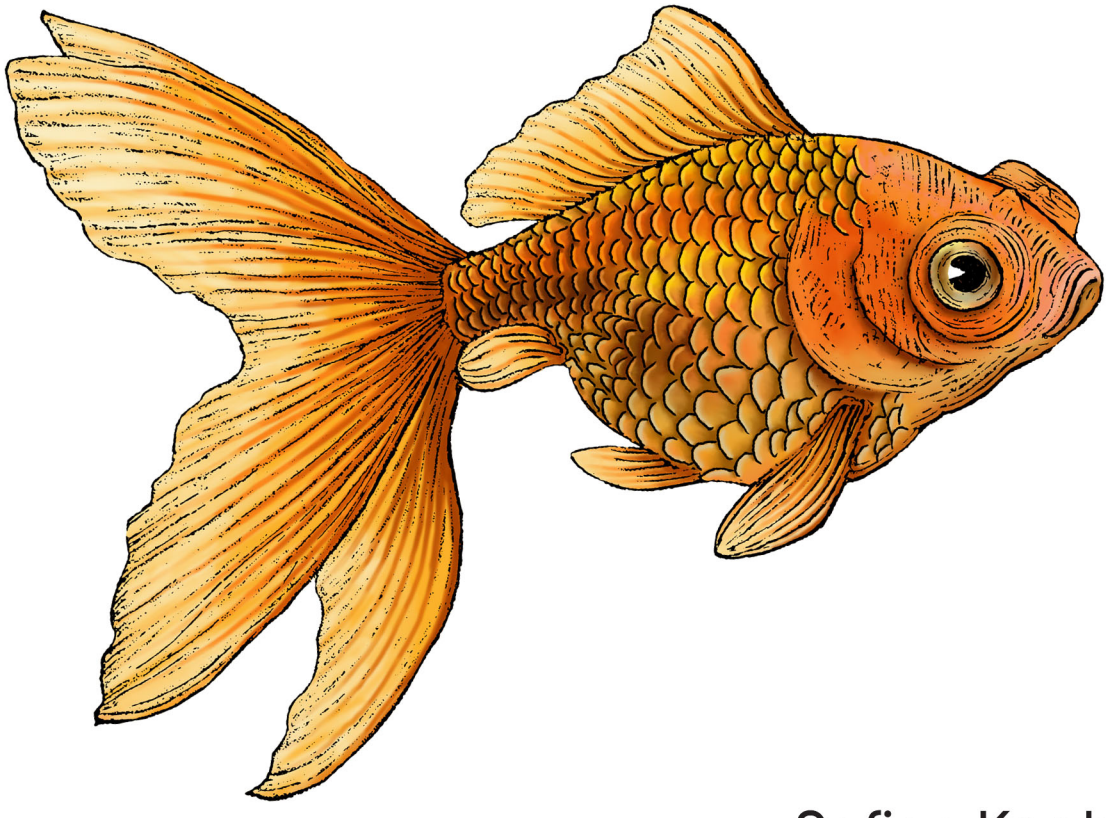
O'REILLY®

Deep Learning for Finance

Creating Machine & Deep Learning Models for Trading in Python



Sofien Kaabar

Praise for *Deep Learning for Finance*

As the scientific director of a leading program in market finance for over 10 years, I can testify to the immense quality of this book. *Deep Learning for Finance* is a magisterial work that stands as a landmark in the field of quantitative trading, data science, and financial algorithms. The author's profound knowledge and deep insights are evident throughout the book, which is written with clarity and precision. It will undoubtedly become a reference in this specialized field in which the inherent complexity of the subject is rarely well served in disclosure books. This one is an exception, striking the perfect balance between clarity and precision without becoming oversimplistic or overcomplex. It is an essential read for anyone interested in the cutting edge of quantitative trading/finance, both for master's degree students in finance and for practitioners.

—Amaury Goguel, Head of MSc Financial Markets & Investments, SKEMA Business School, Paris

This is the book I wish I had read when I started developing ML trading algorithms as a quantitative investment strategist.

—Ning Wang, Quantitative Investment Structurer, Barclays

Sofien is a master, providing the right balance of detail and autonomy, allowing readers to connect the dots themselves.

—Timothy Kipper, Head of Business Development, Coperniq.io

Deep Learning for Finance

Creating Machine and Deep Learning Models for Trading in Python

Sofien Kaabar

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Deep Learning for Finance
by Sofien Kaabar

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Michelle Smith
Development Editor: Corbin Collins
Production Editor: Elizabeth Faerm
Copyeditor: Audrey Doyle
Proofreader: Piper Editorial Consulting, LLC

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

January 2024: First Edition

Revision History for the First Edition
2024-01-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098148393* for release details.

978-1-098-14839-3

[LSI]

Table of Contents

Preface

Learning never exhausts the mind.
—Leonardo da Vinci

Machine learning and deep learning have completely changed the finance industry in recent years. The different learning models are well suited to a world where data is abundant and continuous. Data is the new gold, and its value keeps rising as proper analyses lead to key business decisions, which are the driver of economic shifts.

The rise of quantitative funds is living proof that the world of data science has much to offer to the trading world. After fundamental and technical traders, a new breed of leaders of the universe is emerging. These are the quantitative traders who rely on machine-based algorithms with extremely complex operations that seek to forecast and outperform the markets.

