# CHAPTER 1: Introduction to Machine Learning with TensorFlow

## Overview

In this chapter, you will learn how to create, utilize, and apply linear transformations to the fundamental building blocks of programming with TensorFlow: tensors. You will then utilize tensors to understand the complex concepts associated with neural networks, including tensor reshaping, transposition, and multiplication.

## Introduction

Machine learning (ML) has permeated various aspects of daily life that are unknown to many. From the recommendations of your daily social feeds to the results of your online searches, they are all powered by machine learning algorithms. These algorithms began in research environments solving niche problems, but as their accessibility broadened, so too have their applications for broader use cases. Researchers and businesses of all types recognize the value of using models to optimize every aspect of their respective operations. Doctors can use machine learning to decide diagnosis and treatment options, retailers can use ML to get the right products to their stores at the right time, and entertainment companies can use ML to provide personalized recommendations to their customers.

In the age of data, machine learning models have proven to be valuable assets to any data-driven company. The large quantities of data available allow powerful and accurate models to be created to complete a variety of tasks, from regression to classification, recommendations to time series analysis, and even generative art, many of which will be covered in this workshop. And all can be built, trained, and deployed with TensorFlow.

The TensorFlow API has a huge amount of functionality that has made it popular among all machine learning practitioners building machine learning models or working with tensors, which are multidimensional numerical arrays. For researchers, TensorFlow is an appropriate choice to create new machine learning applications due to its advanced customization and flexibility. For developers, TensorFlow is an excellent choice of machine learning library due to its ease in terms of deploying models from development to production environments. Combined, TensorFlow's flexibility and ease of deployment make the library a smart choice for many practitioners looking to build performant machine learning models using a variety of different data sources and to replicate the results of that learning in production environments.

This chapter provides a practical introduction to TensorFlow's API. You will learn how to perform mathematical operations pertinent to machine learning that will give you a firm foundation for building performant ML models using TensorFlow. You will first learn basic operations such as how to create variables with the API. Following that, you will learn how to perform linear transformations such as addition before moving on to more advanced tasks, including tensor multiplication.

## Implementing Artificial Neural Networks in TensorFlow

The advanced flexibility that TensorFlow offers lends itself well to creating artificial neural networks (ANNs). ANNs are algorithms that are inspired by the connectivity of neurons in the brain and are intended to replicate the process in which humans learn. They consist of layers through which information propagates from the input to the output.

Figure 1.1 shows a visual representation of an ANN. An input layer is on the left-hand side, which, in this example, has two features (X1 and X2). The input layer is connected to the first hidden layer, which has three units. All the data from the previous layer gets passed to each unit in the first hidden layer. The data is then passed to the second hidden layer, which also has three units. Again, the information from each unit of the prior layer is passed to each unit of the second hidden layer. Finally, all the information from the second hidden layer is passed to the output layer, which has one unit, representing a single number for each set of input features.

Diagram

Description automatically generated

Figure 1.1: A visual representation of an ANN with two hidden layers

ANNs have proven to be successful in learning complex and nonlinear relationships with large, unstructured datasets, such as audio, images, and text data. While the results can be impressive, there is a lot of variability in how ANNs can be configured. For example, the number of layers, the size of each layer, and which nonlinear function should be used are some of the factors that determine the configuration of ANNs. Not only are the classes and functions that TensorFlow provides well-suited to building and training ANNs, but the library also supplies a suite of tools to help visualize and debug ANNs during the training process.

Compared with traditional machine learning algorithms, such as linear and logistic regression, ANNs can outperform them when provided with large amounts of data. ANNs are advantageous since they can be fed unstructured data and feature engineering is not necessarily required. Data pre-processing can be a time-intensive process. Therefore, many practitioners prefer ANNs if there is a large amount of data.

Many companies from all sectors utilize TensorFlow to build ANNs for their applications. Since TensorFlow is backed by Google, the company utilizes the library for much of its research, development, and production of machine learning applications. However, there are many other companies that also use the library. Companies such as Airbnb, Coca-Cola, Uber, and GE Healthcare all utilize the library for a variety of tasks. The use of ANNs is particularly appealing since they can achieve remarkable accuracy if provided with sufficient data and trained appropriately. For example, GE Healthcare uses TensorFlow to build ANNs to identify specific anatomy regardless of orientation from magnetic resonance images to improve speed and accuracy. By using ANNs, they can achieve over 99% accuracy in identifying anatomy in seconds, regardless of head rotation, which would otherwise take a trained professional much more time.

While the number of companies utilizing ANNs is vast, ANNs may not be the most appropriate choice for solving all business problems. In such an environment, you must answer the following questions to determine whether ANNs are the most appropriate choice:

* **Does the problem have a numerical solution?** Machine learning algorithms, ANNs included, generate predicted numerical results based on input data. For example, machine learning algorithms may predict a given number, such as the temperature of a city given the location and previous weather conditions, or the stock price given previous stock prices, or label images into a given number of categories. In each of these examples, a numerical output is generated based on the data provided and, given enough labeled data, models can perform well. However, when the desired result is more abstract, or creativity is needed, such as creating a new song, then machine learning algorithms may not be the most appropriate choice, since a well-defined numerical solution may not be available.
* **Is there enough appropriately labeled data to train a model?** For a supervised learning task, you must have at least some labeled data to train a model. For example, if you want to build a model to predict financial stock data for a given company, you will first need historical training data. If the company in question has not been public for very long, there may not be adequate training data. ANNs can often require a lot of data. When working with images, ANNs often need millions of training examples to develop accurate, robust models. This may be a determining factor for consideration when deciding which algorithm is appropriate for a given task.

Now that you are aware of what TensorFlow is, consider the following advantages and disadvantages of TensorFlow.

