e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:09/September-2024 Impact Factor- 7.868 www.irjmets.com

DEVELOPING AN ADVANCED AI-BASED 24 CARAT GOLD PRICE PREDICTION MODEL IN INDIA UTILIZING LONG SHORT-TERM MEMORY (LSTM) NEURAL NETWORKS WITH AN ACCURACY OF 96.79%: A COMPREHENSIVE APPROACH TO FORECASTING MARKET TRENDS AND ECONOMIC INFLUENCES

Siddhant Ray[*1]

[*1]Student Pursuing The IB Diploma Programme, Jamnabai Narsee International School, Mumbai, Maharashtra, India.

DOI : https://www.doi.org/10.56726/IRJMETS61395

ABSTRACT

This research explores the application of neural networks to predict gold prices, leveraging machine learning techniques to enhance financial analysis based on historical data. The research involves the development of a predictive model utilizing data normalization, the Adam optimiser, and the mean square error function. The model was trained on historical prices in the last 20 years, achieving an impressively high accuracy of 96.79% in predicting the future price one year ahead. Despite the promising results, the research identified key challenges, such as the model's sensitivity to input data and the inherent complexity of financial markets. These findings emphasize the potential of machine learning in finance, while also highlighting the need for ongoing refinement and the incorporation of broader, more comprehensive datasets to improve the model's precision. This study essentially focuses on the interlacing of machine learning and the world of financial analysis, offering key insights for future research and practical applications in market prediction.

Keywords: Gold Predictor, Neural Networks, LSTM, Adam Optimiser, Parameter, Calculation.

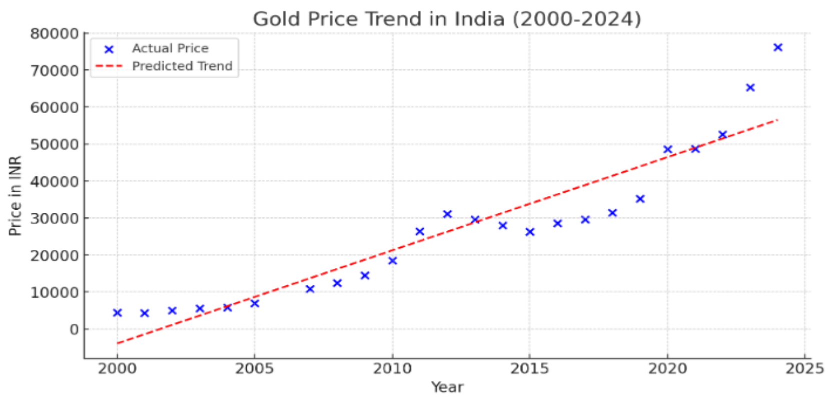
I. INTRODUCTION

In India, several stakeholders consider gold to be a lucrative investment due to its periodic annual appreciation. Historically, gold has served as a safe haven—a barricade against inflation and currency fluctuations, serving as a sense of security and stability in uncertain economic conditions. The upward curve in the price of gold encourages enterprises and individuals to invest in gold, leading to substantial returns in the long run. Culturally looked at, gold holds high value in India, especially on occasions such as weddings and festivals, further boosting its demand, and ensuring a steady rise in its value. The apparently ever-increasing nature of the price of gold makes it easy to develop a trend in the price of gold in the future. This apparent predictability can make investments more systematic and precise for stakeholders in both the short run and the long run. Given the vital role that gold plays in the Indian economy, developing an advanced AI-based gold price vaticination model, specifically acclimatized for the Indian market, can provide significant advantages.

For reasons of market efficiency, forecasting the price of gold should be difficult. [1] However, other authors have successfully applied neural networks, including long short-term memory (LSTM) networks, to forecasting the price of gold. Livieris, Pintelas and Pintelas successfully combined convolutional neural networks (CNNs) with LSTM networks to forecast the price of gold. [2] Whilst Yurtsever created models for forecasting the price of gold using an LSTM, a bidirectional LSTM (Bi-LSTM) and gated recurrent units (GRU). [3] The LSTM model performed the best. Liang, Lin and Lu proposed a novel decomposition-ensemble model for forecasting gold spot and futures prices. [4] First, gold prices were decomposed into sublayers with different frequencies by the improved complete ensemble empirical model decomposition with adaptive noise (ICEEMDAN) method. Next, LSTM, CNN and a convolutional block attention module (LSTM-CNN-CBAM) were used to forecast the sublayers. Finally, the forecasts for the sublayers were summed. The model performed well. By utilizing LSTM architectures, we can capture intricate temporal patterns and economic influences that drive gold prices. This AI-based gold price predictor enables investors, firms, and policymakers to optimize their investment strategies and maximize returns. The rest of the paper provides details of how to build such a model.

