Forecasting Cryptocurrency Prices by using Elman Neural Network Model

Enes Celik

July 8, 2024

Abstract

In the rapidly evolving domain of cryptocurrency, the ability to accurately predict price movements represents a significant advantage for traders, investors, and policy makers. The inherent volatility and unpredictability of cryptocurrency prices, driven by complex factors such as market sentiment, regulatory changes, and macroeconomic indicators, present a formidable challenge to traditional forecasting methods. In this context, the Elman Neural Network (ENN), a type of recurrent neural network (RNN), emerges as a powerful tool for modeling and forecasting time-series data, particularly in non-linear and dynamic systems like cryptocurrency markets.

This article presents a comprehensive study on the application of the Elman Neural Network model for predicting cryptocurrency prices. The ENN's unique architecture, featuring context units that store previous hidden layer states, enables it to capture temporal dependencies and patterns in time-series data, a critical aspect of financial markets. We detail the methodology for implementing the ENN model, including data preprocessing, feature selection, network training, and validation processes.

The Characteristics of Time Series Data

A time series refers to a series of data points, which can be either discrete or continuous, and are linked to specific time intervals. Time plays a crucial role in various natural phenomena, including fluctuations in air temperature, the heartbeat's rhythm, or the consistent pattern of waves hitting a shoreline. Similarly, time is a critical factor in numerous business activities, exemplified by the sales volume of a newly launched book during its initial month or the influx of customer inquiries at a call center during a long holiday weekend.

Time Series in Practice

Time is an inherent aspect that accompanies data collection. Analyzing time series involves dealing with data associated with time to forecast future trends. The intervals for these analyses can span various units of time, including years, seasons, months, days, hours, minutes, or seconds, depending on what is most appropriate for the context.

Business and Economic Time Series

Every year, business time series data is gathered worldwide to track economic performance, aid in managerial decision-making, and help policymakers choose the right strategies. An example of this can be seen in Figure 1, which illustrates the monthly cost of whole chicken from 2001 to 2015, typical of many business and economic time series. Throughout this timeframe, there's a noticeable upward trajectory in prices, reaching highs of more than 110 cents per pound, alongside significant fluctuations and intervals of heightened instability.

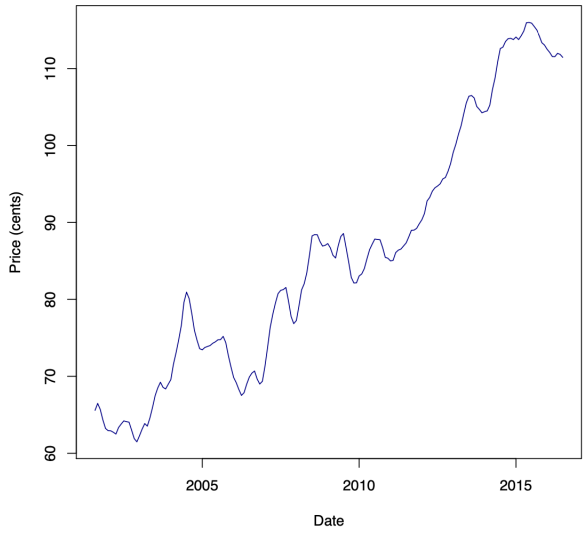

Figure 1: Monthly price of chicken

Health and Medical Time Series

In the medical and healthcare sectors, time series data are systematically captured. Metrics such as blood pressure, weight, heart rate, and many others are frequently monitored, sometimes in real-time, using diverse medical instruments. For instance, Figure 2 depicts an electrocardiogram (ECG), a tool that tracks the heart's electrical movements over time. This is typically represented by a wavy pattern featuring pronounced peaks and valleys.

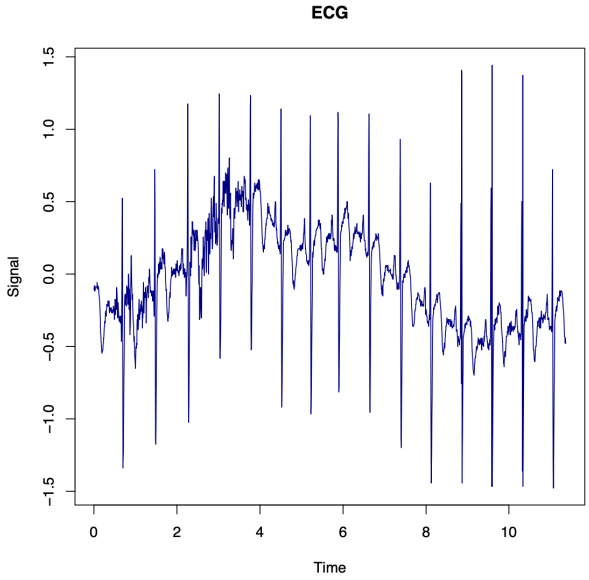

Figure 2: Electrocardiogram activity

Physical and Environmental Time Series

In the realms of physical and environmental sciences, a significant volume of time series data is amassed and examined. Long-term collection of environmental factors contributes to identifying climate patterns and forecasting future scenarios. For example, Figure 3 presents a chart of the average monthly temperatures in Nottingham, England. While the data doesn't seem to follow a clear trend, it clearly demonstrates a strong seasonal cycle, with the highest temperatures observed in the summer and the lowest in the winter.

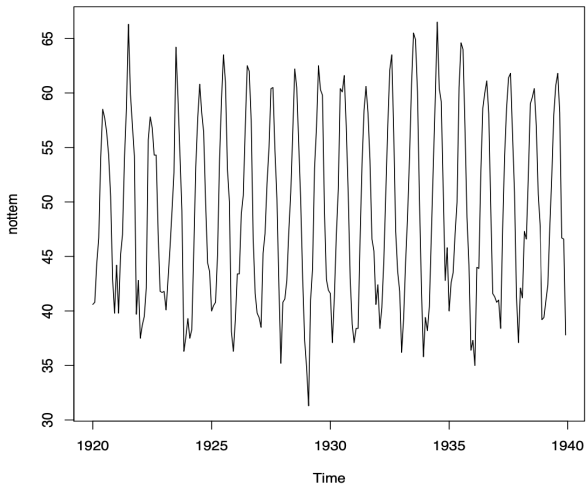

Figure 3: Average Monthly Temperatures at Nottingham 1920–1939

Time Series of Counts

Often, a time series consists of recorded counts over time. Take, for instance, how public health authorities in Germany tally the instances of E.coli infections within a specific area. As illustrated in Figure 4, the weekly E.coli case counts in North Rhine-Westphalia from January 2001 to May 2013 are displayed. Notably, there was a significant increase in cases toward the end of 2011. Given the lack of details regarding the reasons behind this surge, our time series analysis must be adaptable enough to accommodate such abrupt increases (or decreases).

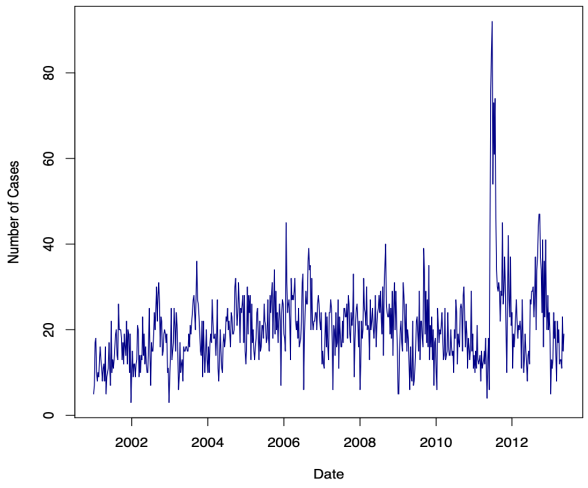

Figure 4: Weekly number of Cases of E. coli infection in the state of North Rhine- Westphalia

Discrete Time Series

Time series data often pertains to discrete occurrences. Sometimes, these occurrences are binary in nature, like the presence or absence of low-intensity seismic tremors in the Japanese

Islands. Discrete data might also encompass multiple possible outcomes. For instance, the tracking of eye movements involves recording a sequence of numbers that denote the position of gaze. Figure 5 charts the eye movement patterns of someone engaged in a conversation, revealing significant variations in the positions where the eyes rest over time.