## Advantages of TensorFlow

The following are a few of the main advantages of using TensorFlow that many practitioners consider when deciding whether to pursue the library for machine learning purposes:

* **Library Management**: There is a large community of practitioners that maintain the TensorFlow library to keep it up to date with frequent new releases to help fix bugs, add new functions and classes to reflect current advances in the field, and add support for multiple programming languages.
* **Pipelining**: TensorFlow supports end-to-end model production, from model development in highly parallelizable environments that support GPU processing to a suite of model deployment tools. Also, there are lightweight libraries in TensorFlow that are used for deploying trained TensorFlow models on mobile and embedded devices, such as **Internet of Things** (**IoT**) devices.
* **Community Support**: The community of practitioners that use and support the library is vast and they support each other, because of which those practitioners who are new to the library achieve the results they are looking for easily.
* **Open Source**: TensorFlow is an open source library, and its code base is available for anyone to use and modify for their own applications.
* **Works with Multiple Languages**: While the library is natively designed for Python, models can now be trained and deployed in JavaScript.

## Disadvantages of TensorFlow

The following are a few of the disadvantages of using TensorFlow:

* **Computational Speed**: Since the primary programming language of TensorFlow is Python, the library is not as computationally fast as it could be if it were native to other languages, such as C++.
* **Steep Learning Curve**: Compared to other machine learning libraries, such as Keras, the learning curve is steeper, and this can make it challenging for new practitioners to create their own models outside of given example code.

Now that you have understood what TensorFlow is, the next section will demonstrate how to use the TensorFlow library using Python.

## The TensorFlow Library in Python

TensorFlow can be used in Python by importing certain libraries. You can import libraries in Python using the import statement:



In the preceding command, you have imported the TensorFlow library and used the shorthand tf.

In the next exercise, you will learn how to import the TensorFlow library and check its version so that you can utilize the classes and functions supplied by the library, which is an important and necessary first step when utilizing the library.

## Exercise 1.01: Verifying Your Version of TensorFlow

In this exercise, you will load TensorFlow and check which version is installed on your system.

Perform the following steps:

1. Open a Jupyter notebook to implement this exercise by typing jupyter notebook in the terminal.
2. Import the TensorFlow library by entering the following code in the Jupyter cell:



1. Verify the version of TensorFlow using the following command:

A picture containing graphical user interface

Description automatically generated

This will result in the following output:



As you can see from the preceding output, the version of TensorFlow is 2.7.0.

Note:

The version may vary on your system if you have not set up the environment using the steps provided in Preface.

In this exercise, you successfully imported TensorFlow. You have also checked which version of TensorFlow is installed on your system.

This task can be done for any imported library in Python and is useful for debugging and referencing documentation.

The potential applications of using TensorFlow are numerous, and it has already achieved impressive results, as evidenced by the results from companies such as Airbnb, which uses TensorFlow to classify images on their platform, to GE Healthcare, which uses TensorFlow to identify anatomy on MRIs of the brain. To learn how to create powerful models for your own applications, you first must learn the basic mathematical principles and operations that make up the machine learning models that can be achieved in TensorFlow. The mathematical operations can be intimidating to new users, but a comprehensive understanding of how they operate is key to making performant models.

## Introduction to Tensors

Tensors can be thought of as the core components of ANNs—the input data, output predictions, and weights that are learned throughout the training process are all tensors. Information propagates through a series of linear and nonlinear transformations to turn the input data into predictions. This section demonstrates how to apply linear transformations such as additions, transpositions, and multiplications to tensors. Other linear transformations, such as rotations, reflections, and shears, also exist. However, their applications as they pertain to ANNs are less common.

## Scalars, Vectors, Matrices, and Tensors

Tensors can be represented as multi-dimensional arrays. The number of dimensions a tensor spans is known as the tensor's rank. Tensors with ranks 0, 1, and 2 are used often and have their own names, which are scalars, vectors, and matrices, respectively, although the term tensors can be used to describe each of them. Figure 1.2 shows some examples of tensors of various ranks. From left to right are a scalar, vector, matrix, and a 3-dimensional tensor, where each element represents a different number, and the subscript represents the location of the element in the tensor:

Diagram

Description automatically generated

Figure 1.2: A visual representation of a scalar, vector, matrix, and tensor

The formal definitions of a scalar, vector, matrix, and tensor are as follows:

* **Scalar:** A scalar consists of a single number, making it a zero-dimensional array. It is an example of zero-order tensors. Scalars do not have any axes. For instance, the width of an object is a scalar.
* **Vector:** Vectors are one-dimensional arrays and are an example of first-order tensors. They can be considered lists of values. Vectors have one axis. The size of a given object denoted by the width, height, and depth is an example of a vector field.
* **Matrix:** Matrices are two-dimensional arrays with two axes. They are an example of second-order tensors. Matrices might be used to store the size of several objects. Each dimension of the matrix comprises the size of each object (width, height, depth) and the other matrix dimension is used to differentiate between objects.
* **Tensor:** Tensors are the general entities that encapsulate scalars, vectors, and matrices, although the name is generally reserved for tensors of rank 3 or more. A tensor can be used to store the size of many objects and their locations over time. The first dimension of the matrix comprises the size of each object (width, height, depth), the second dimension is used to differentiate between the objects, and the third dimension describes the location of these objects over time.

Graphical user interface

Description automatically generated

Tensors can be created using the Variable class present in the TensorFlow library and passing in a value representing the tensor. A float or integer can be passed for scalars, a list of floats or integers can be passed for vectors, a nested list of floats or integers for matrices, and so on. The following command demonstrates the use of the Variable class where a list of the intended values for the tensor as well as any other attributes that are required to be explicitly defined are passed:

Graphical user interface, text

Description automatically generated with medium confidence

The resultant Variable object has several attributes that may be commonly called, and these are as follows:

* **dtype**: The datatype of the **Variable** object (for the tensor defined above, the datatype is tf.int32). The default value for this attribute is determined from the values passed.
* **shape**: The number of dimensions and length of each dimension of the **Variable** object (for the tensor defined above, the shape is **[3]**). The default value for this attribute is also determined from the values passed.
* **name**: The name of the **Variable** object (for the tensor defined above, the name of the tensor is defined as **'my\_tensor'**). The default for this attribute is **Variable**.
* **trainable**: This attribute indicates whether the **Variable** object can be updated during model training (for the tensor defined above, the **trainable** parameter is set to **true**). The default for this attribute is true.

