear strategic decision to prioritize the model's ability to generalize its directional predictions to unseen data above all other metrics, including the overall composite loss value. This goal-oriented training process aims to produce models that are optimized specifically for the task characteristic deemed most critical, likely reflecting an intended use case where correct directional signals are the primary output desired from the forecast.

5. Forecasting Future Price Movements

This section describes how the fully trained ensemble model is utilized to generate a prediction for the next hour's SOL-USD price and the necessary post-processing steps to convert the model's output back into a meaningful price value.

5.1. Generating Predictions with the Ensemble Model (predict_next_hour_ensemble function)

- **Input Preparation:** To generate a forecast for the hour immediately following the available data, the predict_next_hour_ensemble function first retrieves the most recent sequence of data points. It selects the last T (24) rows from the *scaled* DataFrame (df_scaled.iloc[features].values), ensuring it uses the same features that the models were trained on. This numpy array is converted into a PyTorch FloatTensor. Since the models expect input in batches, an extra dimension is added at the beginning using unsqueeze(0), creating a tensor of shape ``.

- **Ensemble Prediction:** This prepared input tensor (last_seq) is then passed to the predict method of the ensemble object (an instance of EnsemblePredictor). As detailed in Section 3.3, this method internally iterates through each of the constituent trained models (operating in evaluation mode and without gradient tracking), obtains their individual predictions for the input sequence, and computes the simple average of these predictions. The result is a single scalar value (pred_scaled), representing the ensemble's consensus prediction in the scaled space of the target variable (either scaled log price or scaled differenced log price).

5.2. Post-processing: Reconstructing Price Values

The raw output (pred_scaled) from the ensemble model is not directly interpretable as a price. It needs to be transformed back to the original price scale by reversing the preprocessing steps applied to the target variable during the data preparation phase (Section 2.4). This reconstruction is performed meticulously:

- **Inverse Scaling:** The first step is to reverse the scaling applied by the target_scaler (the MinMaxScaler fitted on the training target data). This is done using the scaler's inverse_transform method: pred_diff_or_log = target_scaler.inverse_transform([[pred_scaled]]). The input [[pred_scaled]] is shaped as a 2D array as required by the scaler. The output pred_diff_or_log now represents the prediction in the space of log prices or differenced log prices, depending on whether differencing was applied.
- **Inverse Differencing (Conditional):** The next step depends on the differenced flag, which was set during the stationarity check (Section 2.3).
    - If differenced is True, it means the model was trained to predict the *change* in the log price from one hour to the next. Therefore, pred_diff_or_log represents this predicted change. To obtain the predicted log price for the next hour (pred_log_close), this predicted change must be added to the *actual* log price of the *last observed hour*. Critically, the implementation retrieves this last actual log price from the Original_Log_Close column (last_log_close = df_scaled['Original_Log_Close'].iloc[-1]), which stored the log prices *before* any differencing or scaling was applied. The calculation is: pred_log_close = last_log_close + pred_diff_or_log. Using the original, undifferenced log close value as the base for reconstruction is essential to avoid accumulating errors and to ensure the predicted price level is correctly anchored.
    - If differenced is False, the model was trained to predict the log price directly. In this case, the inverse-scaled prediction pred_diff_or_log already represents the predicted log price, so pred_log_close = pred_diff_or_log.
- **Inverse Log Transformation:** Finally, regardless of whether differencing was applied, the predicted log price (pred_log_close) must be converted back to the actual price scale. This is achieved by applying the exponential function (the inverse of the natural logarithm): pred_price = np.exp(pred_log_close).

This careful, step-by-step post-processing sequence, which accurately reverses scaling, conditionally reverses differencing using the correct base value, and reverses the log transformation, is crucial for generating a final, meaningful price prediction from the model's scaled output. It demonstrates attention to detail in ensuring the integrity of the transformations throughout the pipeline.

6. Discussion

The analyzed framework represents a sophisticated and multifaceted approach to the challenging problem of hourly Solana price prediction. It integrates several advanced deep learning techniques and data processing methodologies.

6.1. Synthesis of Strengths

- **Hybrid Architecture:** The core SolanaPredictor model effectively combines the feature extraction capabilities of CNNs with the sequence modeling strengths of LSTMs, suitable for capturing complex spatio-temporal patterns in multivariate financial data.
- **Attention Mechanisms:** The inclusion of both learnable time-step attention (applied before CNNs) and fixed positional encoding (applied after CNNs) provides mechanisms for the model to focus on temporally relevant parts of the input sequence and maintain positional awareness, potentially enhancing predictive power.
- **Tailored Optimization:** The custom DirectionalLoss function is a significant strength, explicitly aligning the model's optimization objective with key financial forecasting goals by balancing robust value prediction (Huber loss) and direction prediction, further enhanced by class weighting to handle imbalances.
- **Performance-Driven Training:** The use of validation directional accuracy as the primary metric for both learning rate scheduling (ReduceLROnPlateau) and early stopping demonstrates a focused training strategy that prioritizes generalization of the directional signal, a critical aspect for many trading applications.
- **Robustness Measures:** Several components contribute to robustness: RobustScaler for input features, Huber loss for the value component of the loss function, and the EnsemblePredictor strategy which averages predictions from multiple models to reduce variance and improve stability.
- **Comprehensive Preprocessing:** The data pipeline is thorough, incorporating proactive outlier clipping, variance-stabilizing log transformation, extensive feature engineering (including standard indicators and custom signals), statistically-driven adaptive stationarity handling (ADF test and conditional differencing), and appropriate scaling techniques with leakage prevention.