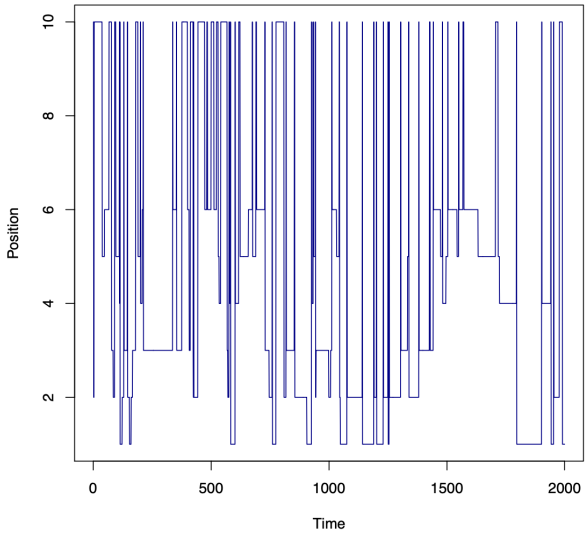

Figure 5: Time series plot of eye fixation positions

The Data Generating Mechanism

Time series data originate from what's termed a "data generating process. A fresh observation is produced at each time point. Hence, at a specific moment, denoted as time t , we might record a particular observation, referred to as y. This is represented as y_t. Moving to the subsequent time interval, $t+1$, another observation is noted, indicated as y_{t+1}.These intervals, represented by t, could span various durations such as seconds, hours, days, weeks, months, or years. For instance, the fluctuation in stock market prices is computed on a daily basis, while unemployment figures are typically updated monthly.

Time Series Decomposition

The process responsible for producing time series y_t is commonly conceptualized as being composed of three distinct elements: a seasonal element S_t, a trend element T_t, and a stochastic error or residual element ε_t. Assuming that the seasonal fluctuations relative to the trend do not change in proportion to the time series' magnitude, the relationship can be expressed as:

$$y_t = S_t + T_t + \varepsilon_t$$

This formulation is referred to as the additive model.

An Example

Upon revisiting Figure 6, it presents itself as a suitable instance for the additive model application. The `decompose` function in R can be employed to deduce the individual components as follows:

```
data("nottem", package = "datasets")
y <- nottem
y_dec <- decompose(y, type = "additive")
plot(y_dec)
```

The explanation for the R code snippets is as follows: The initial command retrieves the dataset, stored in the `nottem` variable within the `datasets` package. This dataset is then assigned to the variable `y`, which is subsequently inputted into the `decompose` function in the third command. By setting `type = "additive"`, we direct R to apply the additive model approach. The outcome of this decomposition is stored in the variable `y_dec`, which is then graphically represented using the `plot` function.

Figure 6 delineates both the original series (in the uppermost panel) and the decomposed elements (spanning the three lower panels). These components, when combined, replicate the initial time series illustrated in the upper panel. It is important to note that:

1. The trend component is exhibited as non-linear;
2. The seasonal element remains consistent, suggesting a uniform pattern across all the sampled years;
3. The residual component, depicted in the lowest panel, lacks any discernible trend or cyclical pattern.

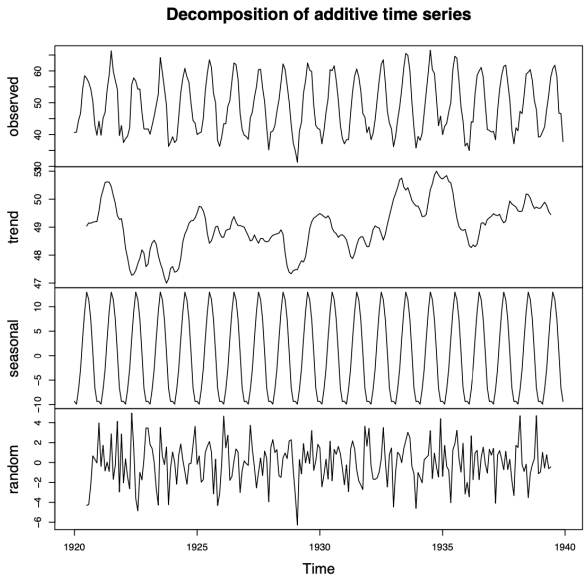

Figure 6: Additive model decomposition of the nottem dataframe

The multiplicative model

Should the seasonal fluctuations relative to the trend cycle change in proportion to the time series magnitude, one may employ a multiplicative decomposition approach, which is articulated as:

$$y_t = S_t \times T_t \times \varepsilon_t$$

The Benefit of Neural Networks for Time Series Modeling

In traditional time series analysis literature, the emphasis on discerning seasonality, understanding trend dynamics, and ensuring stationarity (uniform mean and variance) in data is paramount. However, the genesis of many real-world time series is often intricate or entirely obscured, rendering these series challenging to encapsulate with straightforward analytical formulations.

In practice, discerning and neutralizing trend and seasonal influences can be arduous. Defining trend and seasonality with precision is frequently fraught with difficulty. Additionally, even with accurate trend identification, it's crucial to model the trend appropriately, be it linear or nonlinear, localized or widespread. This precision is vital in conventional statistical methodologies, which necessitate predetermining a time series model, such as autoregressive models, Linear Dynamical Systems, or Hidden Markov Models.

A significant hurdle is the obscurity of the time series' foundational dynamics. Numerous treatises have been devoted to hypothesizing these dynamics, yet consensus on an optimal approach remains elusive.

In the field, applying classical statistical tools to time series analysis demands substantial expertise to choose a fitting model for the data at hand. The advantage of neural networks lies in their obviation of the need to define the exact relationship (linear, nonlinear, seasonal, or trend-based) between inputs and outputs. Neural networks, through their hidden layers, eliminate the prerequisite to predetermine the nature of the data generation process, given their capability to model highly intricate decision functions.

Should the data generation process be completely understood, forecasting future observations could be done with impeccable accuracy. Regrettably, this process is often enigmatic, leading us to forecast a new observation, denoted as $\hat{y}_t$. Our objective is to craft neural network-based forecasting models that minimize the discrepancy between the forecasted $\hat{y}_t$ and the actual

Feed-Forward Neural Networks

A multi-layer perceptron, commonly referred to as a feed-forward neural network, consists of numerous interconnected units termed neurons, as depicted in Figure 7. These neurons are typically organized into different layers, including at least an input layer, one or more hidden layers, and an output layer. The input layer's neurons correspond to the input features or variables, similar to the independent variables in linear regression. The output neurons are designed to match the number of outputs you aim to predict or categorize. The hidden layers' neurons mainly undertake the task of applying non-linear transformations to the input data.

At its core, a feed-forward neural network functions by forwarding the input data through the layers to generate a prediction, which can be either a continuous value for regression tasks or a categorical label for classification purposes.

An example of such a network structure is shown in Figure 7, where the network is configured to estimate a child's age. This network includes 2 input neurons, a single hidden layer comprising 3 neurons, and one output neuron. The input neurons receive specific attributes (such as Height and Weight), with each neuron dedicated to one attribute. The data then progresses to the hidden layer, where each neuron conducts a specific calculation before passing the results to the output neuron. The output neuron then aggregates these inputs to provide an age prediction. This process exemplifies the "feed-forward" nature of the network, where data moves in a forward direction through the network's layers.

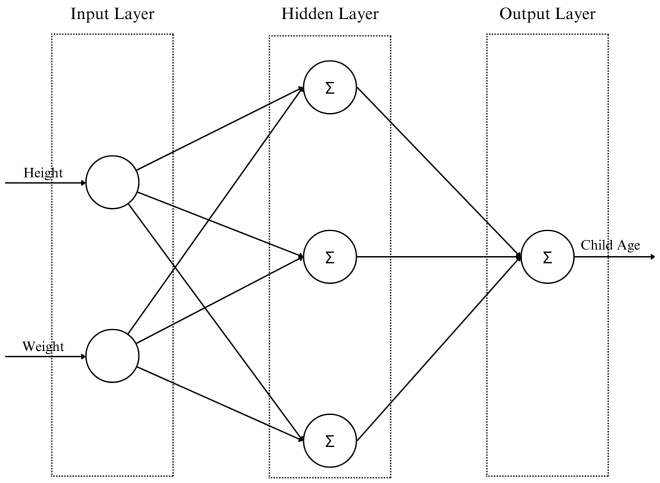

Figure 7: A basic neural network

The Role of Neuron

Central to the concept of an artificial neural network is the neuron, also known as a node or unit, which serves as the fundamental computational entity. Neurons in the input layer are the initial recipients of data, which they process using a specific mathematical function before passing it along to neurons in the hidden layer. Subsequently, this processed information is relayed to the neurons in the output layer for further handling.

As depicted in Figure 8, both biological and artificial neurons process information through what is known as an activation function. This function determines whether a neuron will activate ("fire") based on the input signal's intensity, mirroring the behavior of neurons in the brain. The output of this activation process is then weighted and forwarded to subsequent layer neurons, thereby establishing a system where neurons influence each other's activation through weighted connections. This mechanism adjusts the impact of one neuron on another in proportion to the significance of the information being processed.

