CHAPTER 2

Developing a Machine Learning Model in Python

In terms of the platforms used for machine learning, there are many algorithms and programming languages. However, the Python ecosystem is one of the most dominant and fastest-growing programming languages for machine learning.

Given the popularity and high adoption rate of Python, we will use it as the main programming language throughout the book. This chapter provides an overview of a Python-based machine learning framework. First, we will review the details of Python-based packages used for machine learning, followed by the model development steps in the Python framework.

The steps of model development in Python presented in this chapter serve as the foundation for the case studies presented in the rest of the book. The Python framework can also be leveraged while developing any machine learning–based model in finance.

Why Python?

Some reasons for Python's popularity are as follows:

- High-level syntax (compared to lower-level languages of C, Java, and C++). Applications can be developed by writing fewer lines of code, making Python attractive to beginners and advanced programmers alike.
- Efficient development lifecycle.
- Large collection of community-managed, open-source libraries.
- Strong portability.

The simplicity of Python has attracted many developers to create new libraries for machine learning, leading to strong adoption of Python.

Python Packages for Machine Learning

The main Python packages used for machine learning are highlighted in Figure 2-1.

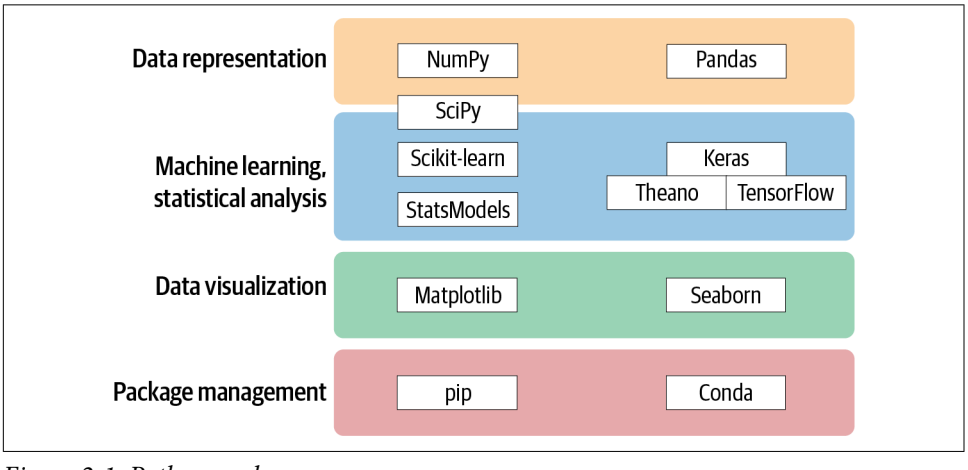


Figure 2-1. Python packages

Here is a brief summary of each of these packages:

NumPy
: Provides support for large, multidimensional arrays as well as an extensive collection of mathematical functions.

Pandas
: A library for data manipulation and analysis. Among other features, it offers data structures to handle tables and the tools to manipulate them.

Matplotlib
: A plotting library that allows the creation of 2D charts and plots.

SciPy
: The combination of NumPy, Pandas, and Matplotlib is generally referred to as SciPy. SciPy is an ecosystem of Python libraries for mathematics, science, and engineering.

Scikit-learn (or sklearn)
: A machine learning library offering a wide range of algorithms and utilities.

StatsModels
: A Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration.

TensorFlow and Theano
: Dataflow programming libraries that facilitate working with neural networks.

Keras
: An artificial neural network library that can act as a simplified interface to TensorFlow/Theano packages.

Seaborn
: A data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

pip and Conda
: These are Python package managers. pip is a package manager that facilitates installation, upgrade, and uninstallation of Python packages. Conda is a package manager that handles Python packages as well as library dependencies outside of the Python packages.

Python and Package Installation

There are different ways of installing Python. However, it is strongly recommended that you install Python through Anaconda. Anaconda contains Python, SciPy, and Scikit-learn.

After installing Anaconda, a Jupyter server can be started locally by opening the machine's terminal and typing in the following code:

```
$jupyter notebook
```

All code samples in this book use Python 3 and are presented in Jupyter notebooks. Several Python packages, especially Scikit-learn and Keras, are extensively used in the case studies.