This book covers in detail the subject of *deep learning for finance*.

Why This Book?

I have spent my career researching trading strategies, techniques, and all things related to the financial world. Through the years, I have become familiar with a few algorithmic models that have the potential of adding value to the trading framework. In this book, I discuss different learning models and their applications in the trading world, with a focus on deep learning and neural networks. My main aim is to cover them in such a way that everyone understands how they function.

Machines can perform operations and detection better than humans for many reasons, one of which is their objectivity. This means that one of the key skills you will learn is how to use Python to create the algorithms required to do such operations.

As mentioned, my objective is to provide a comprehensive introduction to the use of deep learning in finance. I do this by discussing a wide range of topics, including data science, trading, machine and deep learning models, and reinforcement learning applications for trading.

The book begins with an overview of the field of data science and its role in the finance world. It then delves into the knowledge requirements, such as statistics, math, and Python, before focusing on how to use machine and deep learning in trading strategies.

Who Should Read It?

This book is intended for a wide audience, including professionals and academics in finance, data scientists, quantitative traders, and students of finance of any level. It provides a thorough introduction to the use of machine and deep learning in time series prediction, and it is an essential resource for anyone who wants to understand and apply these powerful techniques.

The book assumes you have basic background knowledge in both Python programming (professional Python users will find the code very straightforward) and financial trading. I take a clear and simple approach that focuses on the key concepts so that you understand the purpose of every idea.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/sofienkaabar/deep-learning-for-finance*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution, which usually includes the title, author, publisher, and ISBN. For example: "*Deep Learning for Finance* by Sofien Kaabar (O'Reilly). Copyright 2024 Sofien Kaabar, 978-1-098-14839-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/deep-learning-for-finance*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Nothing would be the same without the support of my parents, which is why I can't help but acknowledge their direct and indirect impact on the book.

I would also like to acknowledge the debt I owe to the editors, Michelle Smith and Corbin Collins, as well as to the production editor, Elizabeth Faerm, for their continued support, the amazing job they do, and their patience. Similarly, I would like to thank every person at O'Reilly who was involved in the production of this book.

Additionally, my special thanks go to the great technical reviewers for their immense contributions. They had a sizable impact on making this book readable, useful, and straightforward. I could not ask for better people to review my book.

Finally, I am deeply grateful to you, the reader, for investing your time into reading my work and for placing your trust in my research. I hope you find it useful.

CHAPTER 1

Deep Learning for Time Series Prediction I

Deep learning is a slightly more complex and more detailed field than machine learning. Machine learning and deep learning both fall under the umbrella of data science. As you will see, deep learning is mostly about neural networks, a highly sophisticated and powerful algorithm that has enjoyed a lot of coverage and hype, and for good reason: it is very powerful and able to catch highly complex nonlinear relationships between different variables.

The aim of this chapter is to explain the functioning of neural networks before using them to predict financial time series in Python, just like you saw in ???.

A Walk Through Neural Networks

Artificial neural networks (ANNs) have their roots in the study of neurology, where researchers sought to comprehend how the human brain and its intricate network of interconnected neurons functioned. ANNs are designed to produce computational representations of biological neural network behavior.

ANNs have been around since the 1940s, when academics first started looking into ways to build computational models based on the human brain. Logician Walter Pitts and neurophysiologist Warren McCulloch were among the early pioneers in this subject. They published the idea of a computational model based on simplified artificial neurons in a paper.[1]

The development of artificial neural networks gained further momentum in the 1950s and 1960s when researchers like Frank Rosenblatt worked on the *perceptron*, a

1 W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* 5 (1943): 115–33.

type of artificial neuron that could learn from its inputs. Rosenblatt's work paved the way for the development of single-layer neural networks capable of pattern recognition tasks.

With the creation of multilayer neural networks, also known as *deep neural networks*, and the introduction of more potent algorithms, artificial neural networks made significant strides in the 1980s and 1990s. This innovation made it possible for neural networks to learn hierarchical data representations, which enhanced their performance on challenging tasks. Although multiple researchers contributed to the development and advancement of artificial neural networks, one influential figure is Geoffrey Hinton. Hinton, along with his collaborators, made significant contributions to the field by developing new learning algorithms and architectures for neural networks. His work on deep learning has been instrumental in the recent resurgence and success of artificial neural networks.

An ANN consists of interconnected nodes, called artificial neurons, organized into layers. The layers are typically divided into three types:

Input layer
: The input layer receives input data, which could be numerical, categorical, or even raw sensory data. Input layers are explanatory variables that are supposed to be predictive in nature.

Hidden layers
: The hidden layers (one or more) process the input data through their interconnected neurons. Each neuron in a layer receives inputs, performs a computation (discussed later), and passes the output to the next layer.

Output layer
: The output layer produces the final result or prediction based on the processed information from the hidden layers. The number of neurons in the output layer depends on the type of problem the network is designed to solve.

Figure 1-1 shows an illustration of an artificial neural network where the information flows from left to right. It begins with the two inputs being connected to the four hidden layers where calculation is done before outputting a weighted prediction in the output layer.