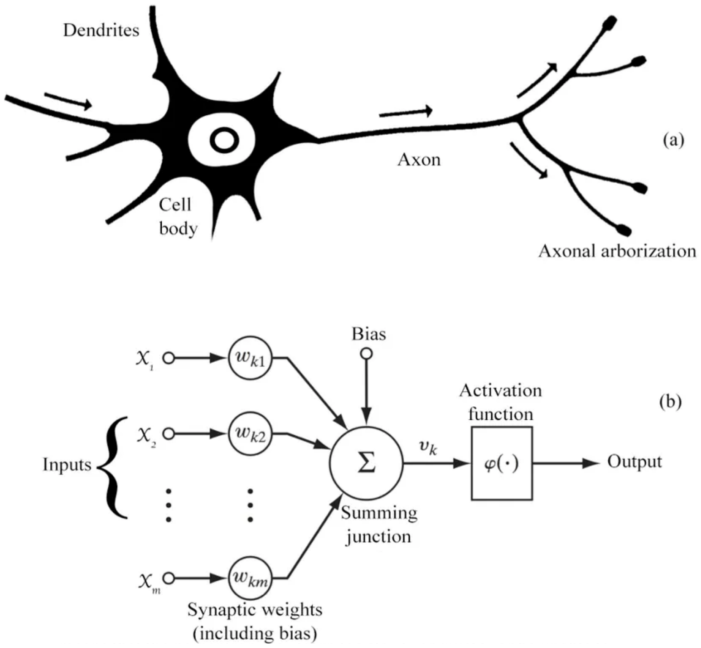

Figure 8: Biological and artificial neuron

Activation Functions

Every neuron is equipped with an activation function, as depicted in Figure 9, and a set threshold. This threshold represents the minimal input required to trigger the neuron's activation. Upon activation, the neuron processes the input through its activation function, and the resultant output is forwarded to the following neurons in the sequence.

The role of an activation function is to regulate the neuron's output, typically constraining it within a specific range such as 0 to 1 or -1 to 1. It's common practice to utilize the same activation function across all neurons within a given network. While nearly any nonlinear function can fulfill this role, the chosen function needs to be differentiable to be compatible with algorithms like stochastic gradient descent and having a bounded output is often advantageous.

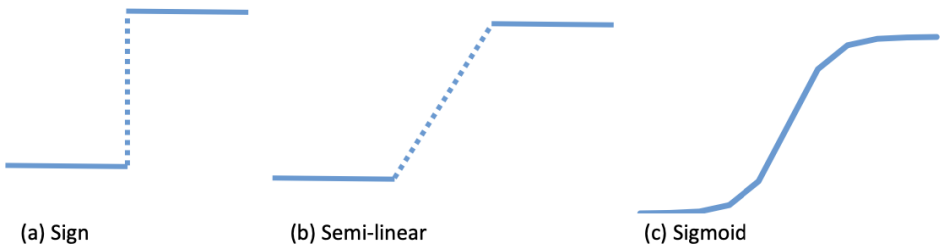

Figure 9: Three activation functions

The core task

The neuron's function involves calculating a weighted sum of its input signals and then processing this sum through an activation function before sending the result onwards to the subsequent layer. Thus, the data flows from the input layer to the initial hidden layer, where the hidden layer's neurons aggregate the data received from the input layer. Following this, the neurons in the output layer compile the aggregated and weighted data from the hidden layer to produce the final output.

Sigmoid Activation Function

The sigmoid, or logistic, function is a widely utilized activation function characterized by its "S"-shaped curve and differentiability. This function is depicted in Figure 10, with the parameter c set to a constant value of 1.5. The sigmoid function transforms any real number into a value within the 0 to 1 range. Specifically, it converts large negative inputs to 0 and large positive inputs to 1. The function is defined as:

$$f(u) = \frac{1}{1 + \exp(-cu)}$$

Its adoption is partly due to the function's output, which can be interpreted as the likelihood of the neuron being activated, or "firing."

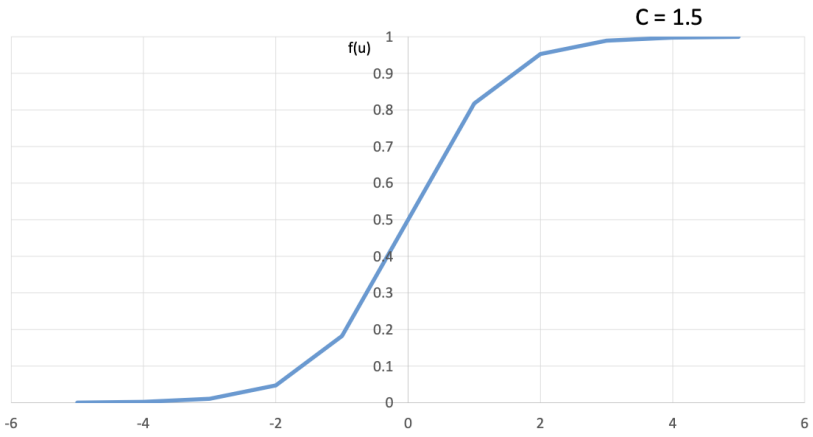

Figure 10: The sigmoid function with c = 1.5

Computational Cost

The sigmoid function is favored in fundamental neural network architectures due to its straightforward differentiability, which eases computational demands during the training phase. It is established that:

$$\frac{\partial f(u)}{\partial u} = f(u)(1 - f(u))$$

Hence, the derivative $\frac{\partial f(u)}{\partial u}$ equates to the logistic function $f(u)$ times $1 - f(u)$. This derivative is instrumental in optimizing the weight vectors through a technique referred to as stochastic gradient descent. A critical aspect to understand is that this property of the sigmoid function greatly simplifies the gradient calculations necessary for neural network training.

Tanh Activation Function

The Tanh activation function stands as a favored choice alongside the sigmoid function, defined by $f(u) = \tanh(cu)$. Mirroring the sigmoid's "s"-shaped curve, Tanh, however, spans a range from -1 to 1. Being essentially a rescaled version of the sigmoid, Tanh inherits many similar characteristics but offers a wider output spectrum ($[-1, 1]$ as opposed to $[0, 1]$ for the sigmoid), which can be advantageous in capturing complex non-linear relationships.

As illustrated in Figure 11, Tanh distinguishes itself from the sigmoid by its symmetry about the origin, mapping only zero-valued inputs to near-zero outputs, while strongly negative inputs yield negative outputs. These attributes contribute to reducing the likelihood of the training process becoming "stuck."

Although Tanh presents several advantages, its superiority over the sigmoid activation function is not guaranteed in practical applications. The most effective approach is to engage in empirical experimentation to determine the optimal choice.

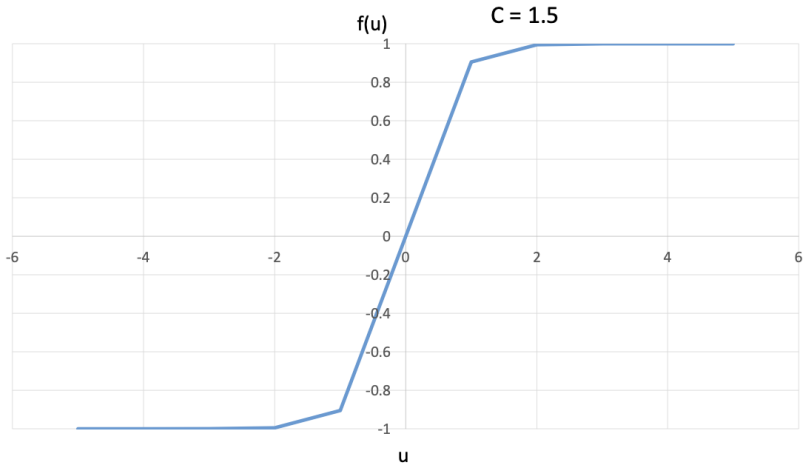

Figure 11: Hyperbolic Tangent Activation Function

The Working of the Neuron Simplified

As depicted in Figure 12, an individual neuron's operation can be described as follows: given a set of input attributes $\{x_1, \ldots, x_n\}$, each input x_j is multiplied by a corresponding weight w_{ij}, and the neuron computes the sum of these weighted inputs. This can be mathematically represented by:

$$f(u) = \sum_{i=1}^{n} w_{ij} x_j + b_j$$

Here, b_j represents the bias term, akin to the intercept in linear regression, which serves to adjust the activation function linearly, either "upwards" or "downwards". This capability to modulate the function is crucial for the adaptability required in machine learning models.

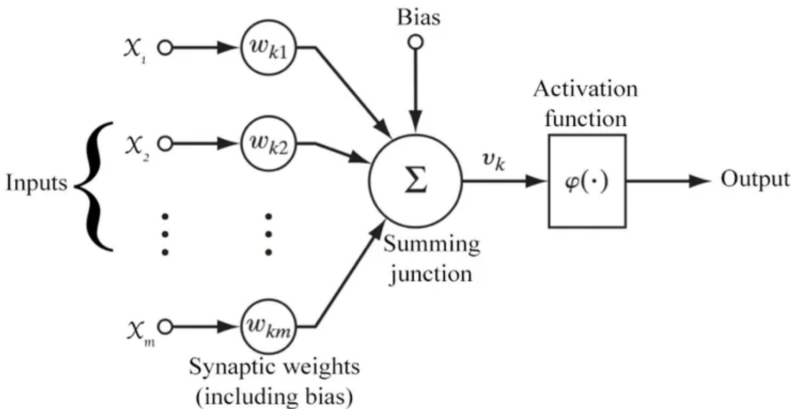

Figure 12: Artificial neuron

Network Size

The complexity of a neural network is frequently quantified by the count of parameters that need estimation. For instance, the network shown in Figure 7 comprises $2 \times 3 + 3 \times 1 = 9$ weight parameters and $3 + 1 = 4$ bias parameters, summing up to 13 trainable parameters in total. This quantity is considerably large when compared to the parameter set typically found in conventional statistical models.