Steps for Model Development in Python Ecosystem

Working through machine learning problems from end to end is critically important. Applied machine learning will not come alive unless the steps from beginning to end are well defined.

Figure 2-2 provides an outline of the simple seven-step machine learning project template that can be used to jump-start any machine learning model in Python. The

first few steps include exploratory data analysis and data preparation, which are typical data science–based steps aimed at extracting meaning and insights from data. These steps are followed by model evaluation, fine-tuning, and finalizing the model.

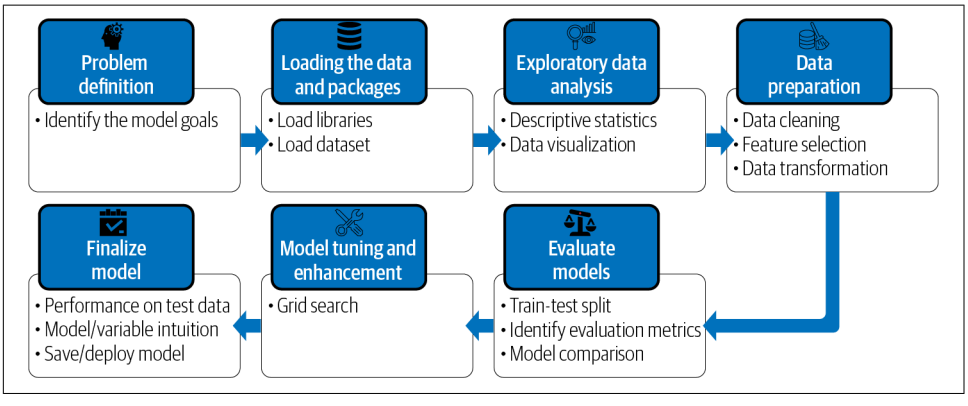


Figure 2-2. Model development steps

All the case studies in this book follow the standard seven-step model development process. However, there are a few case studies in which some of the steps are skipped, renamed, or reordered based on the appropriateness and intuitiveness of the steps.

Model Development Blueprint

The following section covers the details of each model development step with supporting Python code.

1. Problem definition

The first step in any project is defining the problem. Powerful algorithms can be used for solving the problem, but the results will be meaningless if the wrong problem is solved.

The following framework should be used for defining the problem:

1. Describe the problem informally and formally. List assumptions and similar problems.
2. List the motivation for solving the problem, the benefits a solution provides, and how the solution will be used.
3. Describe how the problem would be solved using the domain knowledge.

2. Loading the data and packages

The second step gives you everything needed to start working on the problem. This includes loading libraries, packages, and individual functions needed for the model development.

2.1. Load libraries. A sample code for loading libraries is as follows:

```
# Load libraries
import pandas as pd
from matplotlib import pyplot
```

The details of the libraries and modules for specific functionalities are defined further in the individual case studies.

2.2. Load data. The following items should be checked and removed before loading the data:

- Column headers
- Comments or special characters
- Delimiter

There are many ways of loading data. Some of the most common ways are as follows:

`Load CSV files with Pandas`

```
from pandas import read_csv
filename = 'xyz.csv'
data = read_csv(filename, names=names)
```

`Load file from URL`

```
from pandas import read_csv
url = 'https://goo.gl/vhm1eU'
names = ['age', 'class']
data = read_csv(url, names=names)
```

`Load file using pandas_datareader`

```
import pandas_datareader.data as web

ccy_tickers = ['DEXJPUS', 'DEXUSUK']
idx_tickers = ['SP500', 'DJIA', 'VIXCLS']

stk_data = web.DataReader(stk_tickers, 'yahoo')
ccy_data = web.DataReader(ccy_tickers, 'fred')
idx_data = web.DataReader(idx_tickers, 'fred')
```

3. Exploratory data analysis

In this step, we look at the dataset.

3.1. Descriptive statistics. Understanding the dataset is one of the most important steps of model development. The steps to understanding data include:

1. Viewing the raw data.
2. Reviewing the dimensions of the dataset.
3. Reviewing the data types of attributes.
4. Summarizing the distribution, descriptive statistics, and relationship among the variables in the dataset.