6.2. Potential Weaknesses and Limitations

- **Hyperparameter Sensitivity:** The framework involves numerous hyperparameters (e.g., CNN channels, LSTM hidden units, sequence length T, learning rate, loss function weights,

attention/PE dimensions, dropout rates, optimizer parameters, patience values for LR scheduling and early stopping). The performance can be sensitive to these choices, and the provided code uses fixed values without demonstrating systematic optimization or sensitivity analysis.

- **Data Limitations:** The model relies solely on historical OHLCV data obtained from yfinance. Its predictive power might be limited by the exclusion of other potentially relevant data sources, such as order book depth, social media sentiment, blockchain-specific metrics (e.g., transaction counts, active addresses), or macroeconomic indicators. The chosen period and interval also constrain the historical context available.
- **Static Feature Set:** While the engineered feature set is comprehensive, it is predefined. The relative importance of these features is not assessed, and the framework does not incorporate automated feature selection or dynamic feature engineering techniques that might adapt the input representation over time.
- **Market Regime Shifts:** Financial markets, especially cryptocurrencies, are prone to sudden regime shifts or "black swan" events. While ensembling provides some robustness, models trained primarily on historical data may struggle to adapt quickly to unprecedented market conditions not represented in the training set. The framework lacks explicit mechanisms for detecting or adapting to such shifts dynamically.
- **Interpretability:** The complexity of the hybrid architecture, particularly the interplay between CNNs, attention mechanisms, parallel LSTMs, and the final heuristic combination, makes interpreting *why* the model makes a specific prediction challenging compared to simpler linear models or tree-based methods.
- **Heuristic Combination:** The final step combining value and direction predictions (value_adjusted = value_pred * (0.8 + 0.2 * dir_sign)) uses a fixed, somewhat arbitrary formula. While it attempts to modulate magnitude based on direction, its theoretical justification is less clear than other model components, and its effectiveness might be data-dependent.

6.3. Directions for Future Work

Based on the analysis, several avenues for future research and development emerge:

- **Hyperparameter Optimization:** Conduct systematic hyperparameter tuning using techniques like Bayesian optimization, random search, or grid search to potentially identify configurations that yield superior performance.
- **Alternative Architectures:** Explore state-of-the-art sequence modeling architectures, such as Transformer-based models (which rely heavily on self-attention), or investigate different CNN/RNN variants (e.g., GRUs, dilated CNNs). Graph Neural Networks could be considered if incorporating relational data (e.g., inter-asset correlations).
- **Expanded Feature Set:** Integrate diverse data sources beyond OHLCV, including sentiment analysis from news/social media, on-chain metrics specific to the Solana network, order book data for microstructure analysis, and relevant macroeconomic indicators.

- **Dynamic Modeling and Adaptivity:** Investigate methods for online learning or adaptive model updating to allow the framework to adjust more rapidly to changing market dynamics and potential regime shifts. Techniques like transfer learning or context-aware modeling could be explored.
- **Advanced Ensembling:** Experiment with more sophisticated ensemble techniques, such as stacking (training a meta-model to combine base model predictions), blending, or creating ensembles of models trained on different timeframes or data subsets to capture multi-scale patterns.
- **Interpretability Methods:** Apply model-agnostic interpretability techniques (e.g., SHAP, LIME) or architecture-specific methods (e.g., attention map visualization) to gain deeper insights into the model's decision-making process and identify key predictive features or time steps.
- **Probabilistic Forecasting:** Modify the model's output layer (e.g., using mixture density networks or quantile regression) to predict a probability distribution over future prices rather than just a single point estimate. This would provide valuable uncertainty quantification alongside the forecast.
- **Refined Value/Direction Combination:** Investigate alternative methods for combining the value and direction predictions, potentially learning the combination function itself or using probabilistic approaches to model their joint distribution.

7. Conclusion

This paper has provided a comprehensive technical examination of a Python-based deep learning framework designed for the hourly price prediction of Solana (SOL-USD). The analysis reveals a sophisticated system that thoughtfully integrates multiple advanced components. Key elements include the hybrid CNN-LSTM architecture (SolanaPredictor) designed to capture spatio-temporal patterns, the incorporation of attention mechanisms and positional encoding for enhanced temporal focus, a robust data preprocessing pipeline featuring outlier handling and adaptive stationarity treatment, and an ensemble strategy for improved stability.

A particularly notable aspect is the custom DirectionalLoss function and the associated training protocol (LR scheduling, early stopping) explicitly prioritizing validation directional accuracy. This demonstrates a clear alignment of the model's design and optimization process with the practical requirements of financial forecasting, where correctly predicting the direction of movement is often paramount. The framework thus represents a well-considered attempt to leverage deep learning for a complex financial task, moving beyond generic regression objectives.

While the framework showcases numerous strengths in its design and implementation, potential limitations related to hyperparameter sensitivity, data scope, interpretability, and handling extreme market shifts were also identified. Nevertheless, the detailed exposition presented here serves as a valuable case study and reference for researchers and practitioners interested in

applying and refining deep learning techniques for cryptocurrency time series analysis. Empirical validation against benchmark models and further exploration of the suggested future work directions would be necessary to fully ascertain the practical trading effectiveness and robustness of this specific implementation.