Note

You can read more about the attributes of the Variable object here:

[https://www.tensorflow.org/api\_docs/python/tf/Variabl](https://www.tensorflow.org/api_docs/python/tf/Variable)e.

The **shape** attribute of the **Variable** object can be called as follows:



The **shape** attribute gives the shape of the tensor, that is, is it a scalar, vector, matrix, and so on. The output of the preceding command will be **[3]** since the tensor has a single dimension with three values along that dimension.

The rank of a tensor can be determined in TensorFlow using the **rank** function. It can be used by passing the tensor as the single argument to the function and the result will be an integer value:

Text

Description automatically generated

The output of the following command will be a zero-dimensional integer tensor representing the rank of the input. In this case, the rank of **tensor1** will be **1** as the tensor has only one dimension.

In the following exercise, you will learn how to create tensors of various ranks using TensorFlow's **Variable** class.

In a TensorFlow a:

* Rank 0 Tensor is a Scalar
* Rank 1 Tensor is a Vector
* Rank 2 Tensor is a Matrix
* Rank 3 Tensor is a 3-Tensor
* Rank n Tensor is a n-Tensor

## Exercise 1.02: Creating Scalars, Vectors, Matrices, and Tensors in TensorFlow

1. Import the TensorFlow library:



1. Create an **integer** variable using TensorFlow’s **Variable** class and pass 365 to represent the number of day in a year. Also, pass tf.int16 as second argument to ensure that the input number is an integer datatype. Print the result:

Note

The datatype does not have to be explicitly defined. If one is not defined, the datatype will be determined by TensorFlow’s **convert\_to\_tensor** function

A picture containing text

Description automatically generated

This will result in the following output:



Here, you can see the attribute of the variable created, including the name, Variable: 0, the **shape**, **datatype**, and the **NumPy** representation of the tensor.

1. Use TensorFlow's **rank** function to print the rank of the **variable** created:



This will result in the following output:

Text

Description automatically generated with low confidence

You can see that the rank of the integer variable that was created is **0** from the **NumPy** representation of the tensor.

1. Access the integer variable of the **rank** by calling the **numpy** attribute:

Text

Description automatically generated with medium confidence

This will result in the following output:

Graphical user interface

Description automatically generated with medium confidence

The rank of the **scalar** is **0**.

Note

All attributes of the result of the rank function can be called, including the **shape** and **dtype** attributes.

1. Call the **shape** attribute of the integer to find the **shape** of the **tensor**:



This will result in the following output:



The preceding output signifies that the shape of the **tensor** has **no size**, which is representative of a **scalar**.

1. Print the shape of the **scalar** variable as a Python list:



This will result in the following output:



1. Create a **Vector** variable using TensorFlow’s **Variable** class. Pass a list for the vector to represent the score percentage of the student, and pass in a second argument for the datatype as **tf.float32** to ensure that it is a **float** datatype. Print the result:

Graphical user interface

Description automatically generated with medium confidence

This will result in the following output:

A picture containing text

Description automatically generated

You can see that the **shape** and **NumPy** attributes are different from the scalar variable created earlier. The **shape** is now **(3,),** indicating that the **tensor** is **one-dimensional** with **three** elements along that dimension.

1. Print the **rank** of the vector variable using TensorFlow's **rank** function as a **NumPy** variable:



This will result in the following output:



Here, you can see that the rank of the vector variable is **1**, confirming that this variable is **one-dimensional**.

1. Print the **shape** of the vector variable as a Python list:



This will result in the following output:

Graphical user interface

Description automatically generated with low confidence

1. Create a **matrix** variable using TensorFlow's **Variable** class. Pass a list of lists of integers for the matrix to represent the marks of three different students in two different entrance exams. This matrix will have three columns representing the students, and two rows representing the entrance exams. Pass in a second argument for the datatype as **tf.int32** to ensure that it is an integer datatype. Print the result:

Graphical user interface, text

Description automatically generated

This will result in the following output:

A picture containing graphical user interface

Description automatically generated

1. Print the **rank** of the **matrix** variable as a **NumPy** variable:



This will result in the following output:



Here, you can see that the rank of the **matrix** variable is **2**, confirming that this variable is **two-dimensional**.

1. Print the **shape** of the **matrix** variable as a Python list:



This will result in the following output:



1. Create a **tensor** variable using TensorFlow's **Variable** class. Pass in a triple nested list of integers for the tensor to represent the marks for three different students in two different entrance exams, for three universities. Print the result:

Text

Description automatically generated

This will result in the following output:

Text, letter

Description automatically generated

1. Print the **rank** of the **tensor** variable as a **NumPy** variable:

Text

Description automatically generated with medium confidence

This will result in the following output:



Here, you can see that the **rank** of the **tensor** variable is **3**, confirming that this variable is **three-dimensional**.

1. Print the **shape** of the **tensor** variable as a Python list:



This will result in the following output:



In this exercise, you have successfully created tensors of various ranks from political voting data using TensorFlow's **Variable** class. First, you created **scalars**, which are tensors that have a rank of 0. Next, you created **vectors**, which are tensors with a rank of 1. **Matrices** were then created, which are tensors of rank 2. Finally, **tensors** were created that have rank 3 or more. You confirmed the rank of the tensors you created by using TensorFlow's rank function and verified their shape by calling the tensor's shape attribute.

In the next section, you will combine tensors to create new tensors using tensor addition.

## Tensor Addition

Tensors can be added together to create new tensors. You will use the example of matrices in this chapter, but the concept can be extended to tensors with any rank. Matrices may be added to scalars, vectors, and other matrices under certain conditions in a process known as broadcasting. Broadcasting refers to the process of array arithmetic on tensors of different shapes.

Two matrices may be added (or subtracted) together if they have the same shape. For such matrix-matrix addition, the resultant matrix is determined by the element-wise addition of the input matrices. The resultant matrix will therefore have the same shape as the two input matrices. You can define the matrix Z = [Zij] as the matrix sum Z = X + Y, where zij = xij + yij and each element in Z is the sum of the same element in X and Y.

