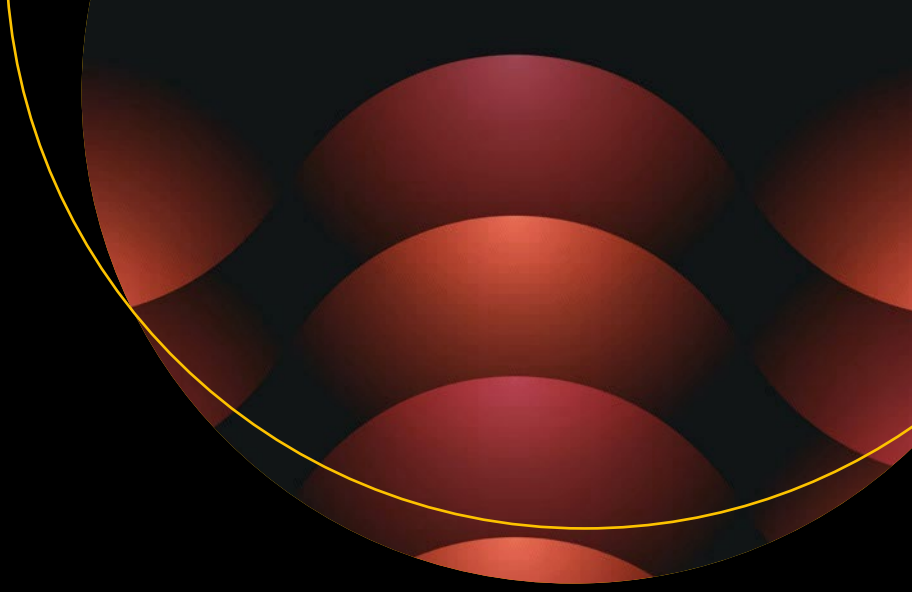

# Reinforcement Learning for Finance

Solve Problems in Finance with CNN and RNN Using the TensorFlow Library

Samit Ahlawat

Apress®

# Reinforcement Learning for Finance

## Solve Problems in Finance with CNN and RNN Using the TensorFlow Library

**Samit Ahlawat**

Apress®

*Reinforcement Learning for Finance: Solve Problems in Finance with CNN and RNN Using the TensorFlow Library*

Samit Ahlawat
Irvington, NJ, USA

ISBN-13 (pbk): 978-1-4842-8834-4 ISBN-13 (electronic): 978-1-4842-8835-1
https://doi.org/10.1007/978-1-4842-8835-1



Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Laura Berendson
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Joel Filipe on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

*To my family and friends without whose support this book would not have been possible.*

# Table of Contents

# About the Author

**Samit Ahlawat** is Senior Vice President in Quantitative Research, Capital Modeling, at JPMorgan Chase in New York, USA. In his current role, he is responsible for building trading strategies for asset management and for building risk management models. His research interests include artificial intelligence, risk management, and algorithmic trading strategies. He has given CQF Institute talks on artificial intelligence, has authored several research papers in finance, and holds a patent for facial recognition technology. In his spare time, he contributes to open source code.

# Acknowledgments

I would like to express my heartfelt appreciation for my friends and coworkers, in academia and the workplace, who encouraged me to write this book.

# Preface

When I began using artificial intelligence tools in quantitative financial research, I could not find a comprehensive introductory text focusing on financial applications. Neural network libraries like TensorFlow, PyTorch, and Caffe had made tremendous contributions in the rapid development, testing, and deployment of deep neural networks, but I found most applications restricted to computer science, computer vision, and robotics. Having to use reinforcement learning algorithms in finance served as another reminder of the paucity of texts in this field. Furthermore, I found myself referring to scholarly articles and papers for mathematical proofs of new reinforcement learning algorithms. This led me to write this book to provide a one-stop resource for Python programmers to learn the theory behind reinforcement learning, augmented with practical examples drawn from the field of finance.

In practical applications, reinforcement learning draws upon deep neural networks. To facilitate exposition of topics in reinforcement learning and for continuity, this book also provides an introduction to TensorFlow and covers neural network topics like convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

Finally, this book also introduces readers to writing modular, reusable, and extensible reinforcement learning code. Having worked on developing trading strategies using reinforcement learning and publishing papers, I felt existing reinforcement learning libraries like TF-Agents are tightly coupled with the underlying implementation framework and do not

express central concepts in reinforcement learning in a manner that is modular enough for someone conversant with concepts to pick up TF-Agent library usage or extend its algorithms for specific applications. The code samples covered in this book provide examples of how to write modular code for reinforcement learning.