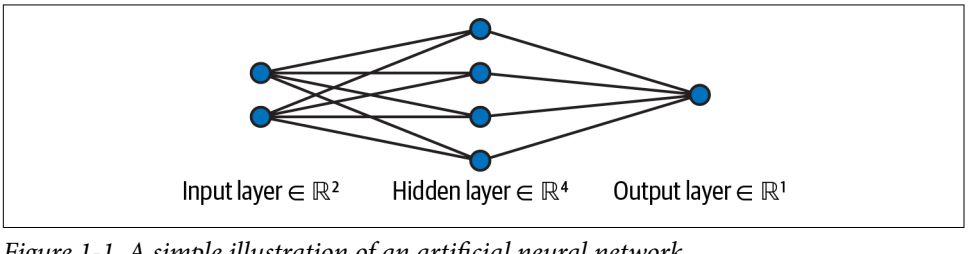


Figure 1-1. A simple illustration of an artificial neural network.

Each neuron in the ANN performs two main operations:

1. The neuron receives inputs from the previous layer or directly from the input data. Each input is multiplied by a weight value, which represents the strength or importance of that connection. The weighted inputs are then summed together.
2. After the weighted sum, an activation function (discussed in the next section) is applied to introduce nonlinearity into the output of the neuron. The activation function determines the neuron's output value based on the summed inputs.

During the training process, the ANN adjusts the weights of its connections to improve its performance. This is typically done through an iterative optimization algorithm, such as gradient descent, where the network's performance is evaluated using a defined loss function. The algorithm computes the gradient of the loss function with respect to the network's weights, allowing the weights to be updated in a way that minimizes the error.

ANNs have the ability to learn and generalize from data, making them suitable for tasks like pattern recognition and regression. With the advancements in deep learning, ANNs with multiple hidden layers have shown exceptional performance on complex tasks, leveraging their ability to learn hierarchical representations and capture intricate patterns in the data.

It is worth noting that the process from inputs to outputs is referred to as *forward propagation*.

Activation Functions

Activation functions in neural networks introduce nonlinearity to the output of a neuron, allowing neural networks to model complex relationships and learn from nonlinear data. They determine the output of a neuron based on the weighted sum of its inputs. Let's discuss these activation functions in detail.

The *sigmoid activation function* maps the input to a range between 0 and 1, making it suitable for binary classification problems or as a smooth approximation of a step function. The mathematical representation of the function is as follows:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Figure 1-2 shows the sigmoid function.

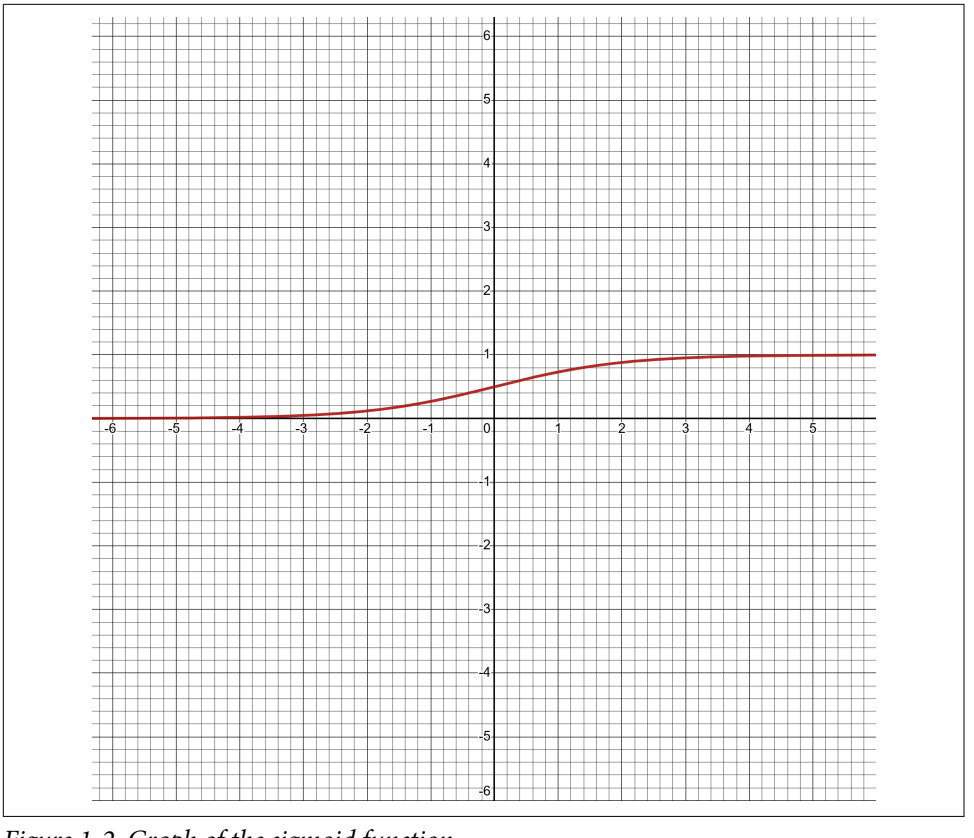


Figure 1-2. Graph of the sigmoid function.

Among the advantages of the sigmoid activation function are the following:

- It is a smooth as well as differentiable function that facilitates gradient-based optimization algorithms.
- It squashes the input to a bounded range, which can be interpreted as a probability or confidence level.