These steps are demonstrated below using sample Python code:

`Viewing the data`

```
set_option('display.width', 100)
dataset.head(1)
```

`Output`

	Age	Sex	Job	Housing	SavingAccounts	CheckingAccount	CreditAmount	Duration	Purpose	Risk
0	67	male	2	own	NaN	little	1169	6	radio/TV	good

`Reviewing the dimensions of the dataset`

```
dataset.shape
```

`Output`

```
(284807, 31)
```

The results show the dimension of the dataset and mean that the dataset has 284,807 rows and 31 columns.

`Reviewing the data types of the attributes in the data`

```
# types
set_option('display.max_rows', 500)
dataset.dtypes
```

`Summarizing the data using descriptive statistics`

```
# describe data
set_option('precision', 3)
dataset.describe()
```

Output

	Age	Job	CreditAmount	Duration
count	1000.000	1000.000	1000.000	1000.000
mean	35.546	1.904	3271.258	20.903
std	11.375	0.654	2822.737	12.059
min	19.000	0.000	250.000	4.000
25%	27.000	2.000	1365.500	12.000
50%	33.000	2.000	2319.500	18.000
75%	42.000	2.000	3972.250	24.000
max	75.000	3.000	18424.000	72.000

3.2. Data visualization. The fastest way to learn more about the data is to visualize it. Visualization involves independently understanding each attribute of the dataset.

Some of the plot types are as follows:

Univariate plots
: Histograms and density plots

Multivariate plots
: Correlation matrix plot and scatterplot

The Python code for univariate plot types is illustrated with examples below:

```
Univariate plot: histogram
```

```
    from matplotlib import pyplot
    dataset.hist(sharex=False, sharey=False, xlabelsize=1, ylabelsize=1,\
    figsize=(10,4))
    pyplot.show()
```

```
Univariate plot: density plot
```

```
    from matplotlib import pyplot
    dataset.plot(kind='density', subplots=True, layout=(3,3), sharex=False,\
    legend=True, fontsize=1, figsize=(10,4))
    pyplot.show()
```

Figure 2-3 illustrates the output.

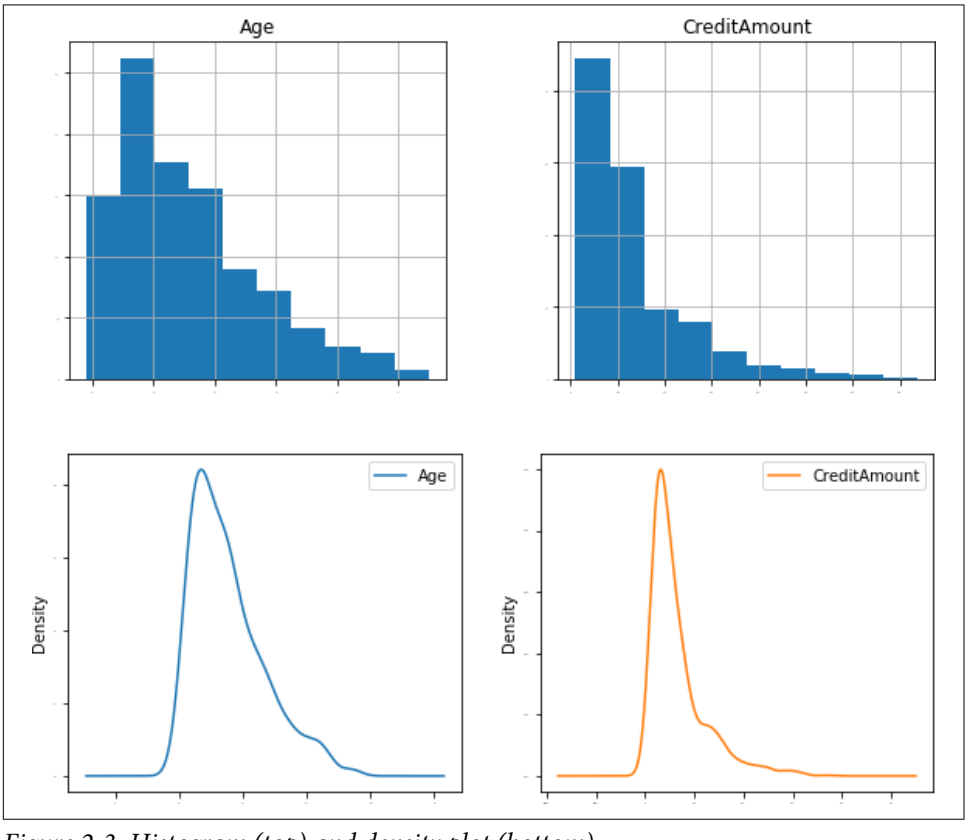


Figure 2-3. Histogram (top) and density plot (bottom)