II. DEVELOPING A TREND IN THE PRICE OF GOLD IN INDIA IN THE LAST 24 YEARS

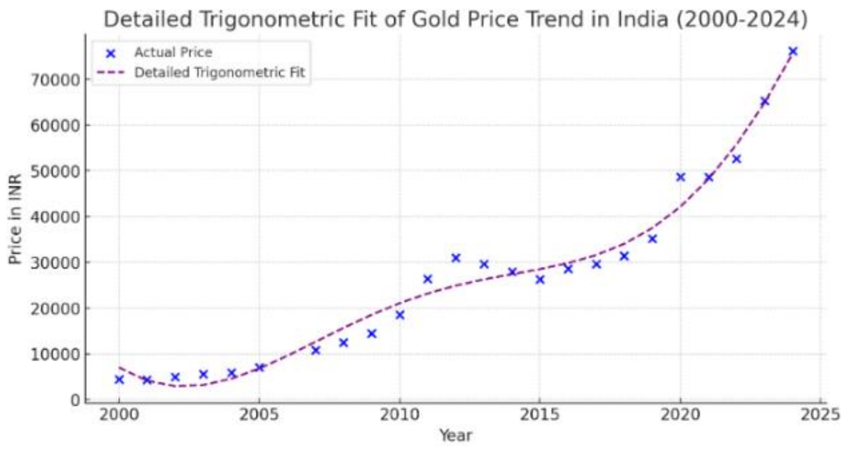
Linear model (incorrect and inaccurate):



Linear model function:

Price of Gold per 10 grams in India = (2519.62×Year) − 5,043,172.14

Trigonometric function (partially accurate but incorrect)



Trigonometric Function:

Price = 101465.48−27061.55×sin(0.235·Year−71.77) −109158.25×cos(0.100×Year)

III. WHAT IS THE CORRECT FUNCTION?

At first glance, the trigonometric function looks more accurate than the linear model. However, both the linear and the trigonometric functions are inaccurate. The linear function is evidently inaccurate since most of the values are off the line of best fit. A trigonometric function is known for its periodic (repeating) nature. Since the price of gold in the last 24 years does not inherently follow a simple or repetitive pattern, even a trigonometric function is inaccurate for the same reason. Hence, for a clearer analysis of this raw data, it is necessary to implement an LSTM (long short-term memory), a type of recurrent neural network. The primary advantage of LSTM networks lies in their ability to capture long-term dependencies and patterns in sequential data. LSTM networks can learn from past price movements and external variables to provide more accurate and reliable forecasts. Their ability to handle non-linearities and complex patterns makes them a preferred choice for financial forecasting tasks, ensuring that predictions are based on a robust understanding of historical data and potential future fluctuations.

IV. EXPLAINING LSTM (LONG SHORT-TERM MEMORY) THROUGH AN EXAMPLE

LSTM (Smart Student)

LSTM (Long Short-Term Memory) is like a smart student in a classroom. This student is smart since the student has an excellent memory and can remember important and unimportant things for long periods. It can also know when to forget the insignificant things.

Notebook (Cell State)

The smart student carries a notebook where he/she jots down important notes. This notebook acts as the 'Cell State' where important information is stored.

Important v/s Unimportant (Gates)

The student uses three types of pens to manage his/her notes.

- **Forget Pen (Gate)**: This pen (gate) is used to mark what information can be forgotten from the notebook (cell state).
- **Input Pen (Gate)**: This pen (gate) is used to decide what new information should be added to the notebook (cell state).

Output Pen (Gate): This pen (gate) is used to decide what information from the notebook (cell state) should be shared or used to answer questions (commands).

V. HOW LSTM (LONG SHORT-TERM MEMORY) WORKS FROM THE POINT OF VIEW OF THE SAME STUDENT EXAMPLE?

- The student starts with a clean notebook (null cell state).
- As the lesson progresses, the student gathers new information (input).
- The student uses the forget pen (forget gate) to cross out any information in the notebook (cell state) that is no longer needed.
- The student decides what new information is important and uses the input pen to add it to the notebook (cell state).
- When asked a question, the student uses the output pen to refer to the notes in the notebook (cell state) and gives the correct/matching answer.

VI. HOW LSTM (LONG SHORT-TERM MEMORY) WORKS IN THE GOLD PREDICTOR

There are two major steps before layers are formed in the LSTM neural network:

1) Our data consists of years from 2000 to 2023 and the price of gold in each corresponding year. Before feeding this data into the LSTM, it is scaled between 0 and 1 in a process called normalization which helps the neural network to train more efficiently.
2) The LSTM model is created using the '**Sequential**' API in Keras, which allows stacking layers sequentially. [5, 6]

VII. LAYERS PRESENT IN THE LSTM NEURAL NETWORK IN THE GOLD PREDICTOR

Layers in our LSTM model help the model learn different patterns of data. Layers could be thought of as stages of understanding/learning. The LSTM model consists of three LSTM layers followed by a dense layer. However, it is not necessarily the case that more LSTM layers lead to a more accurate model. Following is the explanation of each LSTM layer with the respective syntax in the code:

Layer 1: First LSTM Layer

Syntax: LSTM(200, input_shape=(1, 1), return_sequences=True)

```
LSTM(200, input_shape=(1, 1), return_sequences=True),
```

- In my code, I have defined the LSTM Layer to contain 200 units/memory cells.
- Each cell can remember important information from the input data.
- Each input sample will have one time step and one feature (in our case, the normalized year).
- The final part of the syntax tells the LSTM to return the full sequence of outputs for each input sequence. This is necessary since the next LSTM layer will need the full sequence of outputs.

Layer 2: Second LSTM Layer

Syntax: LSTM(100, return_sequences=True)

```
LSTM(100, return_sequences=True),
```

- I have defined this LSTM layer to hold 100 units.

- Once again, like previously, the full sequence of outputs is returned to be used by the next LSTM layer.