# Introduction

Reinforcement learning is a rapidly growing area of artificial intelligence that involves an agent learning from past experience of rewards gained by taking specific actions in certain states. The agent seeks to learn a policy prescribing the optimum action in each state with the objective of maximizing expected discounted future rewards. It is an unsupervised learning technique where the agent learns the optimum policy by past interactions with the environment. Supervised learning, by contrast, seeks to learn the pattern of output corresponding to each state in training data. It attempts to train the model parameters in order to get a close correspondence between predicted and actual output for a given set of inputs. This book outlines the theory behind reinforcement learning and illustrates it with examples of implementations using TensorFlow. The examples demonstrate the theory and implementation details of the algorithms, supplemented with a discussion of corresponding APIs from TensorFlow and examples drawn from quantitative finance. It guides a reader familiar with Python programming from basic to advanced understanding of reinforcement learning algorithms, coupled with a comprehensive discussion on how to use state-of-the-art software libraries to implement advanced algorithms in reinforcement learning.

Most applications of reinforcement learning have focused on robotics or computer science tasks. By focusing on examples drawn from finance, this book illustrates a spectrum of financial applications that can benefit from reinforcement learning.

# CHAPTER 1

# Overview

Deep neural networks have transformed virtually every scientific human endeavor – from image recognition, medical imaging, robotics, and self-driving cars to space exploration. The extent of transformation heralded by neural networks is unrivaled in contemporary human history, judging by the range of new products that leverage neural networks. Smartphones, smartwatches, and digital assistants – to name a few – demonstrate the promise of neural networks and signal their emergence as a mainstream technology. The rapid development of artificial intelligence and machine learning algorithms has coincided with increasing computational power, enabling them to run rapidly. Keeping pace with new developments in this field, various open source libraries implementing neural networks have blossomed. Python has emerged as the *lingua franca* of the artificial intelligence programming community. This book aims to equip Python-proficient programmers with a comprehensive knowledge on how to use the TensorFlow library for coding deep neural networks and reinforcement learning algorithms effectively. It achieves this by providing detailed mathematical proofs of key theorems, supplemented by implementation of those algorithms to solve real-life problems.

Finance has been an early adopter of artificial intelligence algorithms with the application of neural networks in designing trading strategies as early as the 1980s. For example, White (1988) applied a simple neural network to find nonlinear patterns in IBM stock price. However, recent cutting-edge research on reinforcement learning has focused


S. Ahlawat, *Reinforcement Learning for Finance*,
https://doi.org/10.1007/978-1-4842-8835-1_1

predominantly on robotics, computer science, or interactive game-playing. The lack of financial applications has led many to question the applicability of deep neural networks in finance where traditional quantitative models are ubiquitous. Finance practitioners feel that the lack of rigorous mathematical proofs and transparency about how neural networks work has restricted their wider adoption within finance. This book aims to address both of these concerns by focusing on real-life financial applications of neural networks.

## 1.1 Methods for Training Neural Networks

Neural networks can be trained using one of the following three methods:

1. **Supervised learning** involves using a training dataset with known output, also called ground truth values. For a classification task, this would be the true labels, while for a regression task, it would be the actual output value. A loss function is formulated that measures the deviation of the model output from the true output. This function is minimized with respect to model parameters using stochastic gradient descent.

2. **Unsupervised learning** methods use a training dataset made up of input features without any knowledge of the true output values. The objective is to classify inputs into clusters for clustering or dimension reduction applications or for identifying outliers.

3. **Reinforcement learning** involves an agent that learns an optimal policy within the framework of a Markov decision problem (MDP). The training

dataset consists of a set of actions taken in different states by an agent, followed by rewards earned and the next state to which the agent transitions. Using the history of rewards, reinforcement learning attempts to learn an optimal policy to maximize the expected sum of discounted future rewards. This book focuses on reinforcement learning.