However, it has its limitations as well:

- It suffers from the *vanishing gradient problem*, where gradients become very small for extreme input values. This can hinder the learning process.
- Outputs are not zero centered, making it less suitable for certain situations, such as optimizing weights using symmetric update rules like the gradient descent.

The next activation function is the *hyperbolic tangent function* (tanh), which you saw in ???. The mathematical representation of the function is as follows:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Among the advantages of the hyperbolic tangent function are the following:

- It is similar to the sigmoid function but is zero centered, which helps alleviate the issue of asymmetric updates in weight optimization.
- Its nonlinearity can capture a wider range of data variations compared to the sigmoid function.

The following are among its limitations:

- It suffers from the vanishing gradient problem, particularly in deep networks.
- Outputs are still susceptible to saturation at the extremes, resulting in gradients close to zero.

Figure 1-3 shows the hyperbolic tangent function.

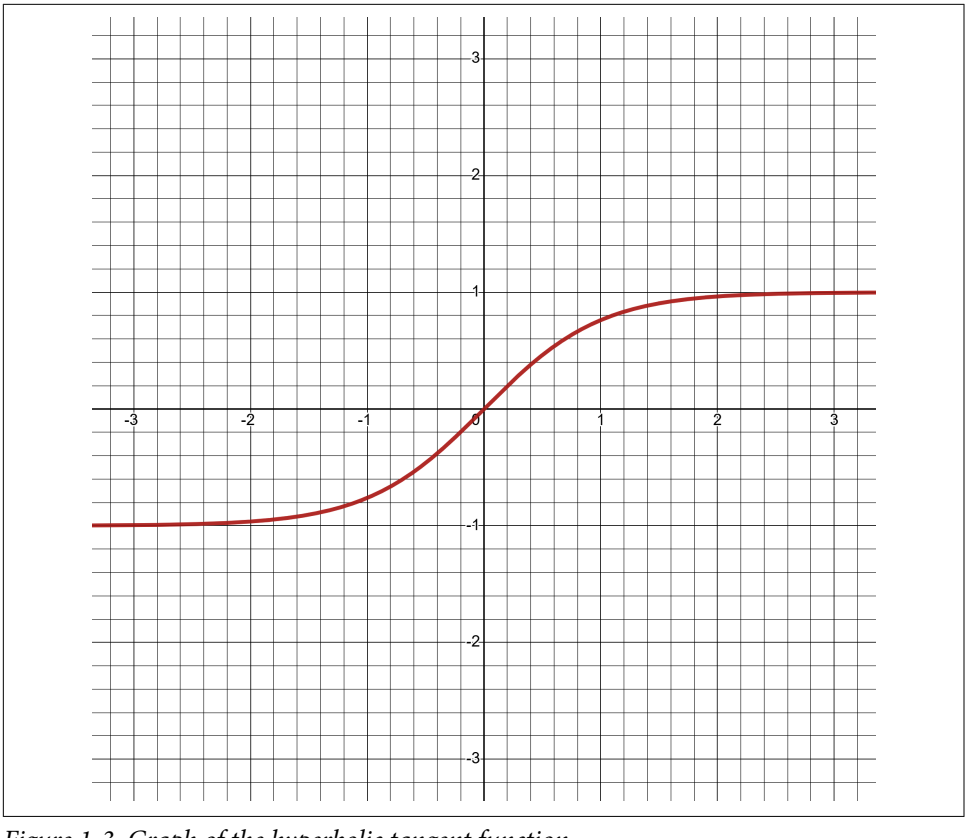


Figure 1-3. Graph of the hyperbolic tangent function.

The next function is called the *ReLU activation function*. ReLU stands for *rectified linear unit*. This function sets negative values to zero and keeps the positive values unchanged. It is efficient and helps avoid the vanishing gradient problem. The mathematical representation of the function is as follows:

$$f(x) = max(0, x)$$

Among the advantages of the ReLU function are the following:

- It is simple to implement, as it only involves taking the maximum of 0 and the input value. The simplicity of ReLU leads to faster computation and training compared to more complex activation functions.

- It helps mitigate the vanishing gradient problem that can occur during deep neural network training. The derivative of ReLU is either 0 or 1, which means that the gradients can flow more freely and avoid becoming exponentially small as the network gets deeper.

Among the limitations of the function are the following:

- It outputs 0 for negative input values, which can lead to information loss. In some cases, it may be beneficial to have activation functions that can produce negative outputs as well.
- It is not a smooth function, because its derivative is discontinuous at 0. This can cause optimization difficulties in certain scenarios.

Figure 1-4 shows the ReLU function.