Layer 3: Third LSTM Layer

Syntax: LSTM(50)

```
LSTM(50)
```

- This layer is defined to have 50 units.
- In this case, the '**return_sequence**' is not specified since it equates to '**False**' by default.
- This essentially means only the last output in the sequence will be returned, which is suitable for the following layer (Dense layer).

Layer 4: Dense Layer

Syntax: Dense (1)

```
Dense(1)
```

- A dense layer has one unit.
- This layer takes the output from the last LSTM layer and produces a single value, which is the predicted gold price.

VIII. MEANING OF WEIGHTS IN THE CONTEXT

In the context of machine learning and neural networks, "weights" are the parameters within the model that are learned from the training data. These weights are crucial because they determine how the input data is transformed as it passes through the network to produce the final output. Given below is a breakdown of weights and their significance:

- **Neurons and Layers**: A neural network is made up of layers with each layer consisting of neurons (nodes). Neurons in a particular layer are connected to neurons in the succeeding layer with the help of weights.
- **Functions of weights**: Weights aid in controlling the strength and direction of a connection between neurons. Essentially, weights determine how a neuron's output will be impacted by a change in a particular input.
- **Learning weights**: During the training period, the model adjusts its weights based on the error between the predicted output and the actual output. This adjustment is done using optimization algorithms like gradient descent, which eventually find the set of weights that minimize the error.
- **Impact of weights**: Weights are the core parameters that help to define the behaviour of a neural network. Weights that have undergone sufficient training enable the model to make predictions of new, unseen data.

IX. MATHEMATICAL REPRESENTATION OF WEIGHTS IN AN LSTM MODEL

When data is passed through a neural network, it undergoes a set of mathematical operations. Each neuron performs a weighted sum of inputs, followed by an activation function.

$$Z = (W_1 \cdot X_1) + (W_2 \cdot X_2) + \cdots + (W_n \cdot X_n) + b$$

MATHEMATICAL FORMULA FOR WEIGHTS IN AN LSTM MODEL

Z = Neuron's output before activation

W_1, W_2, W_n = Weights

X_1, X_2, X_n = Inputs

b = Bias term (separate learnable parameter)

The result Z is then passed through an activation function to introduce non-linearity.

X. IMPORTANCE OF WEIGHTS

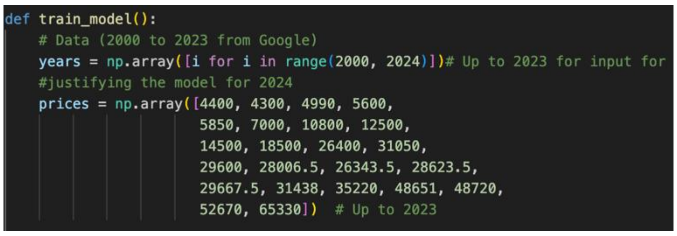
- **Pattern recognition**: Weights enable the pattern recognition feature in the input data. Different sets of weights help to capture different patterns, making them extremely crucial for tasks like time series forecasting, image recognition, and speech recognition.

- **Training and Learning**: Finding the optimal (perfect) weights is essentially the process of training the neural network. Effective training algorithms adjust these weights in a way that the model gains accuracy over time.
- **Model behaviour**: The specific values of weights determine how the model processes inputs and generates outputs. They directly affect the accuracy, performance, and ability of the model to generalize data.

XI. FACTORS AFFECTING THE ACCURACY OF AN LSTM NETWORK/MODEL

- **Depth (Number of Layers)**: Adding more layers can help the model learn more complex patterns. However, adding too many layers could lead to overfitting, a scenario in which the model performs well on training data but poorly on unseen data. More layers could also mean computation complexity and longer training times.
- **Quality of data**: The quality of data is extremely crucial. Adding more layers has no impact if the data is noisy or insufficient. Properly preprocessed and clean data can significantly improve model accuracy.
- **Quantity of data**: The quantity of data is crucial since the more data provided, the more the model can learn diverse patterns and nuances. With limited data, the model might memorize the training data instead of learning models; more data essentially helps to prevent overfitting. There is also a reduction in variance which refers to how much the model's prediction changes with different subsets of training data. Higher variance can lead to overfitting. More data also provides higher consistency in the training process, leading to reliable and robust models. Lastly, if the data has complex relationships/patterns, a larger dataset helps capture these complexities better than a smaller dataset. Larger datasets also increase the chances of capturing rare events/outliers, which the model can learn to handle properly.
- **Learning rate**: This is a crucial hyperparameter in the training of LSTM models. It controls the size of the steps the model takes in updating its weights during training as it attempts to minimize the loss calculation function. A very high learning rate may cause the model to converge too quickly to a suboptimal solution which may not be accurate, or it might completely overshoot the minimum. A very low learning rate might result in a long training process that could get stuck in a local minimum, missing the global minimum.
- **Optimal learning rate**: Finding the perfect learning rate is crucial because it balances the speed and quality of convergence. Techniques including learning rate schedules (where the learning rate gradually decreases over training epochs) or adaptive learning methods (like Adam) are very helpful.
- **Batch Size**: This hyperparameter defines the number of training samples to work through before the model's internal parameters are updated. A small batch size can provide a regularizing effect and lower generalization error. It makes the learning process noisier, potentially assisting the model in escaping the local minima. A large batch size can compute the gradient more accurately, but it might lead to a smooth optimization landscape that could make it harder to generalize.
- **Regularization techniques**: The purpose of regularization techniques is to prevent the model from overfitting, which is crucial when working with a complex model like an LSTM. During optimization, L1 and L2 regularization add penalties to layer activity/layer parameters. L2 regularization is particularly useful in reducing overfitting by penalizing large weights. Dropout is another regularization technique to prevent overfitting which randomly sets input elements to zero during training at a pre-defined rate, which helps in breaking happenstance correlations in the training data.