Matrix addition is commutative, which means that the order of X and Y does not matter, that is, X + Y = Y + X. Matrix addition is also associative, which means that the same result is achieved even when the order of additions is different or even if the operation is applied more than once, that is, X + (Y + Z) = (X + Y) + Z.

The same matrix addition principles apply to scalars, vectors, and tensors. An example is shown in the following figure:

Diagram, text

Description automatically generated

Figure 1.6: A visual example of matrix-matrix addition

Scalars can also be added to matrices. Here, each element of the matrix is added to the scalar individually, as shown in Figure 1.7:

Diagram

Description automatically generated with medium confidence

Addition is an important transformation that can be applied to tensors since the transformation occurs so frequently. For example, a common transformation in developing ANNs is to add a bias to a layer. This is when a constant tensor array of the same size of the ANN layer is added to that layer. Therefore, it is important to know how and when this seemingly simple transformation can be applied to tensors.

Tensor addition can be performed in TensorFlow by using the add function and passing in the tensors as arguments, or simply by using the + operator as follows:

Graphical user interface, text, chat or text message

Description automatically generated

In the following exercise, you will perform tensor addition on scalars, vectors, and matrices in TensorFlow.

## Exercise 1.03: Performing Tensor Addition in TensorFlow

The marks scored by three different students at three different universities in entrance Exam A and Exam B are as follows:

Table

Description automatically generated

* Store the total number of marks scored in University X in Exam A
* Store the total number of marks scored in each University in Exam A
* Store the total number of marks scored in each University in Both Exams

Perform the following steps to complete the exercise:

1. Import the TensorFlow library:

A picture containing rectangle

Description automatically generated

1. Create three scalar variables using TensorFlow's Variable class to represent the marks of three students of University X in Exam A:

Graphical user interface, text, application, chat or text message

Description automatically generated

1. Create a new variable to store the total number marks scored in University X in Exam A:



1. Print the result of the sum of the two variables as a NumPy variable:

A picture containing rectangle

Description automatically generated

This will result in the following output:

Graphical user interface, application

Description automatically generated with medium confidence

1. Create three vectors to represent the marks scored in three different Universities in Exam A, each with one row and three columns:

Text, chat or text message

Description automatically generated

1. Create a new variable to store the total marks scored in each University in Exam A:



1. Print the result of the sum of the two variables as a NumPy array:

A picture containing shape

Description automatically generated

This will result in the following output:



1. Verify that the vector addition is as expected by performing the addition of each element of the vector:

A picture containing icon

Description automatically generated

This will result in the following output:

Graphical user interface

Description automatically generated with low confidence

You can see that the + operation on three vectors is simply element-wise addition of the vectors.

1. Create three matrices to store the votes cast for candidates of each political party in each district:

Text

Description automatically generated

1. Verify that the three tensors have the same shape:

A picture containing text

Description automatically generated

This will result in the following output:



1. Create a new variable to store the total marks scored in each University in both Exams:

A picture containing chart

Description automatically generated

1. Print the result of the sum of the two variables as a NumPy array:



This will result in the following output representing the total votes for each candidate and each party across districts:

Text

Description automatically generated with medium confidence

Figure 1.9: The output of the matrix summation as a NumPy variable

1. Verify that the tensor addition is as expected by performing the addition of each element of the vector:

Text

Description automatically generated

This will result in the following output:

Table

Description automatically generated

You can see that the + operation is equivalent to the element-wise addition of the three matrices created.

In this exercise, you successfully performed tensor addition on data representing votes cast for political candidates. The transformation can be applied by using the + operation. You also verified that addition is performed element by element, and that one way to ensure that the transformation is valid is for the tensors to have the same rank and shape.

In the following activity, you will further practice tensor addition in TensorFlow.

## Activity 1.01: Performing Tensor Addition in TensorFlow

You work in an institute that has three branches, each with two students and in each branch, they teach three subjects. You are required to sum the tensors to represent the total marks scored by each student across branches.

Table

Description automatically generated

Figure 1.10: Number of different products sold by each salesperson at different locations

The steps you will take are as follows:

1. Import the TensorFlow library.
2. Create two scalars to represent the total marks scored in Subject A by all students at Branch X using TensorFlow's Variable class. The first variable will have a value of 70 and the second will have a value of 38.
3. Create a new variable as the sum of the scalars and print the result.

You should get the following output:



1. Create a vector with values [70, 79, 51] and a scalar with the value 95 using TensorFlow's Variable class.
2. Create a new variable as the sum of the scalar with the vector to represent the marks of Student 1 at Branch X and print the result.

You should get the following output:



Figure 1.11: The output of the integer-vector summation as a NumPy variable

1. Create three tensors with a rank of 2 representing the marks of each student, subject, and branch using TensorFlow's Variable class. The first tensor will have the value [[70, 79, 51], [38, 40, 59]], the second will have the value [[90, 28, 64], [62, 98, 48]], and the third will have [[39, 33, 55], [45, 97, 47]].
2. Create a new variable as the sum of the matrices and print the result:

Text

Description automatically generated

Figure 1.12: The output of the matrix summation as a NumPy variable

Note

The solution to this activity can be found via this link

In the following section, you will learn how to change a tensor's shape and rank.

## Reshaping

Some operations, such as addition, can only be applied to tensors if they meet certain conditions. Reshaping is one method for modifying the shape of tensors so that such operations can be performed. Reshaping takes the elements of a tensor and rearranges them into a tensor of a different size. A tensor of any size can be reshaped so long as the number of total elements remains the same.

For example, a (4x3) matrix can be reshaped into a (6x2) matrix since they both have a total of 12 elements. The rank, or number, of dimensions, can also be changed in the reshaping process. For instance, a (4x3) matrix that has a rank equal to 2 can be reshaped into a (3x2x2) tensor that has a rank equal to 3. The (4x3) matrix can also be reshaped into a (12x1) vector in which the rank has changed from 2 to 1.