## 1.2 Machine Learning in Finance

Machine learning applications in finance date back to the 1980s with the use of neural networks in stock price prediction (White, 1988). Within finance, automated trading strategies and portfolio management have been early adopters of artificial intelligence and machine learning tools. Allen and Karjalainen (1999) applied genetic algorithms to combine simple trading rules to form more complex ones. More recent applications of machine learning in finance can be seen in the works of Savin et al. (2007), who used the pattern recognition method presented by Lo et al. (2000) to test if the head-and-shoulders pattern had predictive power; Chavarnakul and Enke (2008), who employed a generalized regression neural network (GRNN) to construct two trading strategies based on equivolume charting that predicted the next day's price using volume- and price-based technical indicators; and Ahlawat (2016), who applied probabilistic neural networks to predict technical patterns in stock prices. Other works include Enke and Thawornwong (2005), Li and Kuo (2008), and Leigh et al. (2005). Chenoweth et al. (1996) have studied the application of neural networks in finance. Enke and Thawornwong (2005) tested the hypothesis that neural networks can provide superior prediction of future returns based on their ability to identify nonlinear relationships. They employed only fundamental measures and did not consider technical ones. Their neural network provided higher returns than the buy-and-hold strategy, but they did not consider transaction costs.

There are many other applications of machine learning in finance besides trading strategies, perhaps less glamorous but equally significant in business impact. This book gives a comprehensive exposition of several machine learning applications in finance that are at cutting edge of research and practical use.

## 1.3 Structure of the Book

This book begins with an introduction to the TensorFlow library in Chapter 2 and illustrates the concepts with financial applications that involve building models to solve practical problems. The datasets for problems are publicly available. Relevant concepts are illustrated with mathematical equations and concise explanations.

Chapter 3 introduces readers to convolutional neural networks (CNNs), and Chapter 4 follows up with a similar treatment of recurrent neural networks (RNNs). These networks are frequently used in building value function models and policies in reinforcement learning, and a comprehensive understanding of CNN and RNN is indispensable for using reinforcement learning effectively on practical problems. As before, all foundational concepts are illustrated with mathematical theory, explanation, and practical implementation examples.

Chapter 5 introduces reinforcement learning concepts: from Markov decision problem (MDP) formulation to defining value function and policies, followed by a comprehensive discussion of reinforcement learning algorithms illustrated with examples and mathematical proofs.

Finally, Chapter 6 provides a discussion of recent, groundbreaking advances in reinforcement learning by discussing technical papers and applying those algorithms to practical applications.

CHAPTER 2

# Introduction to TensorFlow

TensorFlow is an open source, high-performance machine learning library developed by Google and released for public use in 2015. It has interfaces for Python, C++, and Java programming languages. It has the option of running on multiple CPUs or GPUs. TensorFlow offers two modes of execution: eager mode that can be run immediately and graph mode that creates a dependency graph and executes nodes in that graph only where needed.

This book uses TensorFlow 2.9.1. Older TensorFlow constructs from version 1 of the library such as **Session** and **placeholder** are not covered here. Their use has been rendered obsolete in TensorFlow version 2.0 and higher. Output shown in the code listings has been generated using the PyCharm IDE's interactive shell.

## 2.1 Tensors and Variables

Tensors are n-dimensional arrays, similar in functionality to the numpy library's ndarray object. They are instances of the **tf.Tensor** object. A three-dimensional tensor of 32-bit floating-point numbers can be created using code in Listing 2-1. Tensor has attributes shape and dtype that tell the shape and data type of the tensor. Once created, tensors retain their shape.


S. Ahlawat, *Reinforcement Learning for Finance*,
https://doi.org/10.1007/978-1-4842-8835-1_2

***Listing 2-1.*** Creating a Three-Dimensional Tensor

```
import tensorflow as tf

tensor = tf.constant([[list(range(3))],
                      [list(range(1, 4))],
                      [list(range(2, 5))]], dtype=tf.
                      float32)

print(tensor)

tf.Tensor(
[[[0. 1. 2.]]
[[1. 2. 3.]]
[[2. 3. 4.]]], shape=(3, 1, 3), dtype=float32)
```

Most numpy functions for creating ndarrays have analogs in TensorFlow, for example, tf.ones, tf.zeros, tf.eye, tf.ones_like, etc. Tensors support usual mathematical operations like +, −, etc., in addition to matrix operations like transpose, matmul, and einsum, as shown in Listing 2-2.