How a Neural Network Learns

Given a particular dataset and a set of weights within a neural network, an error magnitude is determined via the error function, also referred to as the cost function. This metric assesses the neural network's accuracy against the expected output for the provided training data. The objective is to identify a configuration of weights that diminishes the discrepancy between the neural network's predictions and the actual target values.

Error propagation

In the standard learning approach for neural networks, errors are back-propagated from the output layer towards the input layer, allowing for incremental adjustments to the weights in order to reduce the overall error sum. This iterative training method is graphically represented in Figure 13, with each complete iteration of this procedure termed an epoch.

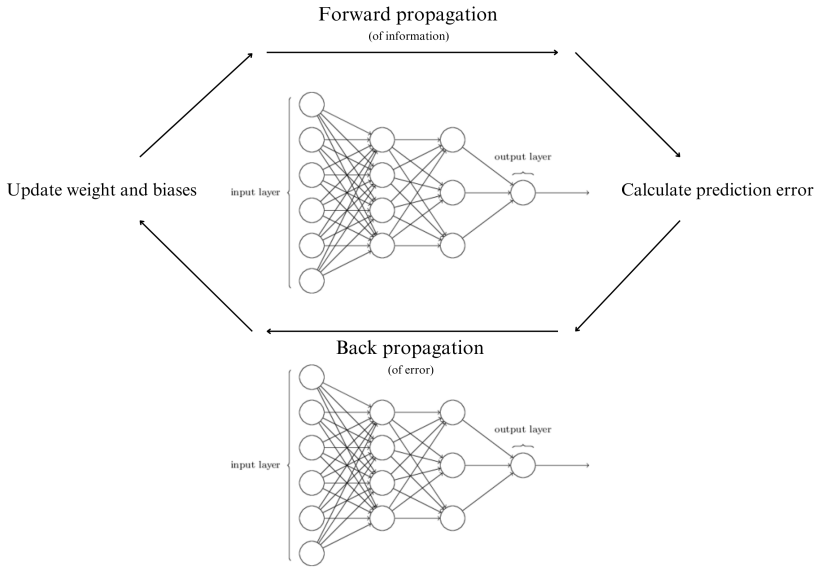

Figure 13: Neural network learning cycle

Backpropagation

Backpropagation stands as the preeminent algorithm for training neural networks. It begins by evaluating the network's output against the target value, assessing the deviation as an error metric. Subsequently, it distributes this error to each neuron in the preceding layer, iteratively continuing this process in reverse until it arrives at the input layer. Given that the error correction traverses in a reverse direction — from the output back through to the inputs — to refine the weights and biases, this method is aptly named backpropagation.

Step by Step Explanation

The fundamental process of neural network training involves computing the network's output error for a specific sample, then backpropagating this error to adjust the weights, aiming to diminish the error. Here's a paraphrased step-by-step breakdown:

Step 1 : Network Initialization – Starting values for the weights are assigned, typically chosen at random to kick off the network's training.

Step 2: Forward Propagation – Data is transmitted forward through the network, from the input to the hidden layers and then to the output layer, guided by the activation functions and the associated weights. Activation functions are generally sigmoidal, being bounded and differentiable.

Step 3: Error Calculation – The error is evaluated to see if it meets the predefined criteria for acceptability, or if the iterative process has run its course. If either of these conditions is fulfilled, training concludes.

Step 4: Backward Propagation – The discrepancy at the output layer guides weight modifications. This error is projected back through the network to calculate the gradient of error with respect to weight alterations.

Step 5: Weight Update – Weights are refined using these gradients to lessen the error. Each neuron's weights and biases are adjusted based on the activation function's derivative, the difference between the network's output and the actual target, and the neuron's outputs. This modification is how the network adapts and 'learns'.

Gradient Descent Clarified

Gradient descent is a widely employed optimization technique within neural networks. The goal is to identify the set of weights and biases that minimize the error function. Through gradient descent, these parameters are adjusted in an iterative fashion to reduce the overall error of the network. This involves continuously updating the weights in the opposite direction of the error function's gradient until a minimum is found. Essentially, this process can be visualized as traversing down the slope of the error function until settling in a low point. This concept is crudely demonstrated in Figure 14:

- When the gradient is negative, the corresponding weight is incremented, as seen on the left side of the figure.

- Conversely, a positive gradient leads to a reduction in the weight, illustrated on the right side of the figure.

- The magnitude of each update step is governed by the learning rate parameter.

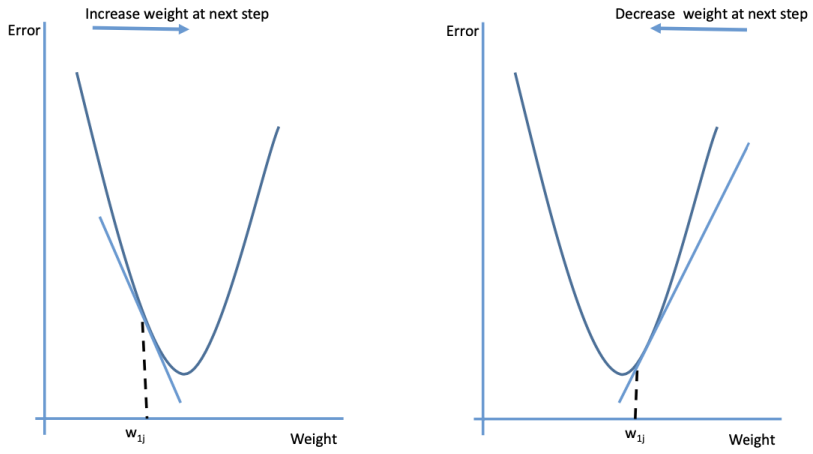

Figure 14: Basic idea of Stochastic Gradient minimization

Stochastic Gradient Descent

In conventional gradient descent, the gradient is calculated using the full dataset at every step. For sizable datasets, this can lead to repetitive calculations, particularly when gradients

of highly similar instances are recalculated before each update to the parameters.

Stochastic Gradient Descent (SGD) offers a different approach, estimating the true gradient by randomly selecting a single data point to update the parameters at each iteration. Consequently, it traces a zigzagging, stochastic route toward the minimum. This randomness, combined with reduced computational redundancy, allows SGD to often reach a solution more swiftly than the traditional method.

A particularly notable aspect of SGD is its theoretical guarantee to locate the global minimum for convex loss functions, provided the learning rate diminishes progressively over the course of training. With this condition, SGD exhibits a convergence pattern that mirrors that of traditional gradient descent.

Exploring the Error Surface

Neural network architectures consist of numerous weights that need to be finely tuned to achieve the best possible performance. Due to the highly nonlinear relationship between the inputs and outputs, optimizing these models is a sophisticated task. The complexity of reaching a global optimum, while evading local minima, stems from the error function typically being neither convex nor concave. This indicates that the matrix comprising all the second-order partial derivatives, commonly referred to as the Hessian matrix, does not consistently maintain positive or negative semi-definiteness. In practical terms, this characteristic implies that neural networks have the potential to become ensnared in local minima due to the variable topology of the error landscape.

A simple example

To draw a comparison to functions of a single variable, observe that $\sin(x)$ does not exhibit convexity or concavity, possessing an infinite number of maxima and minima, as illustrated in Figure 15 (top panel). On the contrary, x^2 features a single minimum, and $-x^2$ a lone maximum, shown in Figure 15 (bottom panel). The implication of this for neural networks is that the form of the error landscape can cause them to be entrapped in local minima.

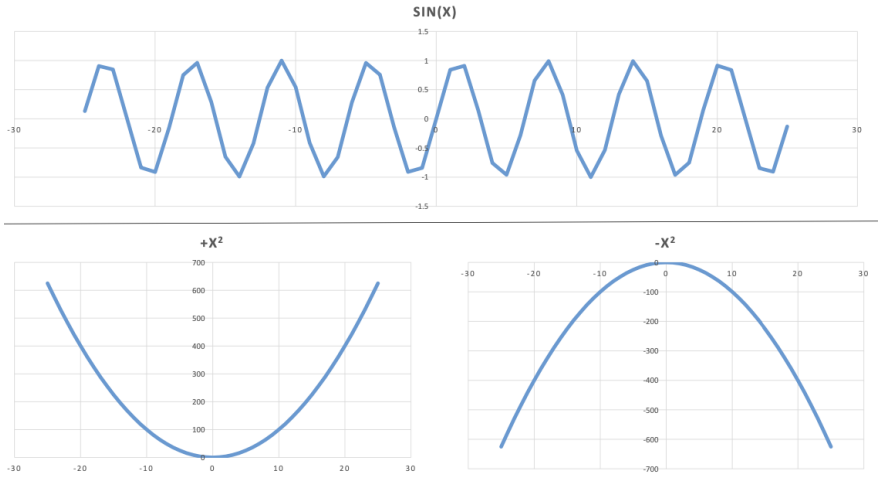

Figure 15: One variable functions $\sin(x)$, x^2 and $-x^2$

Visualizing the error surface

If you were to graph the error of a neural network against its weights, the resulting surface would probably appear extremely irregular, dotted with numerous local minima. As depicted in Figure 16, this landscape might exhibit a multitude of peaks and troughs. The surface could be steeply curved in certain areas while remaining flat in others, adding significant complexity to the optimization task.