Figure 1.13 illustrates tensor reshaping. On the left is a tensor with shape (3x2), which can be reshaped to a tensor of shape equal to either (2x3), (6), or (6x1). Here, the number of elements, that is, six, has remained constant, though the shape and rank of the tensor have changed:

Diagram

Description automatically generated

Figure 1.13: Visual representation of reshaping a (3x2) tensor to tensors of different shapes

Tensor reshaping can be performed in TensorFlow by using the reshape function and passing in the tensor and the desired shape of the new tensor as the arguments:

Graphical user interface, text, application

Description automatically generated

Here, a new tensor is created that has the same elements as the original; however, the shape is [3,2] instead of [6].

The next section introduces tensor transposition, which is another method for modifying the shape of a tensor.

## Tensor Transposition

When a tensor is transposed, the elements in the tensor are rearranged in a specific order. The transpose operation is usually denoted as a T superscript on the tensor. The new position of each element in the tensor can be determined by (x12…k)T = xk…21. For a matrix or tensor of rank equal to 2, the rows become the columns and vice versa. An example of matrix transposition is shown in Figure 1.14. Tensors of any rank can be transposed, and often the shape changes as a result:

A picture containing text, antenna

Description automatically generated

Figure 1.14: A visual representation of tensor transposition on a (3x2) matrix

The following diagram shows the matrix transposition properties of matrices A and B:

Text, letter

Description automatically generated

Figure 1.15: Tensor transposition properties where X and Y are tensors

A tensor is said to be symmetrical if the transpose of a tensor is equivalent to the original tensor.

Tensor transposition can be performed in TensorFlow by using its transpose function and passing in the tensor as the only argument:

Text

Description automatically generated with medium confidence

When transposing a tensor, there is only one possible result; however, reshaping a tensor has multiple possible results depending on the desired shape of the output.

In the following exercise, reshaping and transposition are demonstrated on tensors using TensorFlow.

## Exercise 1.04: Performing Tensor Reshaping and Transposition in TensorFlow

In this exercise, you will learn how to perform tensor reshaping and transposition using the TensorFlow library.

Perform the following steps:

1. Import the TensorFlow library and create a matrix with two rows and four columns using TensorFlow's Variable class:

A picture containing chart

Description automatically generated

1. Verify the shape of the matrix by calling the shape attribute of the matrix as a Python list:

This will result in the following output:

You see that the shape of the matrix is [2, 4].

1. Use TensorFlow's reshape function to change the matrix to a matrix with four rows and two columns by passing in the matrix and the desired new shape:

You should get the following output:

Figure 1.16: The reshaped matrix

1. Verify the shape of the reshaped matrix by calling the shape attribute as a Python list:

This will result in the following output:

Here, you can see that the shape of the matrix has changed to your desired shape, [4,2].

1. Use TensorFlow's reshape function to convert the matrix into a matrix with one row and eight columns. Pass the matrix and the desired new shape as parameters to the reshape function:

You should get the following output:

1. Verify the shape of the reshaped matrix by calling the shape attribute as a Python list:

This will result in the following output:

The preceding output confirms the shape of the reshaped matrix as [1, 8].

1. Use TensorFlow's reshape function to convert the matrix into a matrix with eight rows and one column, passing the matrix and the desired new shape as parameters to the reshape function:

You should get the following output:

Figure 1.17: Reshaped matrix of shape (8, 1)

1. Verify the shape of the reshaped matrix by calling the shape attribute as a Python list:

This will result in the following output:

The preceding output confirms the shape of the reshaped matrix as [8, 1].

1. Use TensorFlow's reshape function to convert the matrix to a tensor of size 2x2x2. Pass the matrix and the desired new shape as parameters to the reshape function:

You should get the following output:

Figure 1.18: Reshaped matrix of shape (2, 2, 2

1. Verify the shape of the reshaped matrix by calling the shape attribute as a Python list:

This will result in the following output:

The preceding output confirms the shape of the reshaped matrix as [2, 2, 2].

1. Verify that the rank has changed using TensorFlow's rank function and print the result as a NumPy variable:

This will result in the following output:

1. Use TensorFlow's transpose function to convert the matrix of size 2X4 to a matrix of size 4x2:

You should get the following output:

Figure 1.19: Transposed matrix

1. Verify that the reshape function and the transpose function create different resulting matrices when applied to the given matrix:

Figure 1.20: Verification that transposition and reshaping produce different results

1. Use TensorFlow's transpose function to transpose the reshaped matrix in *step 9*:

This will result in the following output:

Figure 1.21: The output of the transposition of the reshaped tensor

This result shows how the resulting tensor appears after reshaping and transposing a tensor

In this exercise, you have successfully modified the shape of a tensor either through reshaping or transposition. You studied how the shape and rank of the tensor changes following the reshaping and transposition operation.

In the following activity, you will test your knowledge on how to reshape and transpose tensors using TensorFlow.

## Activity 1.02: Performing Tensor Reshaping and Transposition in TensorFlow

In this activity, you are required to simulate the grouping of 24 school children for class projects. The dimensions of each resulting reshaped or transposed tensor will represent the size of each group.

Perform the following steps:

1. Import the TensorFlow library.
2. Create a one-dimensional tensor with 24 monotonically increasing elements using the Variable class to represent the IDs of the school children. Verify the shape of the matrix.

You should get the following output:



1. Reshape the matrix so that it has 12 rows and 2 columns using TensorFlow’s reshape function representing 12 pairs of school children. Verify the shape of the new matrix.

You should get the following output:



1. Reshape the original matrix so that it has a shape of 3x4x3 using TensorFlow’s reshape function representing 3 groups of 4 sets of pairs of school children. Verify the shape of the new tensor.

You should get the following output:



1. Verify that the rank of this new tensor is 3.
2. Transpose the tensor created is step 3 to represent 2 groups of 12 students using TensorFlow’s transpose function. Verify the shape of the new tensor.

You should get the following output:



Note

This solution to this activity can be found via this link.

In this section, you were introduced to some of the basic components of ANNs-tensors. You also learned about some basic manipulation of tensors, such as addition, transposition, and reshaping. You implemented these concepts by using functions in the TensorFlow library.