***Listing 2-2.*** Mathematical Operations on Tensors

```
import tensorflow as tf

ar = tf.constant([[1, 2], [2, 2]], dtype=tf.float32)

print(ar)
<tf.Tensor: id=1, shape=(2, 2), dtype=float32, numpy=
array([[1., 2.],
[2., 2.]], dtype=float32)>

# elementwise multiplication
```

```
print(ar * ar)
Out[8]:
<tf.Tensor: id=2, shape=(2, 2), dtype=float32, numpy=
array([[1., 4.],
[4., 4.]], dtype=float32)>

# matrix multiplication C = tf.matmul(A, B) => cij =
sum_k (aik * bkj)
print(tf.matmul(ar, tf.transpose(ar)))

<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[5., 6.],
[6., 8.]], dtype=float32)>

# generic way of matrix multiplication
print(tf.einsum("ij,kj->ik", ar, ar))

<tf.Tensor: id=23, shape=(2, 2), dtype=float32, numpy=
array([[5., 6.],
[6., 8.]], dtype=float32)>

# cross product
print(tf.einsum("ij,kl->ijkl", ar, ar))

<tf.Tensor: id=32, shape=(2, 2, 2, 2),
dtype=float32, numpy=
array([[[[1., 2.],
[2., 2.]],
[[2., 4.],
[4., 4.]]],
[[[2., 4.],
```

```
40   [4., 4.]],
41   [[2., 4.],
42   [4., 4.]]]], dtype=float32)>
```

Tensors can be sliced using the usual Python notation with a semicolon. For advanced slicing, use **tf.slice** that accepts a begin index and the number of elements along each axis to slice. **tf.strided_slice** can be used for adding a stride. To obtain specific indices from a tensor, use **tf.gather**. To extract specific elements of a multidimensional tensor specified by a list of indices, use **tf.gather_nd**. These APIs are illustrated using examples in Listing 2-3.

***Listing 2-3.*** Tensor Slicing Operations

```
 1   import tensorflow as tf
 2
 3   tensor = tf.constant([[1, 2], [2, 2]], dtype=tf.float32)
 4
 5   print(tensor[1:, :])
 6   <tf.Tensor: id=37, shape=(1, 2), dtype=float32,
     numpy=array([[2., 2.]], dtype=float32)>
 7
 8   print(tf.slice(tensor, begin=[0,1], size=[2, 1]))
 9   tf.Tensor(
10   [[2.]
11   [2.]], shape=(2, 1), dtype=float32)
12
13   print(tf.gather_nd(tensor, indices=[[0, 1], [1, 0]]))
14   Out[18]: <tf.Tensor: id=42, shape=(2,), dtype=float32,
     numpy=array([2., 2.], dtype=float32)>
```

Ragged tensors are tensors with a nonuniform shape along an axis, as illustrated in Listing 2-4.

***Listing 2-4.*** Ragged Tensors

```
import tensorflow as tf

jagged = tf.ragged.constant([[1, 2], [2]])
print(jagged)
<tf.RaggedTensor [[1, 2], [2]]>
```

TensorFlow allows space-efficient storage of sparse arrays, that is, arrays with most elements as 0. The **tf.sparse.SparseTensor** API takes the indices of non-zero elements, their values, and the dense shape of the sparse array. This is shown in Listing 2-5.

***Listing 2-5.*** Sparse Tensors

```
import tensorflow as tf

tensor = tf.sparse.SparseTensor(indices=[[1,0], [2,2]],
values=[1, 2], dense_shape=[3, 4])
print(tensor)
SparseTensor(indices=tf.Tensor(
[[1 0]
[2 2]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2],
shape=(2,), dtype=int32), dense_shape=tf.Tensor([3 4],
shape=(2,), dtype=int64))

print(tf.sparse.to_dense(tensor))
tf.Tensor(
[[0 0 0 0]
[1 0 0 0]
[0 0 2 0]], shape=(3, 4), dtype=int32)
```

In contrast to **tf.Tensor** that is immutable after creation, a TensorFlow variable can be changed. A variable is an instance of the **tf.Variable** class and can be created by initializing it with a tensor. Variables can be converted to tensors using **tf.convert_to_tensor**. Variables cannot be reshaped after creation, only modified. Calling **tf.reshape** on a variable returns a new tensor. Variables can also be created from another variable, but the operation copies the underlying tensor. Variables do not share underlying data. **assign** can be used to update the variable by changing its data tensor. **assign_add** is another useful method of a variable that replicates the functionality of the += operator. Operations on tensors like **matmul** or **einsum** can also be applied to variables or to a combination of tensor and variable. Variable has a Boolean attribute called **trainable** that signifies if the variable is to be trained during backpropagation. Operations on variables are shown in Listing 2-6.