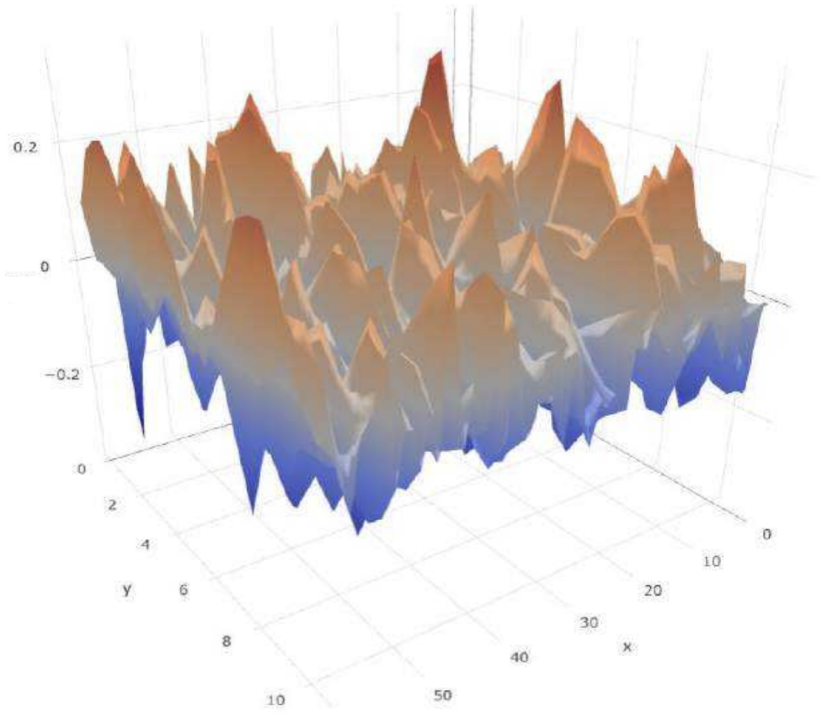

Figure 16: Complex error surface of a typical optimization problem

Choosing a Learning Rate

Our next step involves determining the learning rate, which influences the magnitude of the steps that the gradient descent algorithm uses to approach the minimum, as illustrated in Figure 17. A larger learning rate can accelerate the learning process, whereas a smaller one results in a slower journey to the optimal value.

One might question why not to always opt for a high learning rate. However, as demonstrated in Figure 18, an excessively high learning rate can lead the network to overshoot the global minimum, resulting in suboptimal learning or failure to converge. The process of setting the learning rate typically requires iterative adjustments and is often achieved through a method of trial and error to find the maximum feasible value.

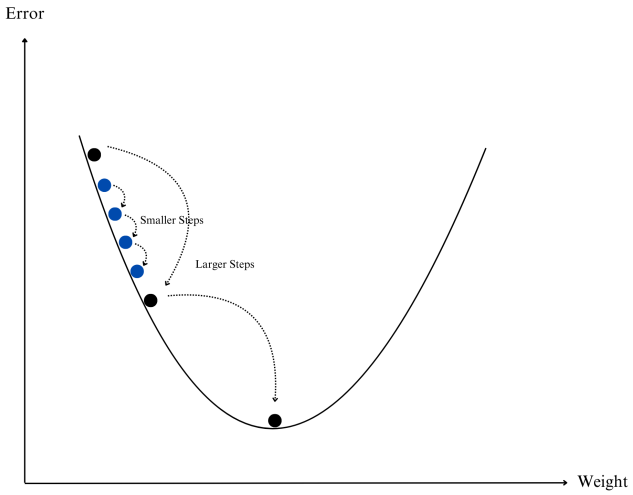

Figure 17: Optimization with small and large learning rates

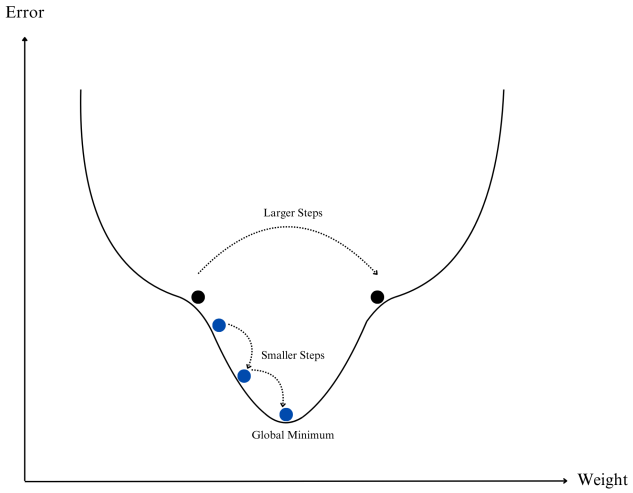

Figure 18: Error surface with small and large learning rate

A Complete Intuitive Guide to Momentum

The training of a neural network is an iterative process that continues until the error is sufficiently minimized. The learning rate governs the size of each step in this process. Larger steps can decrease training time but may also increase the risk of becoming stuck in local minima rather than finding the global minimum.

Introducing a momentum term is another strategy to aid the network in escaping local minima. This term, which ranges from 0 to 1, incorporates a fraction of the previous weight update into the current update. As depicted in Figure 19, a high momentum value, such as 0.9, can accelerate training and help prevent the network from getting trapped in local minima. However, setting the momentum too high might lead to overshooting the global minimum, especially if combined with a high learning rate. Conversely, a very low momentum might not provide enough impetus for the network to overcome local minima.

Determining the ideal momentum value can be challenging, and it is usually beneficial to try various settings. A common approach is to lower the learning rate when employing a higher momentum. This adjustment can help balance the acceleration of the learning process with the control needed to achieve stable convergence.

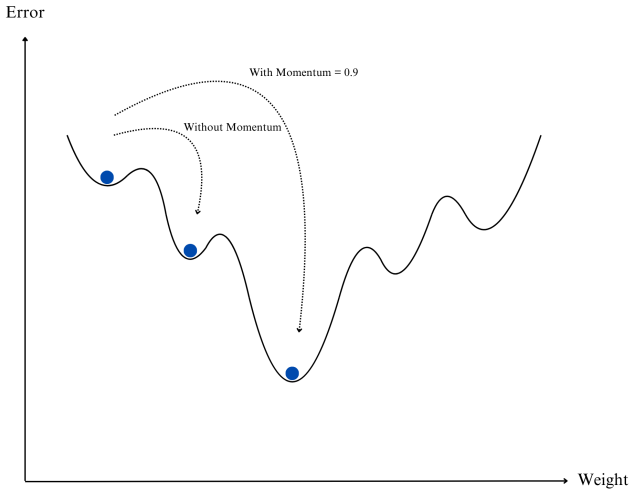

Figure 19: Benefit of Momentum

Deep Neural Networks in a Nutshell

A deep neural network (DNN) is structured with an input layer, an output layer, and multiple hidden layers positioned between them. As illustrated in Figure 20, these hidden layers connect to the input layer, where they integrate and assign weights to the input values, subsequently producing a real-valued number that is forwarded to the output layer. The output layer utilizes the processed values from the hidden layers to make classification or

prediction decisions.

Deep multi-layer neural networks incorporate several layers of nonlinearity, enabling them to efficiently represent highly complex and variable functions. These networks excel at recognizing intricate patterns within data, contributing to advancements in fields such as computer vision, natural language processing, and handling unstructured data challenges. Similar to single-layer neural networks, during the training phase, the weights of the connections between layers are adjusted to align the output as closely as possible with the desired target.

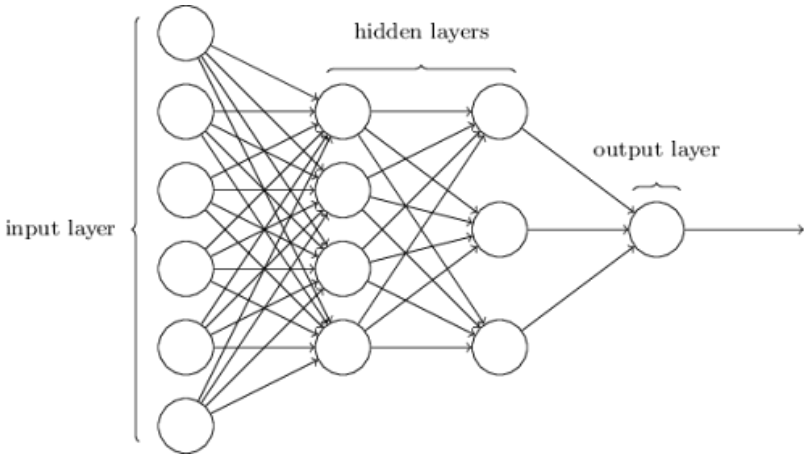

Figure 20: Feed forward neural network with 2 hidden layers

The Role of Batching

The conventional backpropagation algorithm determines the adjustment in neuron weights—often referred to as deltas or gradients—for every neuron across all layers in a deep neural network during each epoch. These deltas are essentially adjustments based on calculus derivatives, aimed at reducing the discrepancy between the network's actual output and the expected output.