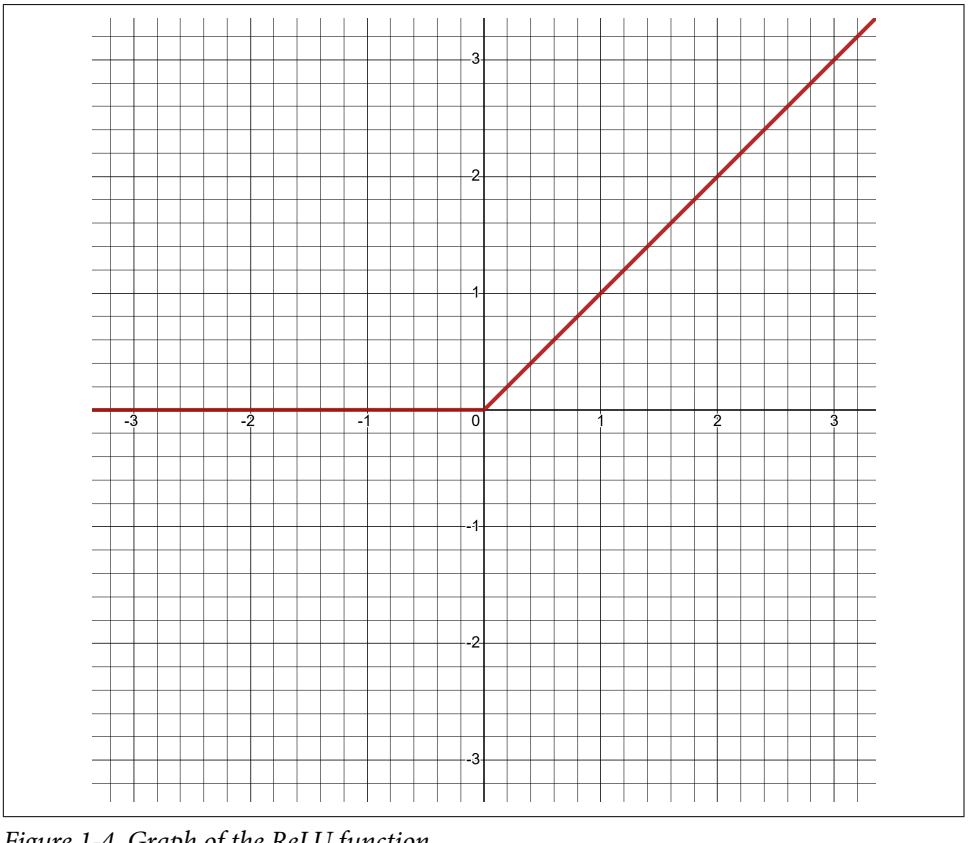


Figure 1-4. Graph of the ReLU function.

The final activation function to discuss is the *leaky ReLU activation function*. This activation function is an extension of the ReLU function that introduces a small slope for negative inputs. The mathematical representation of the function is as follows:

$$f(x) = max(0.01x, x)$$

Leaky ReLU addresses the dead neuron problem in ReLU and allows some activation for negative inputs, which can help with the flow of gradients during training.

Among the advantages of the leaky ReLU function are the following:

- It overcomes the issue of dead neurons that can occur with ReLU. By introducing a small slope for negative inputs, leaky ReLU ensures that even if a neuron is not activated, it can still contribute to the gradient flow during training.
- It is a continuous function, even at negative input values. The nonzero slope for negative inputs allows the activation function to have a defined derivative throughout its input range.

The following are among the limitations of the function:

- The slope of the leaky part is a hyperparameter that needs to be set manually. It requires careful tuning to strike a balance between avoiding dead neurons and preventing too much leakage that may hinder the nonlinearity of the activation function.
- Although leaky ReLU provides a nonzero response for negative inputs, it does not provide the same level of negative activation as some other activation functions, such as the hyperbolic tangent (tanh) and sigmoid. In scenarios where a strong negative activation response is desired, other activation functions might be more suitable.

Figure 1-5 shows the leaky ReLU function.

Your choice of activation function depends on the nature of the problem, the architecture of the network, and the desired behavior of the neurons in the network.

Activation functions typically take the weighted sum of inputs to a neuron and apply a nonlinear transformation to it. The transformed value is then passed on as the output of the neuron to the next layer of the network. The specific form and behavior of activation functions can vary, but their overall purpose is to introduce nonlinearities that allow the network to learn complex patterns and relationships in the data.