In the next topic, you will extend your understanding of linear transformations by covering another important transformation related to ANNs—tensor multiplication.

## Tensor Multiplication

Tensor multiplication is another fundamental operation that is used frequently in the process of building and training ANNs since information propagates through the network from the inputs to the result via a series of additions and multiplications. While the rules for addition are simple and intuitive, the rules for tensors are more complex. Tensor multiplication involves more than simple element-wise multiplication of the elements. Rather, a more complicated procedure is implemented that involves the dot product between the entire rows/columns of each of the tensors to calculate each element of the resulting tensor. This section will explain how multiplication works for two-dimensional tensors or matrices. However, tensors of higher orders can also be multiplied.

Given a matrix, X = [xij]m x n, and another matrix, Y = [yij]n x p, the product of the two matrices is Z = XY = [zij]m x p, and each element, zij, is defined element-wise as Formula. The shape of the resultant matrix is the same as the outer dimensions of the matrix product, or the number of rows of the first matrix and the number of columns of the second matrix. For the multiplication to work, the inner dimensions of the matrix product must match, or the number of columns in the first matrix and the number of columns in the second matrix must correspond.

The concept of inner and outer dimensions of matrix multiplication is shown in the following diagram, where X represents the first matrix and Y represents the second matrix:

Diagram, schematic

Description automatically generated

Figure 1.22: A visual representation of inner and outer dimensions in matrix multiplication

Unlike matrix addition, matrix multiplication is not commutative, which means that the order of the matrices in the product matters:

Text

Description automatically generated

Figure 1.23: Matrix multiplication is non-commutative

For example, say you have the following two matrices:

A picture containing text, clock, watch

Description automatically generated

Figure 1.24: Two matrices, X and Y

One way to construct the product is to have matrix X first, multiplied by Y:

Text

Description automatically generated

Figure 1.25: Visual representation of matrix X multiplied by Y, X•Y

This results in a 2x2 matrix. Another way to construct the product is to have Y first, multiplied by X:

Text, letter

Description automatically generated

Figure 1.26: Visual representation of matrix Y multiplied by X, Y•X

Here you can see that the matrix formed from the product YX is a 3x3 matrix and is very different from the matrix formed from the product XY.

Tensor multiplication can be performed in TensorFlow by using the matmul function and passing in the tensors to be multiplied in the order in which they are to be multiplied as the arguments:

Text

Description automatically generated

Tensor multiplication can also be achieved by using the @ operator as follows:

A picture containing graphical user interface

Description automatically generated

Scalar-tensor multiplication is much more straightforward and is simply the product of every element in the tensor multiplied by the scalar so that λX = [λxij…k], where λ is a scalar and X is a tensor.

Scalar multiplication can be achieved in TensorFlow either by using the matmul function or by using the \* operator:

Graphical user interface, text

Description automatically generated

In the following exercise, you will perform tensor multiplication using the TensorFlow library.

## Exercise 1.05: Performing Tensor Multiplication in TensorFlow

In this exercise, you will perform tensor multiplication in TensorFlow using TensorFlow's matmul function and the @ operator. In this exercise, you will use the example of data from a sandwich retailer representing the ingredients of various Pizzas and the costs of different ingredients. You will use matrix multiplication to determine the costs of each sandwich.

**Pizza’s** **recipe:**

Table

Description automatically generated

Figure 1.27: Pizza recipe

**Ingredient details**

Table

Description automatically generated

Figure 1.28 Ingredient details

**Sales projects:**

Table

Description automatically generated

Figure 1.29: Sales projections

Perform the following steps:

1. Import the TensorFlow library:

A picture containing text

Description automatically generated

1. Create a matrix representing the different sandwich recipes, with the rows representing the three different sandwich offerings and the columns representing the combination and number of the five different ingredients using the Variable class:

A picture containing text

Description automatically generated

You should get the following output:

Diagram

Description automatically generated with medium confidence

Figure 1.30: Matrix representing the number of ingredients needed to make sandwiches

1. Verify the shape of the matrix by calling the shape attribute of the matrix as a Python list:



This will result in the following output:



1. Create a second matrix representing the cost and weight of each individual ingredient in which the rows represent the five ingredients, and the columns represent the cost and weight of the ingredients in grams:

Text

Description automatically generated with medium confidence

You should get the following output:

Text, letter

Description automatically generated

Figure 1.31: A matrix representing the cost and weight of each ingredient

1. Use TensorFlow's matmul function to perform the matrix multiplication of matrix1 and matrix2:

Text

Description automatically generated with medium confidence

This will result in the following output:

A picture containing diagram

Description automatically generated

Figure 1.32: The output of the matrix multiplication

1. Create a matrix to represent the sales projections of five different stores for each of the three sandwiches:

Text

Description automatically generated

1. Multiply matrix3 by the result of the matrix multiplication of matrix1 and matrix2 to give the expected cost and weight for each of the five stores:

Text

Description automatically generated with medium confidence

This will result in the following output:

Text

Description automatically generated

Figure 1.33: The output of matrix multiplication

The resulting tensor from the multiplication shows the expected cost of sandwiches and the expected weight of the total ingredients for each of the stores

In this exercise, you have successfully learned how to perform matrix multiplication in TensorFlow using several operators. You used TensorFlow's matmul function, as well as the shorthand @ operator. Each will perform the multiplication; however, the matmul function has several different arguments that can be passed into the function that make it more flexible.

Note

You can read more about the matmul function here: <https://www.tensorflow.org/api_docs/python/tf/linalg/matmul>.

In the next section, you will explore some other mathematical concepts that are related to ANNs. You will explore forward and backpropagation, as well as activation functions.

## Optimization

In this section, you will learn about some optimization approaches that are fundamental to training machine learning models. Optimization is the process by which the weights of the layers of an ANN are updated such that the error between the predicted values of the ANN and the true values of the training data is minimized.

## Forward Propagation

Forward propagation is the process by which information propagates through ANNs. Operations such as a series of tensor multiplications and additions occur at each layer of the network until the final output. Forward propagation is explained in Figure 1.37, showing a single hidden layer ANN. The input data has two features, while the output layer has a single value for each input record.