XII. TRAINING THE MODEL



```
def train_model():
    # Data (2000 to 2023 from Google)
    years = np.array([i for i in range(2000, 2024)])# Up to 2023 for input for
    #justifying the model for 2024
    prices = np.array([4400, 4300, 4990, 5600,
                       5850, 7000, 10800, 12500,
                       14500, 18500, 26400, 31050,
                       29600, 28006.5, 26343.5, 28623.5,
                       29667.5, 31438, 35220, 48651, 48720,
                       52670, 65330])  # Up to 2023
```

The above syntax is the data that is fed into the program for training the model to develop a trend for the price prediction.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:09/September-2024 Impact Factor- 7.868 www.irjmets.com

The syntax begins with the definition of a Python function called '**train_model**'.

'**years**': This variable is an array of integers from 2000 to 2023. These represent the years for which the model will have the price data to learn from. The array is created using a list comprehension that iterates over a range generated by '**range(2000, 2024)**'. This helps generate a range beginning at 2000 and ending at 2023 since the stopping index in Python is exclusive.

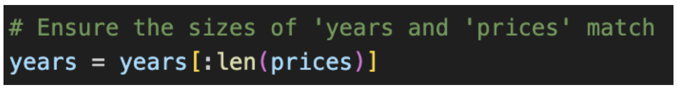
'**prices**': This variable is an array of gold prices corresponding to each year in the '**years**' array. These prices are manually input and represent the historical gold prices for each year up to 2023. The data is collected from Google, as I have indicated in the comment for easier understanding.

PURPOSE OF THIS DATA:

The sole purpose of this data is to act as a dataset for the model. '**years**' will be used as the input features, and '**prices**' will be the target values the model looks to predict. This is a typical standard setup for regression models where the goal is to predict a continuous value (price of gold) based on the input (year).

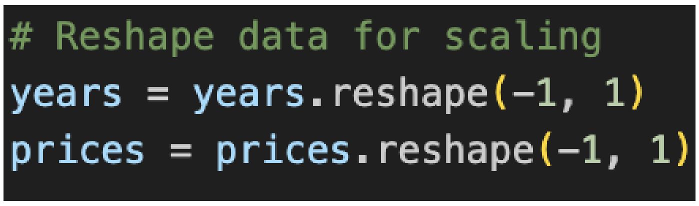
HOW THIS DATA MIGHT BE USED:

- **Preprocessing**: The data might be preprocessed to be suitable for training. For example: '**years**' might be normalized or scaled to a range that is more effective for training neural networks.
- **Model Training**: A machine learning model, possibly a time series model like LSTM, will be trained on this data. The model will learn the relationship between the years and the prices, aiming to predict future prices based on the year.



```
# Ensure the sizes of 'years and 'prices' match
years = years[:len(prices)]
```

This part of the code highlights one of the functions of the previously defined arrays '**years**' and '**prices**'. To recall, '**years**' consists of the range of years from 2000 to 2023, and '**prices**' consists of the range of prices of gold from 2000 to 2023. The main function of the ':**len**' syntax is to create a new list that includes elements only from the 'years' from the start, but excludes the final element, i.e. '**len(prices)**'. Essentially, it is used to truncate the '**years**' list to the same length as the '**prices**' list. It mainly ensures that '**years**' and '**prices**' have the same length (number of items). This form of alignment is essential for the modelling part of the gold predictor model where each year needs to correspond to a specific price. Ensuring the lists are of equal length prevents indexing errors and maintains data integrity.



```
# Reshape data for scaling
years = years.reshape(-1, 1)
prices = prices.reshape(-1, 1)
```

Data Reshaping:

In machine learning algorithms, data reshaping is often needed for algorithms that require the inputs in a specific shape.

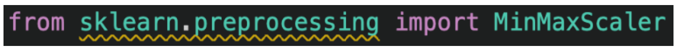
In the above example, '**years**' and '**prices**' are like numpy arrays, and the '**.reshape(-1,1)**' method is used to convert these arrays into 2-dimensional arrays with one column. This is usually done before scaling the data or fitting it into the model.

In the '**.reshape**' method, '**-1**' indicates that the number of rows should be inferred from the length of the array and the specified number of columns. The '**1**' indicates that the reshaped array should have one column.

Advantages of reshaping data:

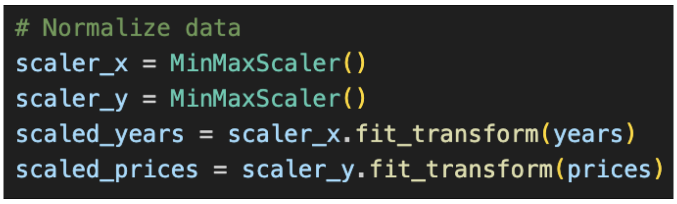
- **Compatibility**: Many ML Libraries expect the input feature to be in a 2D array. If not, the data isn't processed.
- **Scaling**: If scaling is necessary (which in our case it is), these scalers expect a 2D input.