In large-scale neural networks, possibly featuring millions of weights, the task of calculating deltas for all these weights can be overwhelmingly time-consuming. Consider the enormity of computing gradients for millions of weights; this process can significantly extend the time required for the network to reach an acceptable solution, potentially making it impractical for certain applications.

One strategy to enhance the computational speed is batching, which computes gradients for multiple training examples at once, rather than individually as in traditional stochastic gradient descent. For example, if you have a batch size of 500 and a total of 1000 training examples, only two iterations are necessary to complete one epoch, illustrating the efficiency gains batching can offer.

Understanding Recurrent Neural Networks

Contrary to the feed-forward neural networks highlighted in chapter 2, Recurrent Neural Networks (RNNs) feature hidden states that extend through time. This temporal distribution

enables them to retain extensive information about past events. Similar to conventional feed-forward neural networks, the non-linear behavior exhibited by their nodes allows RNNs to effectively model complex dynamics in time series data.

The Difference between Dynamic and Static Neural Networks

Neural networks are categorized into two types: dynamic and static. Static neural networks generate outputs solely based on the input via feed-forward connections. In these networks, such as the basic feedforward neural networks described in chapter 2, information flows uni-directionally from the input to the output without any feedback mechanisms. These models discussed previously are all instances of static neural networks.

Conversely, in dynamic neural networks, the output is influenced not only by the current input but also by past inputs, outputs, and hidden states of the network. Recurrent Neural Networks (RNNs) exemplify this type of dynamic network, where the past significantly affects the network's behavior and output.

A Visual Explanation of Recurrent Neural Networks

Figure 21 depicts two examples of simple recurrent neural networks. The fundamental concept behind these networks is that recurrent connections maintain a memory of past inputs within the network's internal state, which in turn affects the output.

At each time step, the network executes several operations:

1. Processes the input attributes x_t;
2. Updates its hidden state h_t using activation functions;
3. Utilizes the updated state to predict the output y_t.

These steps resemble those in a feed-forward neural network. However, in recurrent neural networks, the information from a delay unit is recycled back into the hidden units, providing them with additional input for processing.

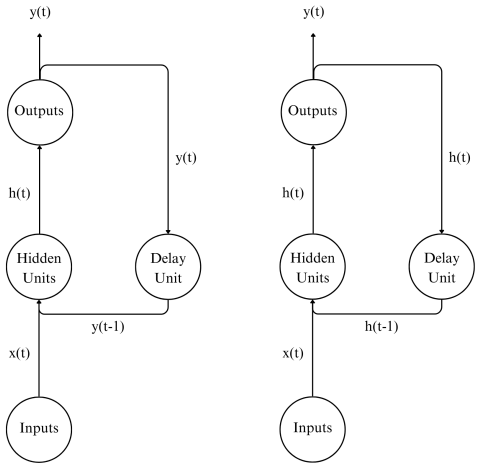

Figure 21: Two Simple Recurrent Neural Network Structures

The Role of the Delay Units

The delay unit is crucial for providing the network with short-term memory capabilities. It achieves this by retaining the activation values from the hidden layers or outputs from previous time steps and reintroducing them into the network at subsequent time steps. Thus, the RNN possesses a form of "memory" that holds details of past calculations conducted by the hidden units.

Time-series data are characterized by patterns that unfold over time, embedding information about the processes that generate the data. RNNs leverage this temporal arrangement due to the persistence characteristic of delay units. This feature of "short-term" memory is essential for enabling RNNs to learn from and generalize across sequences of inputs, capturing and utilizing the temporal dynamics inherent in the data.

Parameter Sharing Clarified

According to Figure 22, a Recurrent Neural Network (RNN) consists of several instances of the same network, each passing information to the next. This configuration allows the RNN to maintain consistent parameters throughout all time steps, as it executes the same operation at each step, albeit with varying inputs. This uniformity significantly lowers the number of parameters an RNN needs to learn compared to a conventional deep neural network, which typically requires a unique set of weights and biases for each layer.

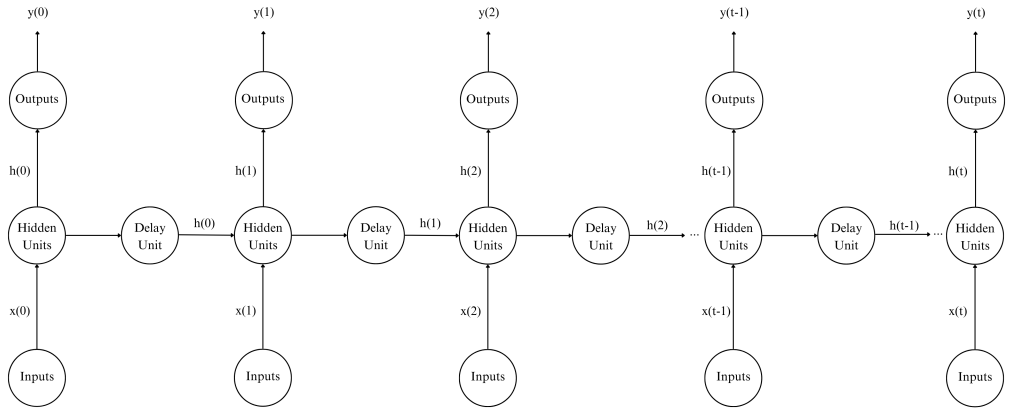

Figure 22: Unfolding a RNN

Backpropagation Through Time

It appears that with certain modifications, any feed-forward neural network can be adapted to train using the backpropagation algorithm. For a Recurrent Neural Network (RNN), this modified method is called Backpropagation Through Time.

The principle behind this is to "unfold" the RNN across time. While this might sound complex, the idea is straightforward. Referring to Figure 22, it shows an unfolded version of the network. By unfolding, we simply lay out the network across all the time steps being analyzed.

This unfolding essentially transforms the recurrent loop into a sequence that resembles a feed-forward neural network. In this unfolded state, the RNN looks like a deep neural network, where each layer represents a time step from the original RNN.

This restructured feed-forward network can then be trained using traditional backpropagation. The process of calculating the error derivatives relative to the weights simplifies to the same process used in layers of a feed-forward network. Once training is complete, the expanded feed-forward structure can be "folded" back into its original RNN form.

Understanding Elman Neural Networks

Elman neural networks are specifically crafted to recognize and learn patterns that evolve over time or are sequential in nature. These networks typically consist of an input layer, a context layer (also referred to as a recurrent or delay layer, as depicted in Figure 6.1), a hidden layer, and an output layer. Each of these layers includes one or more neurons that transmit information to the next layer through the calculation of a nonlinear function applied to their weighted sum of inputs.

In the structure of an Elman neural network, the context layer contains a number of neurons equal to that in the hidden layer, and each neuron in the context layer is fully connected to every neuron in the hidden layer.

Like traditional feedforward neural networks, the connection strength between the neurons in an Elman neural network is governed by weights. Initially, these weights are set randomly and are fine-tuned during the training process to optimize network performance.

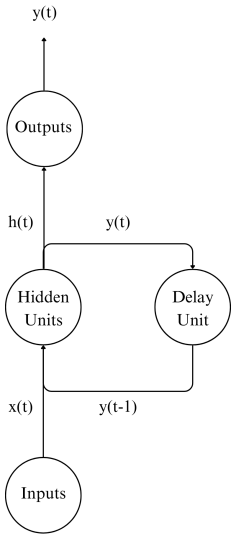

Figure 23: Elman Neural Network

Role of Memory

Memory in Elman neural networks is facilitated by the delay (context) units, which receive inputs from the neurons of the hidden layer. In the standard configuration of these networks, the recurrent connection weights from the hidden layer to the delay units are preset to 1. Consequently, this arrangement ensures that the delay units consistently retain a copy of the hidden units' prior values.

Predicting Cryptocurrency Prices by using Elman Neural Network Model

Step 1 – Collecting Data

The R library 'binacer' enables the retrieval and visualization of Ethereum stock data from Binance. The process involves loading and plotting the data for the ETH/USDT trading pair as follows:

```
eth_usdt <- binance_klines('ETHUSDT',
interval = '15m',
start_time = '2023-09-27',
end_time = '2023-10-02')
eth_usdt <- eth_usdt[, c("open_time", "open", "high", "low", "close")]
eth_usdt_ts <- as.ts(eth_usdt$open)
plot(eth_usdt_ts)
```

This code snippet first retrieves Ethereum price data within a specific date range and at 15-minute intervals. It then selects relevant columns such as open, high, low, and close prices. Subsequently, it converts the opening prices into a time series format and plots this time series.