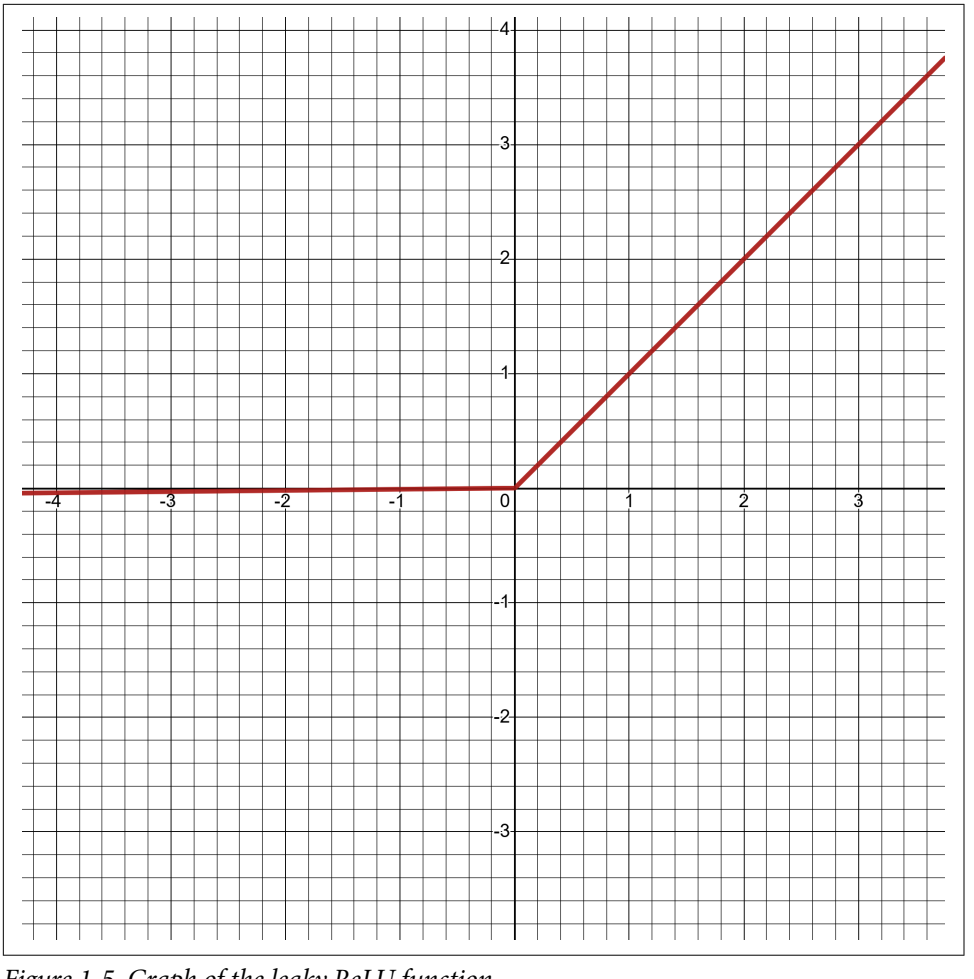


Figure 1-5. Graph of the leaky ReLU function.

To sum up, activation functions play a crucial role in ANNs by introducing nonlinearity into the network's computations. They are applied to the outputs of individual neurons or intermediate layers and help determine whether a neuron should be activated or not based on the input it receives. Without activation functions, the network would only be able to learn linear relationships between the input and output. However, most real-world problems (especially financial time series) involve complex, nonlinear relationships, so activation functions are essential for enabling neural networks to learn and represent such relationships effectively.

Backpropagation

Backpropagation is a fundamental algorithm used to train neural networks. It allows the network to update its weights in a way that minimizes the difference between the predicted output and the desired output.

Backpropagation is a shortened term for *backward propagation of errors*.

Training neural networks involves the following steps:

1. Randomly initialize the weights and biases of the neural network. This allows you to have a first step when you do not have initial information.
2. Perform *forward propagation*, a technique to calculate the predicted outputs of the network for a given input. As a reminder, this step involves calculating the weighted sum of inputs for each neuron, applying the activation function to the weighted sum, passing the value to the next layer (if it's not the last), and continuing the process until reaching the output layer (prediction).
3. Compare the predicted output with the actual output (test data) and calculate the loss, which represents the difference between them. The choice of the loss function (e.g., MAE or MSE) depends on the specific problem being solved.
4. Perform backpropagation to calculate the gradients of the loss with respect to the weights and biases. In this step, the algorithm will start from the output layer (the last layer) and go backward. It will compute the gradient of the loss with respect to the output of each neuron in the current layer. Then it will calculate the gradient of the loss with respect to the weighted sum of inputs for each neuron in the current layer by applying the chain rule. After that, it will compute the gradient of the loss with respect to the weights and biases of each neuron in the current layer using the gradients from the previous steps. These steps are repeated until the gradients are calculated for all layers.
5. Update the weights and biases of the network by using the calculated gradients and a chosen optimization algorithm run on a specific number of batches of data, which are controlled by the hyperparameter (referred to as the batch size). Updating the weights is done by subtracting the product of the learning rate and the gradient of the weights. Adjusting the biases is done by subtracting the product of the learning rate and the gradient of the biases. Repeat the preceding steps until the weights and biases are updated for all layers.
6. The algorithm then repeats steps 2–5 for a specified number of epochs or until a convergence criterion is met. An *epoch* represents one complete pass through the

entire training dataset (the whole process entails passing through the training dataset multiple times ideally).

7. Once the training is completed, evaluate the performance of the trained neural network on a separate validation or test dataset.

The *learning rate* is a hyperparameter that determines the step size at which a neural network's weights are updated during the training process. It controls how quickly or slowly the model learns from the data it's being trained on.

The *batch size* is a hyperparameter that determines the number of samples processed before updating the model's weights during each iteration of the training process. In other words, it specifies how many training examples are used at a time to calculate the gradients and update the weights.

Choosing an appropriate batch size is essential for efficient training and can impact the convergence speed and memory requirements. There is no one-size-fits-all answer to the ideal batch size, as it depends on various factors, such as the dataset size, available computational resources, and the complexity of the model.

Commonly used batch sizes for training MLPs range from small values (such as 16, 32, or 64) to larger ones (such as 128, 256, or even larger). Smaller batch sizes can offer more frequent weight updates and may help the model converge more quickly, especially when the dataset is large or has a lot of variations. However, smaller batch sizes may also introduce more noise and slower convergence due to frequent updates with less accurate gradients. On the other hand, larger batch sizes can provide more stable gradients and better utilization of parallel processing capabilities, leading to faster training on modern hardware. However, they might require more memory, and the updates are less frequent, which could slow down convergence or make the training process less robust.