XIII. NORMALIZING/SCALING THE DATA



```
from sklearn.preprocessing import MinMaxScaler
```

In our code, we have imported the '**MinMaxScaler**' class from '**sklearn.preprocessing**'. [7, 8]



```
# Normalize data
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
scaled_years = scaler_x.fit_transform(years)
scaled_prices = scaler_y.fit_transform(prices)
```

Normalizing (or scaling) is a preprocessing step used to adjust the range of features in the data. The '**MinMaxScaler**' scales and translates each feature individually such that it is in the given range in the training set, typically between 0 and 1.

In the code, '**scaler_x**' and '**scaler_y**' are instances of the '**MinMaxScaler**' class from scikit-learn.

The '**.fit_transform**' function does two things:

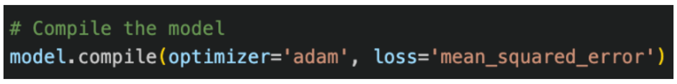
- Computes the minimum and maximum values of the data.
- Scales the data to the specified range using the computed minimum and maximum values.

Advantages of Normalizing/Scaling:

- **Improves convergence**: Many machine learning algorithms perform better or converge faster when normalized/scaled.
- **Prevents dominance**: Features with larger ranges might dominate the learning process, causing the model to prioritize them over others.
- **Enhances performance**: Algorithms like gradient descent-based methods, k-means clustering, and others can greatly benefit from feature normalizing/scaling.

In essence, the '**years**' and '**prices**' are normalized using the '**MinMaxScaler**' from scikit-learn. This scaler transforms the data to a range of [0,1], which is beneficial for many machine learning algorithms. The normalization process involves fitting the scaler to the data to determine the minimum and maximum values and then transforming the data to a specified range.

XIV. MODEL COMPILATION



```
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

This line is essentially used in the context of training a neural network model, specifically with the framework of Keras or TensorFlow. [9]

The '**model.compile**' method configures the model for training.

Adam Optimiser:

Adam (Adaptive Moment Estimation) is an optimization algorithm that combines the best properties of **AdaGrad** and **RMSProp** algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. [10]

Key characteristics of Adam:

- **Adaptive Learning Rate**: Adam computes individual adaptive learning rates for different parameters.
- **First and Second Moment Estimation**: Adam estimates the mean and the uncentered variance of gradients to help adapt the learning rate for each parameter.
- **Bias Correction**: Adam has bias correction mechanisms to aid in adjusting and fixing the estimates of the mean and the uncentered variance.

Adam Update Rule:

Initialization:

- Initialize the first-moment vector $m_t = 0$
- Initialize the second-moment vector $v_t = 0$

- Initialize timestep t = 0

Meaning of first-moment vector, second-moment vector and timestep in our code

- The first-moment vector means the exponentially decaying average of past gradients. It is a moving average of the gradients. The main reason for implementing the first-moment vector is to help level out the gradients, which eventually alleviates the effect of noise / sudden spikes in the gradient values.

CALCULATION:

- At each timestep t, the gradient g_t is calculated and the first-moment vector is updated as $m_t = \beta_1 m_{t-1}+(1-\beta_1) g_t$.
- Here, β_1 (typically set to 0.9) controls the decay rate. The trend is followed in a manner that a higher value of β_1 results in a slower decay and more influence from the history of gradients.
- The second-moment vector, denoted as v_t, represents the exponentially decaying average of the past squared gradients. It helps to normalize the gradients by taking into account their magnitude (size). This eventually helps in controlling the learning rate by adjusting for large gradients that could lead to overshooting.

CALCULATION:

- The second moment vector is updated as $v_t = \beta_2 v_{t-1}+(1-\beta_2) g^2{}_t$.
- Here, β_2 (typically set to 0.999) controls the decay rate. A higher β_2 results in slower decay and more influence from past squared gradients.
- The timestep, denoted as t, is a counter helping to track the number of updates or iterations. Timestep is extremely crucial for bias correction, which is one of the features of the Adam optimiser. Bias correction essentially sets these vectors to account for the initialization bias, leading to more stable and accurate parameter updates.
- The bias-corrected first moment vector: $m't = v_t/1-\beta_1{}^t$.
- The bias-corrected second moment vector: $v't = v_t/1-\beta_2{}^t$.
- These corrected values ensure that in the early stages of training, when t is small, the vectors are appropriately scaled, preventing them from being underestimated.

HOW THE ADAM OPTIMISER FUNCTIONS WHEN THE CODE IS COMPILED

- The first moment vector smooths out the gradient over time and helps avoid the influence of large, noisy gradients.
- The second-moment vector tracks the variance (size) of the gradients and helps normalize the learning rate by adjusting for large gradients.
- The timestep t keeps track of the number of iterations and ensures proper bias correction for the first and second-moment vectors.
- All these elements coordinate with each other to make sure the Adam optimiser adaptively adjusts to the learning rates for each parameter, leading to more efficient and stable convergence during the training of the LSTM.

PARAMETER UPDATES IN ADAM OPTIMISER IN A NUTSHELL

- $t = t + 1$
- $m_t = \beta_1 m_{t-1}+(1-\beta_1) g_t$
- $v_t = \beta_2 v_{t-1}+(1-\beta_2) g^2{}_t$
- $m't = v_t/1-\beta_1{}^t$
- $v't = v_t/1-\beta_2{}^t$
- $\theta_t = \theta_t-1-\alpha\ mt/(vt+\epsilon)$

Here, g_t is the gradient at timestep t, β_1 and β_2 are hyperparameters, α is the learning rate, and ϵ is a small constant to prevent division by zero (leads to undefined otherwise).

MEAN SQUARED ERROR (MSE)

The mean squared error (MSE) is a common loss function used for regression tasks and functions. [11] It measures the average squared difference between the actual values and the predicted values.

Mathematical formula:

$$\text{MSE} = \overset{\text{Mean}}{\boxed{\frac{1}{n}\sum_{i=1}^{n}}}\overset{\text{Error}}{\boxed{(Y_i - \hat{Y}_i)}}^{\overset{\text{Squared}}{\boxed{2}}}$$

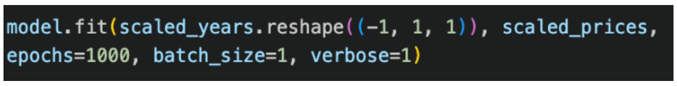
- n is the number of data points
- Y_I is the actual value
- $\hat{Y}_{Ii}$ is the predicted value

Characteristics of Mean Squared Error:

- Larger errors are penalized more due to the squaring of differences.
- It facilitates a smooth gradient, which is extremely useful for gradient-based optimization algorithms like Adam.

To summarize, the Adam optimiser helps configure the neural network model to use the mean squared error loss function during the training of the model. The Adam optimiser adapts the learning rate for each parameter, combining the benefits of AdaGrad and RMSProp, while the MSE Loss function measures the average squared differences between actual and predicted values, providing a smooth gradient for optimization.

XV. TRAINING THE MODEL II