The observed time series demonstrates a positive long-term trend, characterized by significant volatility which intensifies as the overall price level rises. There is also evident strong seasonality within the data, likely contributing substantially to its fluctuating pattern.

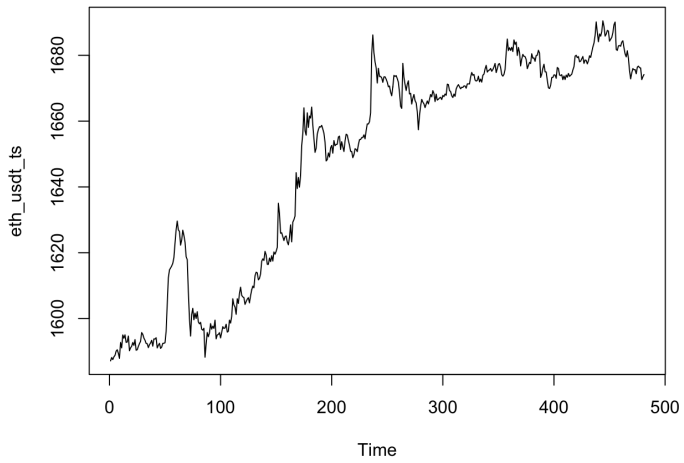

Figure 24: Ether Price

Step 2 – Exploring and Preparing the Data

Observing in Figure 24, it is evident that there is a seasonal pattern that tends to align with the overall trend, indicating the suitability of a multiplicative model:

$$\text{price}_t = \text{trend}_t \times \text{seasonal}_t \times \text{random}_t$$

where:

- price_t represents the ether price at time t,
- trend_t denotes the trend component,
- seasonal_t is the seasonal factor,
- and random_t is the random error component.

To analyze these components, we utilize the decompose function. Instead of using the actual price levels, we choose to work with the natural logarithm of the prices:

```
y <- ts(log(eth_usdt_ts), frequency = 12)
decomp <- decompose(y, type = "multiplicative")
plot(decomp)
```

The `R` object `y` stores the logarithmic prices, and `decomp` represents the decomposed series. Figure 25 illustrates the estimated components.

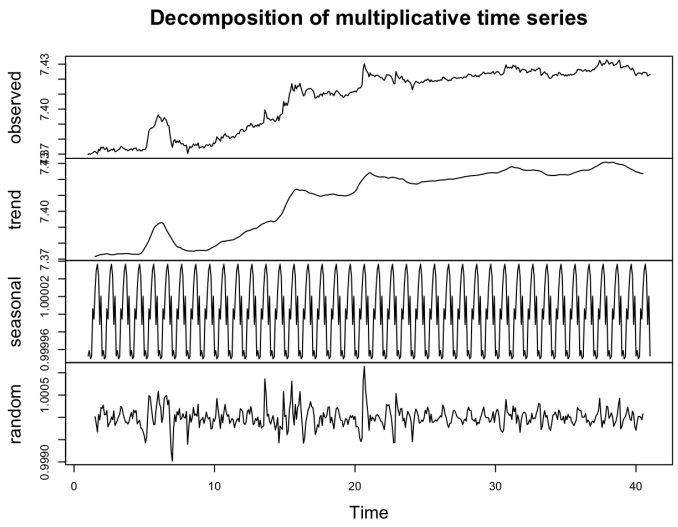

Figure 25: Decomposition of Ether prices

Trend component

Let's examine the trend component more closely:

```
plot(decomp$trend, ylab = "Trend")
```

Figure 26 shows that the trend is not linear. It includes times of quick growth followed by more stable periods.

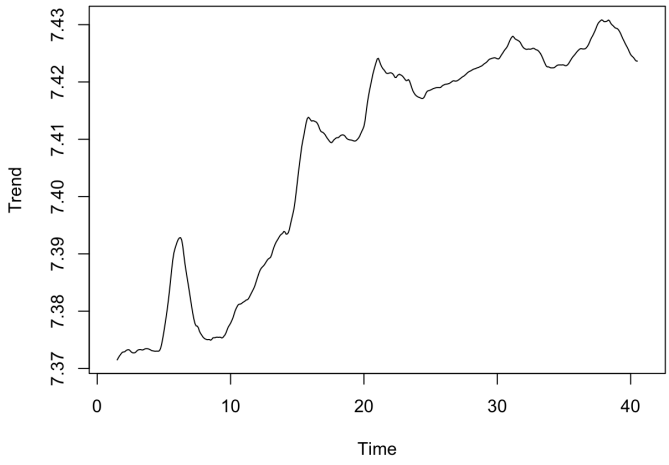

Figure 26: Trend component of Ether Prices

Error component

It is often helpful to look at the distribution of the error component. We apply the density function for a rapid visual overview:

```
random <- newdata <- na.omit(decomp$random)
plot(density(random))
```

Figure 27 suggests that the error component roughly matches the typical bell-shaped curve seen in normally distributed random variables.

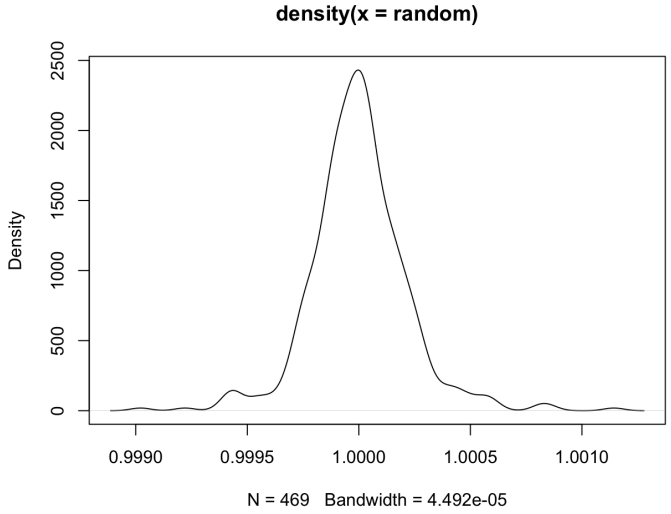

Figure 27: Density plot of the random component of Ether Prices

We can investigate further with a Normal Q-Q plot by using the function qqnorm:

```
qqnorm(random, pch = 1, frame = TRUE)
qqline(random, col = "steelblue", lwd = 2)
```

If the random component was created by a normal distribution, the points on the Q-Q plot would align perfectly with the 45-degree line. Figure 28 shows that the `random` component is approximately normally distributed.

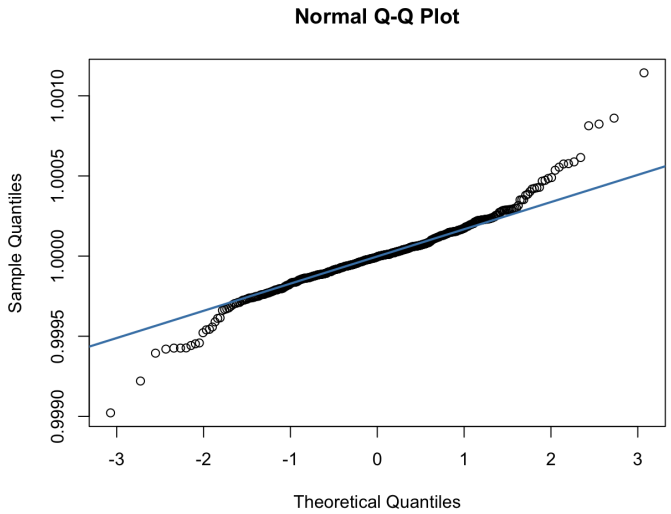

Figure 28: Q-Q plot of the random component of Ether Prices

Inspecting Autocorrelation

Next, let's examine the autocorrelations and partial autocorrelations:

```
qqnorm(random, pch = 1, frame = TRUE)
qqline(random, col = "steelblue", lwd = 2)
```

According to Figure 29, the correlation between prices appears to continue over time. Figure 30 displays the partial autocorrelations, suggesting that previous prices are somewhat related to the current price.