The weights and biases for the hidden layer and output are shown as matrices and vectors with the appropriate indexes. For the hidden layer, the number of rows in the weight matrix is equal to the number of features of the input, and the number of columns is equal to the number of units in the hidden layer. Therefore, W1 has two rows and three columns because the input, X, has two features. Likewise, W2 has three rows and one column, the hidden layer has three units, and the output has the size one. The bias, however, is always a vector with a size equal to the number of nodes in that layer and is added to the product of the input and weight matrix.

Diagram

Description automatically generated

Figure 1.34: A single-layer artificial neural network

The steps to perform forward propagation are as follows:

1. X is the input to the network and the input to the hidden layer. First, the input matrix, X, is multiplied by the weight matrix for the hidden layer, W1, and then the bias, b1, is added:

z1 = X\*W1 + b1

1. Here is an example of what the shape of the resulting tensor will be after the operation. If the input is size nX2, where n is the number of input examples, W1 is of size 2X3, and b1 is of size 1X3, the resulting matrix, z1, will have a size of nX3.
2. z1 is the output of the hidden layer, which is the input for the output layer. Next, the output of the hidden layer is the input matrix multiplied by the weight matrix for the output layer, W2, and the bias, b2, is added:

Y = z1 \* W2 + b2

To understand the shape of the resulting tensor, consider the following example. If the input to the output layer, z1, is of size nX3, W2 is of size 3X1, and b1 is of size 1X1, the resulting matrix, Y, will have a size of nX1, representing one result for each training example.

The total number of parameters in this model is equal to the sum of the number of elements in W1, W2, b1, and b2. Therefore, the number of parameters can be calculated by summing the elements in each of the parameters in weight matrices and biases, which is equal to 6 + 3 + 3 + 1 = 13. These are the parameters that need to be learned in the process of training the ANN.

Following the forward propagation step, you must evaluate your model and compare it to the real target values. This is achieved using a loss function. Mean squared error, that is, the mean value of the squared difference between true and predicted values, is one of the examples of the loss function of the regression task. Once the loss is calculated, the weights must be updated to reduce the loss, and the amount and direction that the weights should be updated are found using backpropagation.

## Backpropagation

Backpropagation is the process of determining the derivative of the loss with respect to the model parameter. The loss is calculated by applying the loss function to the predicted outputs as follows:

loss = L(y\_predicted)

The derivative of the loss with respect to the model parameters will inform you if increasing or decreasing the model parameter will result in increasing or decreasing the loss. The process of backpropagation is achieved by applying the chain rule of calculus from the output layer to the input layer of a neural network, at each layer computing the derivatives of the loss function with respect to the model parameters.

The chain rule of calculus is a technique used to compute the derivative of a composite function via intermediate functions. A generalized version of the function can be written as follows:

dz/dx = dz/dy \* dy/dx

Here, dz/dx is the composite function and y is the intermediate function. In the case of ANNs, the composite function is the loss as a function of the model parameters and the intermediate functions represent the hidden layers. Therefore, the derivative of the loss with respect to the model parameters can be computed by multiplying the derivative of the loss with respect to the predicted output by the derivative of the predicted output with respect to the model parameters.

In the next section, you will learn how the weight parameters are updated given the derivatives of the loss function with respect to each of the weights so that the loss is minimized.

## Learning Optimal Parameters

In this section, you will see how optimal weights are iteratively chosen. You know that forward propagation transfers information through the network via a series of tensor additions and multiplications, and that backpropagation is the process of understanding the change in loss with respect to each model weight. The next step is to use the results from backpropagation to update the weights so that they reduce the error according to the loss function. This process is known as learning the parameters and is achieved using an optimization algorithm. A common optimization algorithm often utilized is called **gradient descent**.

In learning the optimal parameters, you apply the optimization algorithm until a minimum in the loss function is reached. You usually stop after a given number of steps or when there is a negligible change in the loss function. If you plot the loss as a function of each model parameter, the shape of the loss function resembles a convex shape, having only one minimum, and it is the goal of the optimization function to find this minimum.

The following figure shows the loss function of a particular feature:

A picture containing diagram

Description automatically generated

Figure 1.35: A visual representation of the gradient descent algorithm finding the optimal parameter to minimize the loss

This is achieved, first, by randomly setting parameters for each weight, indicated by p1 in the diagram. The loss is then calculated for that model parameter, l1. The backpropagation step determines the derivative of the loss with respect to the model parameter and will determine in which direction the model should be updated. The next model parameter, p2, is equal to the current model parameter minus the learning rate (α) multiplied by the derivative value. The learning rate is a hyperparameter that is set before the model training process. By multiplying by the derivative value, larger steps will be taken when the parameter is far from the minimum where the absolute value for the derivative is larger. The loss, l2, is then calculated and the process continues until the minimum loss is reached, lm, with the optimal parameter, pm.

To summarize, these are the iterative steps that the optimization algorithm performs to find the optimal parameters:

1. Use forward propagation and current parameters to predict the outputs for the entire dataset.
2. Apply the loss function to compute the loss over all the examples from the predicted output.
3. Use backpropagation to compute the derivatives of the loss with respect to the weights and biases at each layer.
4. Update the weights and biases using the derivative values and the learning rate.

## Optimizers in TensorFlow

There are several different optimizers readily available within TensorFlow. Each is based on a different optimization algorithm that aims to reach a global minimum for the loss function. They are all based on the gradient descent algorithm, although they differ slightly in implementation. The available optimizers in TensorFlow include the following:

* **Stochastic Gradient Descent** (**SGD**): The SGD algorithm applies gradient descent to small batches of training data. A momentum parameter is also available when using the optimizer in TensorFlow that applies exponential smoothing to the computed gradient to speed up training.
* **Adam**: This optimization is an SGD method that is based on the continuous adaptive estimation of first and second-order moments.
* **Root Mean Squared Propagation** (**RMSProp**): This is an unpublished, adaptive learning rate optimizer. RMSprop divides the learning rate by an average of the squared gradients when finding the loss minimum after each step, which results in a learning rate that exponentially decays.
* **Adagrad**: This optimizer has parameter-specific learning rates that are updated depending on how frequently the parameter is updated during the training process. As the parameter receives more updates, each subsequent update is smaller in value.