```
model.fit(scaled_years.reshape((-1, 1, 1)), scaled_prices,
epochs=1000, batch_size=1, verbose=1)
```

- '**.fit**' here is the method used to train the model on the provided data.
- '**.scaled_years**' is the input data that has been scaled.
- '**reshape ((-1, 1, 1))**' reshapes the '**scaled_years**' array into a 3-dimensional array with the shape '**(number_of_samples, 1, 1)**'.
- '**-1**' automatically calculates the number of samples based on the length of the '**scaled_years**' array.
- '**1**' means the second dimension is set to 1, indicating that each sample contains only one feature.
- '**1**' means the third dimension is also set to 1, which can be useful for certain types of neural network layers (it is useful in our case of LSTM used for time-series analysis).
- '**scaled_prices**' is the target data (or labels) that has been scaled. The model will learn how to predict these values from the input data.
- '**epochs=1000**' specifies the number of epochs, or complete passes through the training dataset. In our case, the model will iterate and reiterate over the entire training data 1000 times to increase efficacy, accuracy and reliability.
- '**batch_size=1**' specifies the number of samples per gradient update. Here, a batch size of 1 means that the model will update the weights after each sample (online learning). This can be used for models with very small datasets (like our model) where each sample needs to be processed individually, which helps improve accuracy.
- '**verbose=1**' specifies the verbosity mode. '**verbose=1**' means that the progress of the training will be displayed as a progress bar in the terminal.

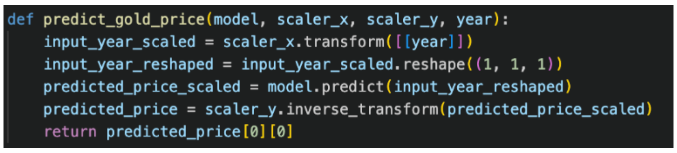
PUTTING IT ALL TOGETHER:

- '**scaled_years**' is reshaped to have three dimensions, which is necessary for certain types of neural network layers.
- The model is trained on the '**scaled_years**' data to predict '**scaled_prices**' over 1000 epochs, updating the weights after each sample (**batch_size = 1**).
- The training progress is displayed in the terminal as '1/1000 epochs completed', '2/1000 epochs completed', '3/1000' epochs completed, and so on.

```
return model, scaler_x, scaler_y
```

This function returns the '**model**', '**scaler_x**' (scaled years), and '**scaler_y**' (scaled prices) objects.

XVI. 'PREDICT_GOLD_PRICE' FUNCTION EXPLANATION