As a general rule of thumb, you can start with a moderate batch size like 32 and experiment with different values to find the best trade-off between convergence speed and computational efficiency for your specific MLP model and dataset.

The backpropagation algorithm leverages the chain rule (refer to ??? for more information on calculus) to calculate the gradients by propagating the errors backward through the network.

By iteratively adjusting the weights based on the error propagated backward through the network, backpropagation enables the network to learn and improve its predictions over time. Backpropagation is a key algorithm in training neural networks and has contributed to significant advancements in various fields.

Optimization Algorithms

In neural networks, optimization algorithms, also known as *optimizers*, are used to update the parameters (weights and biases) of the network during the training process. These algorithms aim to minimize the loss function and find the optimal values for the parameters that result in the best performance of the network. There are several types of optimizers:

Gradient descent (GD)
: Gradient descent is the most fundamental optimization algorithm. It updates the network's weights and biases in the direction opposite to the gradient of the loss function with respect to the parameters. It adjusts the parameters by taking steps proportional to the negative of the gradient, multiplied by a learning rate.

Stochastic gradient descent (SGD)
: SGD is a variant of gradient descent that randomly selects a single training example or a mini batch of examples to compute the gradient and update the parameters. It provides a computationally efficient approach and introduces noise in the training process, which can help escape local optima.

Adaptive moment estimation (Adam)
: Adam is an adaptive optimization algorithm that computes adaptive learning rates for each parameter based on estimates of the first and second moments of the gradients. Adam is widely used due to its effectiveness and efficiency in various applications.

Root mean square propagation (RMSprop)
: The purpose of RMSprop is to address some of the limitations of the standard gradient descent algorithm, such as slow convergence and oscillations in different directions. RMSprop adjusts the learning rate for each parameter based on the average of the recent squared gradients. It calculates an exponentially weighted moving average of the squared gradients over time.

Each optimizer has its own characteristics, advantages, and limitations, and their performance can vary depending on the dataset and the network architecture. Experimentation and tuning are often necessary to determine the best optimizer for a specific task.

Regularization Techniques

Regularization techniques in neural networks are methods used to prevent overfitting, which can lead to poor performance and reduced ability of the model to make accurate predictions on new examples. Regularization techniques help to control the complexity of a neural network and improve its ability to generalize to unseen data.

Dropout is a regularization technique commonly used in neural networks to prevent overfitting (refer to ??? for detailed information on overfitting). It involves randomly omitting (dropping) a fraction of the neurons during training by setting their outputs to zero. This temporarily removes the neurons and their corresponding connections from the network, forcing the remaining neurons to learn more robust and independent representations.

The key idea behind dropout is that it acts as a form of model averaging or ensemble learning. By randomly dropping out neurons, the network becomes less reliant on specific neurons or connections and learns more robust features. Dropout also helps prevent co-adaptation, where certain neurons rely heavily on others, reducing their individual learning capability. As a result, dropout can improve the network's generalization ability and reduce overfitting.

Early stopping is a technique that also prevents overfitting by monitoring the model's performance on a validation set during training. It works by stopping the training process when the model's performance on the validation set starts to deteriorate. The idea behind early stopping is that as the model continues to train, it may start to overfit the training data, causing a decrease in performance on unseen data.

The training process is typically divided into epochs, where each epoch represents a complete pass over the training data. During training, the model's performance on the validation set is evaluated after each epoch. If the validation loss or a chosen metric starts to worsen consistently for a certain number of epochs, training is stopped, and the model's parameters from the epoch with the best performance are used as the final model.

Early stopping helps prevent overfitting by finding the optimal point at which the model has learned the most useful patterns without memorizing noise or irrelevant details from the training data. Both dropout and early stopping are key regularization techniques that help prevent overfitting and help stabilize the model.

Multilayer Perceptrons

A *multilayer perceptron* (MLP) is a type of ANN that consists of multiple layers of artificial neurons, or nodes, arranged in a sequential manner. It is a *feedforward neural network*, meaning that information flows through the network in one direction, from the input layer to the output layer, without any loops or feedback connections (you will learn more about this later in "Recurrent Neural Networks" on page 30).

The basic building block of an MLP is a *perceptron*, an artificial neuron that takes multiple inputs, applies weights to those inputs, performs a weighted sum, and passes the result through an activation function to produce an output (basically, the neuron that you have seen already). An MLP contains multiple perceptrons organized in

layers. It typically consists of an input layer, one or more hidden layers (the more layers, the deeper the learning process up to a certain point), and an output layer.

The term *perceptron* is sometimes used more broadly to refer to a single-layer neural network based on a perceptron-like architecture. In this context, the term *perceptron* can be used interchangeably with *neural network* or *single-layer perceptron*.

As a reminder, the input layer receives the raw input data, such as features from a dataset (e.g., the stationary values of a moving average). The hidden layers, which are intermediate layers between the input and output layers, perform complex transformations on the input data. Each neuron in a hidden layer takes inputs from all neurons in the previous layer, applies weights, performs the weighted sum, and passes the result through an activation function. The output layer produces the final output of the network.

MLPs are trained using backpropagation, which adjusts the weights of the neurons in the network to minimize the difference between the predicted output and the desired output. They are known for their ability to learn complex, nonlinear relationships in data, making them suitable for a wide range of tasks, including pattern recognition. Figure 1-6 shows an example of a deep MLP architecture.