***Listing 2-6.*** Variables

```
import tensorflow as tf

tensor = tf.constant([[1, 2], [3, 4]])
variable = tf.Variable(tensor)
print(variable)
<tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[1, 2],
[3, 4]])>

# return the index of highest element
print(tf.math.argmax(variable))

tf.Tensor([1 1], shape=(2,), dtype=int64)

print(tf.convert_to_tensor(variable))
tf.Tensor(

10
```

```
17   [[1 2]
18   [3 4]], shape=(2, 2), dtype=int32)
19
20   print(variable.assign([[1,2], [1, 1]]))
21   <tf.Variable 'UnreadVariable' shape=(2, 2) dtype=int32,
     numpy=
22   array([[1, 2],
23   [1, 1]])>
```

## 2.2 Graphs, Operations, and Functions

There are two modes of execution within TensorFlow: eager execution and graph execution. Eager mode of execution processes instructions as they occur in the code, while graph execution is delayed. Graph mode builds a dependency graph connecting the data represented as tensors (or variables) using operations and functions. After the graph is built, it is executed. Graph execution offers a few advantages over eager execution:

1. Graphs can be exported to files or executed in non-Python environments such as mobile devices.
2. Graphs can be compiled to speed up execution.
3. Nodes with static data and operations on those nodes can be precomputed.
4. Node values that are used multiple times can be cached.
5. Branches of the graph can be identified for parallel execution.

Operations in TensorFlow are represented using the **tf.Operation** class and can be used as a node. Operation nodes are created using one of the predefined operations such as **tf.matmul**, **tf.reduce_sum**, etc. To create a

new operation, use the **tf.Operation** class. A few important operations are enumerated in the following. All of them can be accessed directly using the **tf.operation_name** syntax.

1. Operations defined in the **tf.math** library:
   - **tf.abs**: Calculates the absolute value of a tensor.
   - **tf.divide**: Divides two tensors.
   - **tf.maximum**: Returns the element-wise maximum of two tensors.
   - **tf.reduce_sum**: Calculates the sum of all tensor elements. It takes an optional axis argument to calculate the sum along that axis.
2. Operations defined in the **tf.linalg** library:
   (a). **tf.det**: Calculates the determinant of a square matrix
   (b). **tf.svd**: Calculates the SVD decomposition of a rectangular matrix provided as a tensor
   (c). **tf.trace**: Returns the trace of a tensor

Functions are defined using the **tf.function** method, passing the Python function as an argument. **tf.function** is a decorator that augments a Python function with attributes necessary for running it in a TensorFlow graph. A few examples of TensorFlow operations and functions are illustrated in Listing 2-7. Each TensorFlow function generates an internal graph from its arguments. By default, a TensorFlow function uses a graph execution model. To switch to eager execution mode, set **tf.config.run_functions_eagerly(True)**. Please note that the following output may not match output from another run because of random numbers used.

***Listing 2-7.*** TensorFlow Operations and Functions

```
import tensorflow as tf
import numpy as np

tensor = tf.constant(np.ones((3, 3), dtype=np.int32))

print(tensor)

<tf.Tensor: id=0, shape=(3, 3), dtype=int32, numpy=
array([[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])>

print(tf.reduce_sum(tensor))
<tf.Tensor: id=2, shape=(), dtype=int32, numpy=9>

print(tf.reduce_sum(tensor, axis=1))
<tf.Tensor: id=4, shape=(3,), dtype=int32, numpy=
array([3, 3, 3])>

@tf.function
def sigmoid_activation(inputs, weights, bias):
    x = tf.matmul(inputs, weights) + bias
    return tf.divide(1.0, 1 + tf.exp(-x))

inputs = tf.constant(np.ones((1, 3), dtype=np.float64))
weights = tf.Variable(np.random.random((3, 1)))
bias = tf.ones((1, 3), dtype=tf.float64)
```

```
27
28   print(sigmoid_activation(inputs, weights, bias))
29   <tf.Tensor: id=195, shape=(1, 3), dtype=float64,
     numpy=array([[0.89564016, 0.89564016, 0.89564016]])>
```

Code shown in Listing 2-8 sets the default execution mode to graph mode.