```
def predict_gold_price(model, scaler_x, scaler_y, year):
    input_year_scaled = scaler_x.transform([[year]])
    input_year_reshaped = input_year_scaled.reshape((1, 1, 1))
    predicted_price_scaled = model.predict(input_year_reshaped)
    predicted_price = scaler_y.inverse_transform(predicted_price_scaled)
    return predicted_price[0][0]
```

The '**predict_gold_price**' is a function which takes into consideration the trained model, and the corresponding scalers with specific matching years, and returns the predicted gold price for that year.

EXPLAINING THE PARAMETERS:

- '**model**' is the trained neural network (LSTM) model used for making the predictions for the price of gold for the specific input year.
- '**scaler_x**' is the specific scaler used to normalize the input data (years) during the training phase.
- '**scaler_y**' is the scaler used to normalize the target data, prices of gold, during the training phase.
- '**year**' is the specific year for which the gold prediction is to be made.

SCALING THE INPUT YEAR

Syntax: input_year_scaled = scaler_x.transform([[year]])

- '**scaler_x.transform([[year]])**' makes use of the '**transform**' method of the '**scaler_x**' which helps scale the input year. This year is provided as a 2D array '**[[year]]**' because the scaler expects input in this format.
- This helps ensure that the input year is scaled in the same way and format as the training data.

RESHAPING THE SCALED INPUT YEAR

Syntax: input_year_reshaped = input_year_scaled.reshape((1, 1, 1))

- '**input_year_scaled.reshape((1,1,1))**' reshapes the scaled input year to a 3D array with shape '**(1,1,1)**'. This shape is required by the model for making predictions.
- The model has been trained with input data of shape '**(number_of_samples, 1, 1)**', hence the input for prediction needs to match this shape.

PREDICTING THE SCALED PRICE

Syntax: predicted_price_scaled = model.predict(input_year_reshaped)

- '**model.predict(input_year_reshaped)**' makes use of the trained model to predict the price of gold for the scaled and reshaped year input into the terminal window.
- The output now is the predicted price, however, it is still in the scale form.

INVERSE TRANSFORM FUNCTION FOR THE PREDICTED PRICE

Syntax: predicted_price=scaler_y.inverse_transform(predicted_price_scaled)

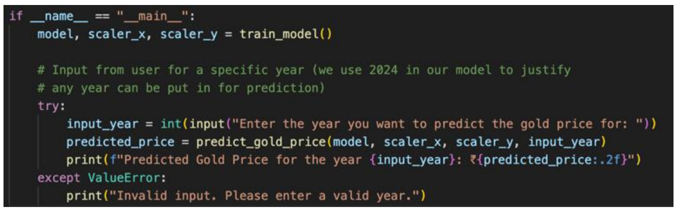
- '**scaler_y.inverse_transform(predicted_price_scaled)**' helps convert the predicted price from the scaled form back to the original scale (i.e. the actual price of gold).
- This step is crucial to interpret the predicted price of gold in a meaningful and feasible way.

RETURNING THE PREDICTED PRICE OF GOLD

Syntax: return predicted_price[0][0]

- ‘**predicted_price[0][0]**’ is a 2D array and it extracts the single predicted value from this array.
- This value is the ultimate and final predicted gold price for the year entered into the terminal window.

XVII. ENTRY POINT OF THE PROGRAM



```python
if __name__ == "__main__":
    model, scaler_x, scaler_y = train_model()

    # Input from user for a specific year (we use 2024 in our model to justify
    # any year can be put in for prediction)
    try:
        input_year = int(input("Enter the year you want to predict the gold price for: "))
        predicted_price = predict_gold_price(model, scaler_x, scaler_y, input_year)
        print(f"Predicted Gold Price for the year {input_year}: ₹{predicted_price:.2f}")
    except ValueError:
        print("Invalid input. Please enter a valid year.")
```

Syntax: if __name__ == "__main__":

- If the script is executed directly in the Visual Studio Environment, the code within this block will be executed.
- This is a regularly used Python standard code that allows parts of the code to be reused as a module without executing the block.

Syntax: model, scaler_x, scaler_y = train_model()

- This function is used to train the neural network model.
- ‘**model**’ returns the trained neural network model.
- ‘**scaler_x**’ is the scaler used to normalize the input data (years).
- ‘**scaler_y**’ is the scaler used to normalize the target data (prices).

Syntax: input_year = int(input("Enter the year you want to predict the gold price for: "))

- The ‘**input**’ function prompts the user to enter a year for which they want to predict the price of gold for.
- The ‘**int**’ function converts the input of the user (taken by the terminal as a string) into an integer. This is crucial because the expected year is supposed to be a numeric value.

Syntax: predicted_price = predict_gold_price(model, scaler_x, scaler_y, input_year)

- This calls the ‘**predict_gold_price**’ function, explained earlier.
- This passes the arguments of ‘**model**’, ‘**scaler_x**’, ‘**scaler_y**’, and the ‘**input_year**’ provided by the user.
- The function returns the predicted gold price for the specified year, which is stored in the ‘**predicted_price**’ variable.

Syntax: print(f"Predicted Gold Price for the year {input_year}: ${predicted_price:.2f}")

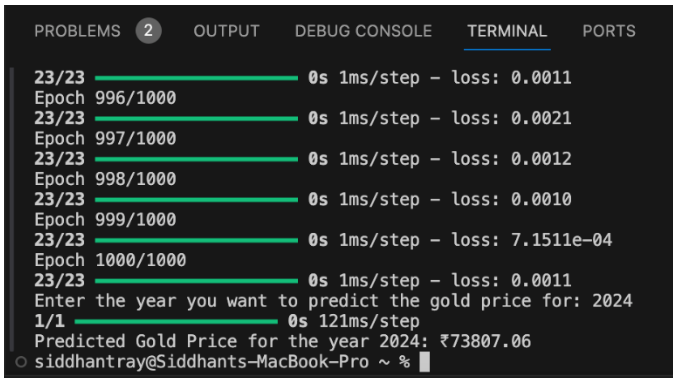
- The ‘**print**’ function uses an f-string (formatted string) to display the result.
- ‘**{input_year}**’ inserts the user-provided year into the string.
- ‘**${predicted_price:.2f}**’ inserts the predicted price into the string, formatted to **2** decimal places for currency display.

Syntax: except ValueError: print("Invalid input. Please enter a valid year.")

- The ‘**except ValueError:**’ catches exceptions that occur if the user inputs something that cannot be converted to an integer (for example: a non-numeric string).
- ‘**print(“Invalid input. Please enter a valid year.”)**’ is used to display this message if a ‘**ValueError**’ is raised; this prompts the user to enter a valid year instead of the non-integer value that has been entered.

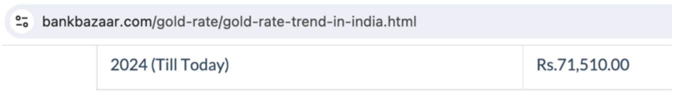
XVIII. JUSTIFICATION OF THE MODEL

In order to truly test the accuracy of the model, I did not feed in the data for 2024 and asked the model to predict the value of gold (10 grams) for 2024. Below is the comparison between the model value and the literature value as per bankbazaar.com:



```
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0011
Epoch 996/1000
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0021
Epoch 997/1000
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0012
Epoch 998/1000
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0010
Epoch 999/1000
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 7.1511e-04
Epoch 1000/1000
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0011
Enter the year you want to predict the gold price for: 2024
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 121ms/step
Predicted Gold Price for the year 2024: ₹73807.06
siddhantray@Siddhants-MacBook-Pro ~ %
```

PREDICTED GOLD PRICE (10 GRAMS OF GOLD) FOR THE YEAR 2024 AS PER MY MODEL: ₹73,807.06



bankbazaar.com/gold-rate/gold-rate-trend-in-india.html

2024 (Till Today)	Rs.71,510.00

ACTUAL GOLD PRICE (10 GRAMS OF GOLD) FOR THE YEAR 2024 AS PER BANKBAZAAR.COM: ₹71,510
CALCULATING THE ACCURACY:

Percentage Accuracy = (1 - |True Value – Predicted Value| / True Value) × 100

True value: ₹71,510

Predicted value: ₹73,807.06 (approximately ₹73,807)

Percentage accuracy

= (1 - |71510 – 73807| / 71510) × 100

= (1 - 2297 / 71510) × 100

= (1 – 0.03212) × 100

= 0.96788 × 100

= 96.79%

Hence, our model forecast the price of gold one year ahead with an accuracy of 96.79%.

XIX. CONCLUSION

To conclude, the development of a gold price prediction model using a neural network highlights the potential and the challenges inherent in applying machine learning to financial forecasting. The model demonstrated a high degree of accuracy, as can be seen by the one-year-ahead percentage accuracy of 96.79%, portraying the effectiveness of methods such as normalization, Adam optimizer, root mean square, etc. However, the model has several limitations. It is sensitive to input data. Also, it is difficult to take into account the changes in real-world market variables. This calls for the need for a more comprehensive dataset and continuous improvement. While the results are promising in the short run, for example one or two years down the line, further research is needed to refine and enhance the model further. This will also ensure the robustness of the model in varying market conditions. Overall, this research underscores the potential of machine learning in financial analysis of data based on historical data, while also implicitly accounting for the complexities and intricacies in real-world data such as future economic events and market instabilities.

ACKNOWLEDGEMENT

I wish to thank Dr Martin Sewell, BSc (Hons), MSc, PhD for his assistance with this paper.

XX. REFERENCE

[1] E. F. Fama, “Efficient Capital Markets: A Review of Theory and Empirical Work,” The Journal of Finance, vol. 25, May 1970, pp. 383-417, doi: 10.1111/j.1540-6261.1970.tb00518.x.

[2] I. E. Livieris, E. Pintelas and P. Pintelas, “A CNN–LSTM model for gold price time-series forecasting,” Neural Computing and Applications, vol. 32, Dec. 2020, pp. 17351–17360, doi:10.1007/s00521-020-04867-x.

[3] M. Yurtsever, “Gold Price Forecasting Using LSTM, Bi-LSTM and GRU,” European Journal of Science and Technology, vol. 31, Dec. 2021, 341-347, doi:10.31590/ejosat.959405.

[4] Y. Liang, Y. Lin and Q. Lu, “Forecasting gold price using a novel hybrid model with ICEEMDAN and LSTM-CNN-CBAM,” Expert Systems with Applications, vol. 206, Nov. 2022, 117847, doi:10.1016/j.eswa.2022.117847.

[5] F. Chollet, The Sequential model, Keras, Apr. 2020, https://keras.io/guides/sequential_model/.

[6] Model training APIs, Keras, https://keras.io/api/models/model_training_apis/.

[7] Preprocessing data, scikit-learn, https://scikit-learn.org/stable/modules/preprocessing.html.

[8] J. Brownlee, How to Use StandardScaler and MinMaxScaler Transforms in Python, Machine Learning Mastery, Aug. 2020, https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/.

[9] tf.keras.Model, TensorFlow, https://www.tensorflow.org/api_docs/python/tf/keras/Model.

[10] V. Bushaev, Adam — latest trends in deep learning optimization, Towards Data Science, Oct. 2018, https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

[11] Wikipedia, Mean squared error, Jun. 2024, https://en.wikipedia.org/wiki/Mean_squared_error.