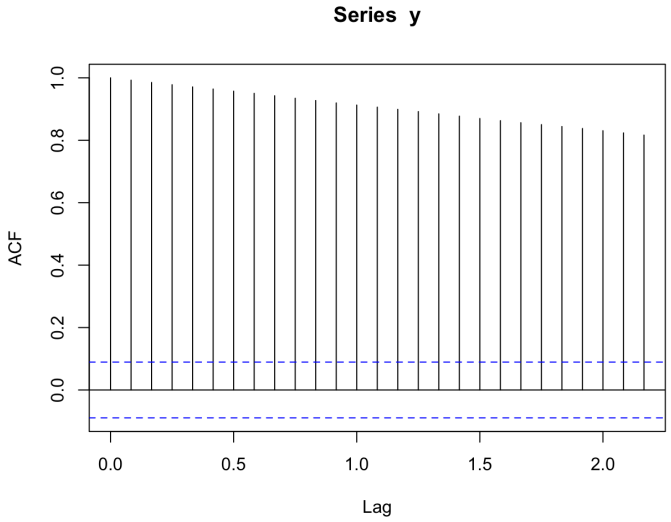

Figure 29: Auto correlations of Ether Prices

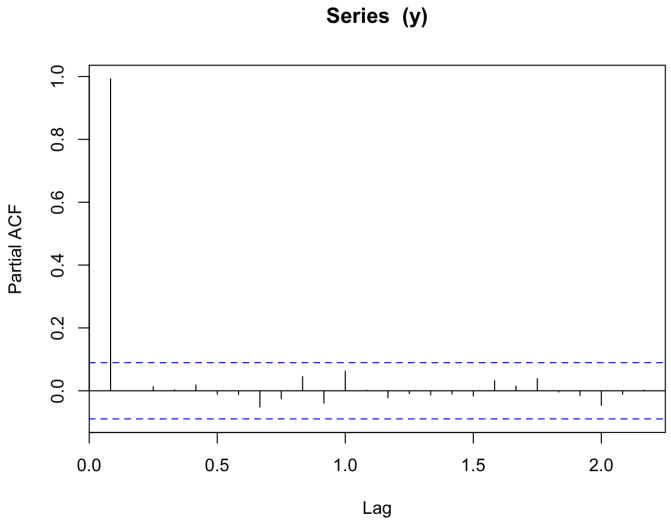

Figure 30: Partial autocorrelations of Ether Prices

Choosing the number of time lagged variables

Considering the data's monthly frequency, we'll create 12 time-lagged variables as input features. We start by normalizing the data with a `range_data` custom function:

```
range_data <- function(x) { (x - min(x)) / (max(x) - min(x))}
y <- range_data(eth_usdt_ts)
min_data <- min(eth_usdt_ts)
max_data <- max(eth_usdt_ts)
```

Note that we are using the actual price levels here, not the logarithmic values.
Next, we employ the `quantmod` package to generate the lagged features:

```
y <- as.zoo(y)

x1 <- Lag(y, k = 1)
x2 <- Lag(y, k = 2)
x3 <- Lag(y, k = 3)
x4 <- Lag(y, k = 4)
x5 <- Lag(y, k = 5)
x6 <- Lag(y, k = 6)
x7 <- Lag(y, k = 7)
x8 <- Lag(y, k = 8)
x9 <- Lag(y, k = 9)
x10 <- Lag(y, k = 10)
x11 <- Lag(y, k = 11)
x12 <- Lag(y, k = 12)

x <- cbind(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, y)
x <- cbind(x, y)
x <- x[-(1:12),]
```

The variable `x` holds the lagged data. The final line of code above eliminates any missing values resulting from the `Lag` function.

Number of observations

We should have enough data remaining to conduct a significant analysis, so let's verify

```
n = nrow(x)
```

We have 469 observation. This is convenient for our analysis.

Step 3 – Training a Model on the Data

We select 300 observations at random to form the training set, and we keep the rest for the test set:

```
set.seed(2018)
n_train <- 300
train <- sample(1:n, n_train, FALSE)
inputs <- x[, 2:13]
outputs <- x[, 1]
```

The `sample` function randomly chooses the training observations without repeating any. The target variable is saved in the `outputs` object, while the features are stored in the `inputs` object in R.

Using the package RSNNS

The `RSNNS` package includes the function `elman`, which is used to estimate an Elman neural network. We construct a model with two hidden layers, each having 2 nodes, and run the model for 1000 iterations:

```
num_input_features <- ncol(inputs)
fit <- elman(inputs, outputs, size = c(num_input_features, 2), maxit = 1000)
```

The `size` argument specifies the number of nodes in each hidden layer, while `maxit` sets the number of iterations. The R object `fit` holds the trained model.

Step 4 – Evaluating Model Performance

The function `plotIterativeError` graphs the error progression across the sample. A quick decrease in error suggests that the model is effectively learning from the data:

```
plotIterativeError(fit)
```

Figure 31 shows that the error decreases sharply during the first 150 iterations. After that, it continues to drop but at a slower pace until it reaches the 1000th iteration.

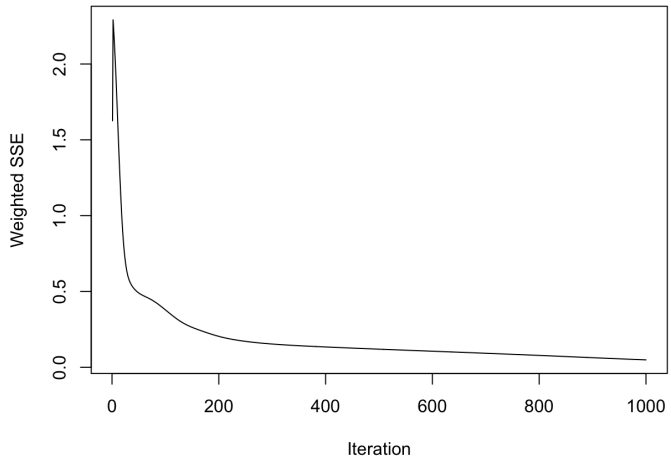

Figure 31: Error by iteration for Elman neural network

The plotRegressionError function

The `plotRegressionError` function is employed to display the relationship between the actual target values and the predicted values:

```
plotRegressionError(outputs, fit$fitted.values)
```

In Figure 32, the actual observations (target values) are plotted on the x-axis, and the predicted values (fitted values) from the training set are on the y-axis. A perfect match between these would appear as a straight line starting at zero with a slope of one, represented by the solid black line in the diagram. A linear approximation of the actual data is depicted by the lighter red line.

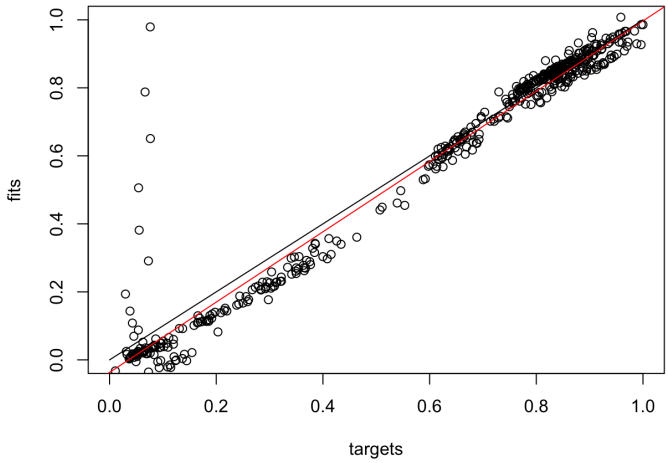

Figure 32: Fitted and actual values using the plotRegressionError function

Step 5 – Assessing Test Set Performance

The `predict` function is utilized for making predictions. It requires the fitted model (here, `fit`) and the sample observations. To evaluate the model's performance on the test set, we input these values along with `fit` into the `predict` function:

```
pred <- predict(fit, inputs[-train])
```

Visual inspection

We need to revert the data to its original scale and visually examine it. We use our custom `unscale_data` function for this task, followed by plotting:

```
unscale_data <- function(x, max_x, min_x) {
(x * (max_x - min_x) + min_x)}
output_actual <- unscale_data(outputs[-train], max_data, min_data)
output_actual <- as.matrix(output_actual)
rownames(output_actual) <- 1:length(output_actual)
output_pred <- unscale_data(pred, max_data, min_data)
result <- cbind(as.ts(output_actual), as.ts(output_pred))
plot(result[,1], type = "l", col = "blue", ylab = "Price")
lines(result[,2], col = "red")
legend("topleft", legend = c("Actual", "Predicted"), col = c("blue", "red"), lty = 1)
```

Overall, the Elman neural network manages to capture both the trend and significant aspects of the dynamics in the ether price time series. These results were obtained without extensive feature engineering or data manipulation, which is often necessary with traditional statistical models.

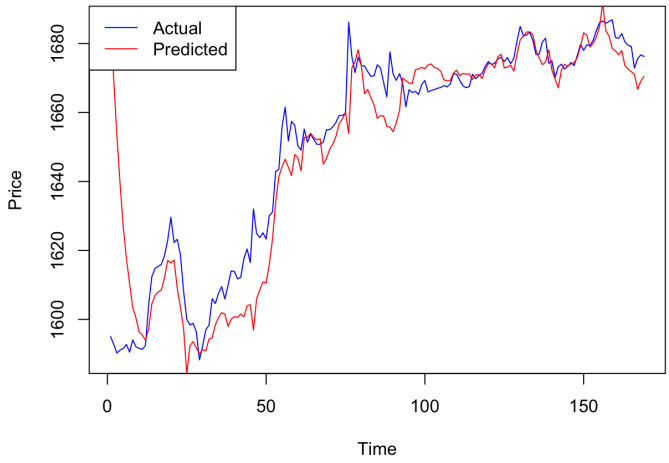

Figure 33: Actual and predicted values for the ether price prediction of Elman model