***Listing 2-8.*** Running TensorFlow Operations in Graph (Non-eager) Mode

```
1   import timeit
2
3   tf.config.experimental_run_functions_eagerly(False)
4   t1 = timeit.timeit(lambda: sigmoid_activation(inputs,
    weights, tf.constant(np.random.random((1, 3)))),
    number=1000)
5   print(t1)
6   0.7758807
```

## 2.3 Modules

TensorFlow uses the base class **tf.Module** to build layers and models. A module is a class that keeps track of its state using instance variables and can be called as a function. To achieve this, it must provide an implementation for the method **__call__**. This is illustrated in Listing 2-9. Due to the use of random numbers, output values may vary from those shown.

***Listing 2-9.*** Custom Module

```
1   import tensorflow as tf
2   import numpy as np
3
4
14
```

```
class ExampleModule(tf.Module):
    def __init__(self, name=None):
        super(ExampleModule, self).__init__(name=name)
        self.weights = tf.Variable(np.random.random(5),
        name="weights")
        self.const = tf.Variable(np.array([1.0]),
        dtype=tf.float64,
        trainable=False, name="constant")

    def __call__(self, x, *args, **kwargs):
        return tf.matmul(x, self.weights[:, tf.newaxis]) +
        self.const[tf.newaxis, :]


em = ExampleModule()
x = tf.constant(np.ones((1, 5)), dtype=tf.float64)
print(em(x))


<tf.Tensor: id=24631, shape=(1, 1), dtype=float64,
numpy=array([[2.45019464]])>
```

Module is the base class for both layers and models. It can be used as a model, serving as a collection of layers. Module shown in Listing 2-10 defers the creation of weights for the first layer until inputs are provided. Once input shape is known, it creates the tensors to store the weights. Decorator **tf.function** can be added to the **__call__** method to convert it to a graph.

***Listing 2-10.*** Module

```
import tensorflow as tf


class InferInputSizeModule(tf.Module):
    def __init__(self, noutput, name=None):
        super().__init__(name=name)
        self.weights = None
        self.noutput = noutput
        self.bias = tf.Variable(tf.zeros([noutput]),
        name="bias")

    def __call__(self, x, *args, **kwargs):
        if self.weights is None:
            self.weights = tf.Variable(tf.random.
            normal([x.shape[-1], self.noutput]))

        output = tf.matmul(x, self.weights) + self.bias
        return tf.nn.sigmoid(output)

class SimpleModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)

        self.layer1 = InferInputSizeModule(noutput=4)
        self.layer2 = InferInputSizeModule(noutput=1)

    @tf.function
    def __call__(self, x, *args, **kwargs):
        x = self.layer1(x)
        return self.layer2(x)
```

```
29
30   model = SimpleModel()
31   print(model(tf.ones((1, 10))))
32
33   <tf.Tensor: id=24700, shape=(1, 1), dtype=float32,
     numpy=array([[0.632286]], dtype=float32)>
```

Objects of type **tf.Module** can be saved to checkpoint files. Creating a checkpoint creates two files: one with module data and another containing metadata with extension **.index**. Saving a module to a checkpoint and loading it back from a checkpoint is illustrated in Listing 2-11.

***Listing 2-11.*** Checkpoint a Model

```
 1   import tensorflow as tf
 2
 3   path = r"C:\temp\simplemodel"
 4   checkpoint = tf.train.Checkpoint(model=model)
 5   checkpoint.write(path)
 6
 7
 8   model2 = SimpleModel()
 9   model_orig = tf.train.Checkpoint(model=model2)
10   model_orig.restore(path)
```

## 2.4 Layers

Layers are objects with **tf.keras.layers.Layer** as the base class. The Keras library is used in TensorFlow for implementing layers and models. The **tf.keras.layers.Layer** class derives from the **tf.Module** class and has a method **call** in place of the **__call__** method in **tf.Module**. There are several advantages to using Keras instead of **tf.Module**. For instance, training variables of nested Keras layers are automatically collected for