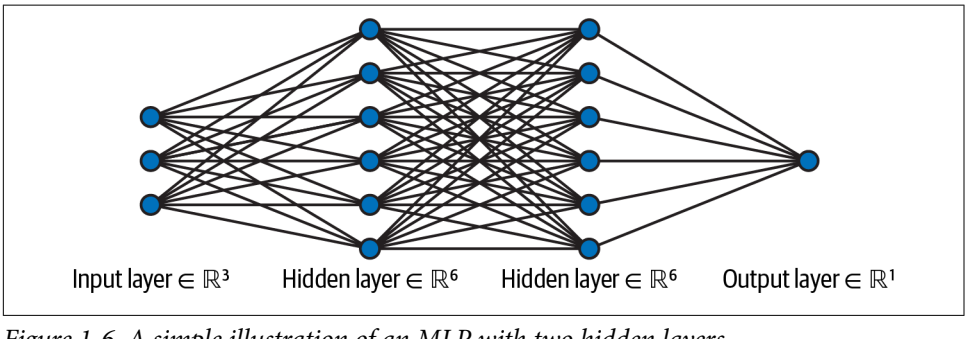


Figure 1-6. A simple illustration of an MLP with two hidden layers.

At this stage, you should understand that deep learning is basically neural networks with many hidden layers that add to the complexity of the learning process.

It is important to download *master_function.py* from this book's GitHub repository to access the functions seen in this book. After downloading it, you must set your Python's interpreter directory as the path where *master_function.py* is stored.

The aim of this section is to create an MLP to forecast daily S&P 500 returns. Import the required libraries:

```
from keras.models import Sequential
from keras.layers import Dense
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Now import the historical data and transform it:

```
# Set the start and end dates for the data
start_date = '1990-01-01'
end_date   = '2023-06-01'
# Fetch S&P 500 price data
data = np.array((pdr.get_data_fred('SP500', start = start_date,
                                   end = end_date)).dropna())
# Difference the data and make it stationary
data = np.diff(data[:, 0])
```

Set the hyperparameters for the model:

```
num_lags = 100
train_test_split = 0.80
num_neurons_in_hidden_layers = 20
num_epochs = 500
batch_size = 16
```

Use the data preprocessing function to create the four required arrays:

```
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data, num_lags,
                                                      train_test_split)
```

The following code block shows how to build the MLP architecture in *keras*. Make sure you understand the notes in the code:

```
# Designing the architecture of the model
model = Sequential()
# First hidden layer with ReLU as activation function
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
                activation = 'relu'))
# Second hidden layer with ReLU as activation function
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(1))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, np.reshape(y_train, (-1, 1)), epochs = num_epochs,
```

```
          batch_size = batch_size)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

When creating a `Dense` layer, you need to specify the `input_dim` parameter in the first layer of your neural network. For subsequent `Dense` layers, the `input_dim` is automatically inferred from the previous layer's output.

Let's plot the results and analyze the performance:

```
Accuracy Train =  92.4 %
Accuracy Test =  54.85 %
RMSE Train =  4.3602984254
RMSE Test =  75.7542774467
Correlation In-Sample Predicted/Train =  0.989
Correlation Out-of-Sample Predicted/Test =  0.044
Model Bias =  1.03
```

Figure 1-7 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

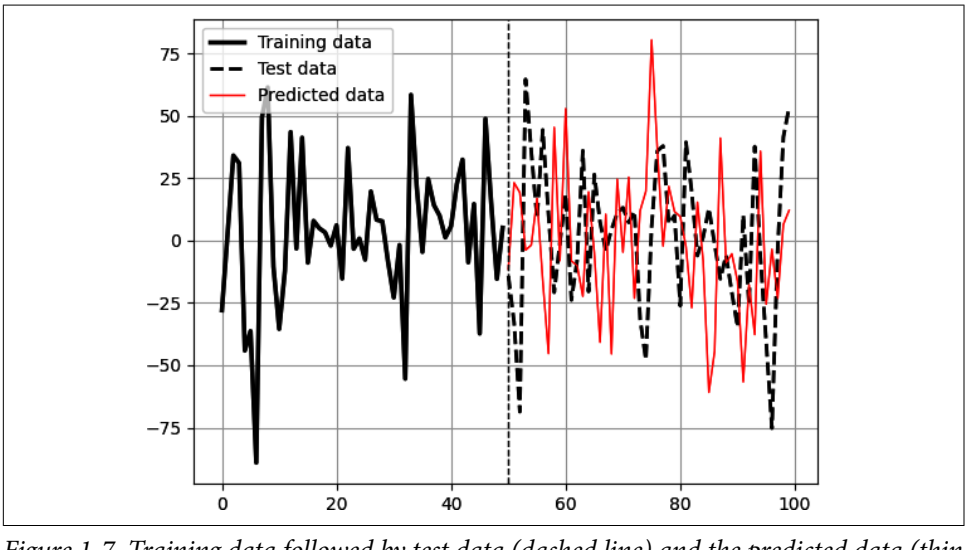


Figure 1-7. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the MLP regression algorithm.

To purchase this book, you can visit O'Reilly's Website or Amazon

Click here to view on O'Reilly Media's Website

Click here to view on Amazon