The choice of optimizer will affect training time and model performance. Each optimizer also has hyperparameters, such as the initial learning rate, that must be selected before training, and tuning of these hyperparameters will also affect training time and model performance. While other optimizers available in TensorFlow are not explicitly stated here (and can be found here: <https://www.tensorflow.org/api_docs/python/tf/keras/optimizers>), those stated above perform well both in terms of training time and model performance and are a safe first choice when selecting an optimizer for your model. The optimizers available in TensorFlow are located in the tf.optimizers module; for example, an Adam optimizer with a learning rate equal to 0.001 can be initialized as follows:



In this topic, you have seen the steps taken in achieving gradient descent to compute the optimal parameters for model training. In gradient descent, every single training example is used to learn the parameters. However, when working with large volume datasets, such as with images and audio, you will often work in batches and make updates after learning from each batch. When using gradient descent on batch data, the algorithm is known as SGD. The SGD optimizer, along with a suite of other performant optimizers, is readily available in TensorFlow, including the Adam, RMSProp, and Adagrad optimizers, and more.

In the next section, you will explore different activation functions, which are generally applied to the output of each layer.

## Activation functions

Activation functions are mathematical functions that are generally applied to the outputs of ANN layers to limit or bound the values of the layer. The reason that values may want to be bounded is that without activation functions, the value and corresponding gradients can either explode or vanish, thereby making the results unusable. This is because the final value is the cumulative product of the values from each subsequent layer. As the number of layers increases, the likelihood of values and gradients exploding to infinity or vanishing to zero increases. This concept is known as the exploding and vanishing gradient problem. Deciding whether a node in a layer should be activated is another use of activation functions, hence their name. Common activation functions and their visual representation in Figure 1.36 are as follows:

* **Step** function: The value is non-zero if it is above a certain threshold, otherwise it is zero. This is shown in *Figure 1.36a*.
* **Linear** function: Formula, which is a scalar multiplication of the input value. This is shown in *Figure 1.36b*.
* **Sigmoid** function: Formula, like a smoothed-out step function with smooth gradients. This activation function is useful for classification since the values are bound from zero to one. This is shown in *Figure 1.36c*.
* **Tanh** function: Formula, which is a scaled version of the sigmoid with steeper gradients around x=0. This is shown in *Figure 1.36d*.
* **ReLU** (**Rectified Linear Unit**) function: Formula, otherwise 0. This is shown in *Figure 1.36e*.
* **ELU** (**Exponential Linear Unit**) function: Formula, otherwise Formula, where Formulais a constant.
* **SELU** (**Scaled Exponential Linear Unit**) function: Formula, otherwise Formula, where Formulaare constants. This is shown in *Figure 1.36f*.
* **Swish** function: Formula. This is shown in *Figure 1.36g*:

Diagram

Description automatically generated

Figure 1.36: A visual representation of the common activation functions

An activation function can be applied to any tensor by utilizing the activation functions in the tf.keras.activations module. For example, a sigmoid activation function can be applied to a tensor as follows:

Now, let's test the knowledge that you have gained so far in the following activity.

## Activity 1.03: Applying Activation Functions

In this activity, you will recall many of the concepts used throughout the chapter as well as apply activation functions to tensors. You will use example data of bike dealership sales, apply these concepts, show the sales records of various salespeople, and highlight those with net positive sales.

**Sales records:**

Table

Description automatically generated

Figure 1.37: Sales records

**Vehicle MSRPs:**

A picture containing table

Description automatically generated

Figure 1.38: Vehicle MSRPs

**Fixed costs:**

Table

Description automatically generated

Figure 1.39: Vehicle MSRPs

Perform the following steps:

1. Import TensorFlow library
2. Create a 3x4 tensor as an input with the values [[-0.013, 0.024, 0.06, 0.022], [0.001, -0.047, 0.039, 0.016], [0.018, 0.030, -0.021, -0.028]]. The rows in this tensor represent the sales of various sales representatives, the columns represent various vehicles available at the dealership, and values represent the average percentage difference from MSRP. The values are positive or negative depending on whether the salesperson was able to sell for more or less than the MSRP.
3. Create a 4x1 weights tensor with the shape 4x1 with the values [[19995.95], [24995.50], [36745.50], [29995.95]] representing the MSRP of the cars.
4. Create a bias tensor of size 3x1 with the values [[-2500.0], [-2500.0], [-2500.0]] representing the fixed costs associated with each salesperson.
5. Matrix multiply the input by the weight to show the average deviation from the MSRP on all cars and add the bias to subtract the fixed costs of the salesperson. Print the result.

A picture containing diagram

Description automatically generated

Figure 1.40: The output of the matrix multiplication

1. Apply a ReLU activation function to highlight the net-positive salespeople and print the result.

You should get the following result:

Text

Description automatically generated with low confidence

Figure 1.41: The output after applying the activation function

Note

The solution to this activity can be found via this link.

In subsequent chapters, you will see how to add activation functions to your ANNs, either between layers or applied directly after a layer when layers are defined. You will learn how to choose which activation functions are most appropriate, which is often by hyperparameter optimization techniques. The activation function is one example of a hyperparameter, a parameter set before the learning process begins, that can be tuned to find the optimal values for model performance.

## SUMMARY

In this chapter, you were introduced to the TensorFlow library. You learned how to use it in the Python programming language. You created the building blocks of ANNs (tensors) with various ranks and shapes, performed linear transformations on tensors using TensorFlow, and implemented addition, reshaping, transposition, and multiplication on tensors—all of which are fundamental for understanding the underlying mathematics of ANNs.

In the next chapter, you will improve your understanding of tensors and learn how to load data of various types and pre-process it such that it is appropriate for training ANNs in TensorFlow. You will work with tabular, visual, and textual data, all of which must be pre-processed differently. By working with visual data (that is, images), you will also learn how to use training data in which the size of the training data cannot fit into memory.