The Python code for multivariate plot types is illustrated with examples below:

```
Multivariate plot: correlation matrix plot
```

```
    from matplotlib import pyplot
    import seaborn as sns
    correlation = dataset.corr()
    pyplot.figure(figsize=(5,5))
    pyplot.title('Correlation Matrix')
    sns.heatmap(correlation, vmax=1, square=True,annot=True,cmap='cubehelix')
```

```
Multivariate plot: scatterplot matrix
```

```
    from pandas.plotting import scatter_matrix
    scatter_matrix(dataset)
```

Figure 2-4 illustrates the output.

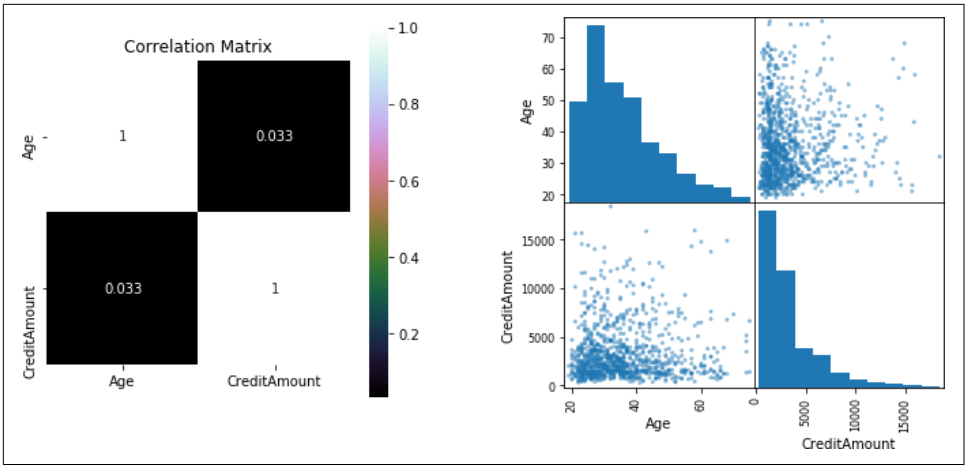


Figure 2-4. Correlation (left) and scatterplot (right)

4. Data preparation

Data preparation is a preprocessing step in which data from one or more sources is cleaned and transformed to improve its quality prior to its use.

4.1. Data cleaning. In machine learning modeling, incorrect data can be costly. Data cleaning involves checking the following:

Validity
: The data type, range, etc.

Accuracy
: The degree to which the data is close to the true values.

Completeness
: The degree to which all required data is known.

Uniformity
: The degree to which the data is specified using the same unit of measure.

The different options for performing data cleaning include:

Dropping "NA" values within data

```
dataset.dropna(axis=0)
```

Filling "NA" with 0

```
dataset.fillna(0)
```

Filling NAs with the mean of the column

```
dataset['col'] = dataset['col'].fillna(dataset['col'].mean())
```

4.2. Feature selection. The data features used to train the machine learning models have a huge influence on the performance. Irrelevant or partially relevant features can negatively impact model performance. Feature selection[1] is a process in which features in data that contribute most to the prediction variable or output are automatically selected.

The benefits of performing feature selection before modeling the data are:

Reduces overfitting[2]
: Less redundant data means fewer opportunities for the model to make decisions based on noise.

Improves performance
: Less misleading data means improved modeling performance.

Reduces training time and memory footprint
: Less data means faster training and lower memory footprint.

The following sample feature is an example demonstrating when the best two features are selected using the `SelectKBest` function under sklearn. The `SelectKBest` function scores the features using an underlying function and then removes all but the *k* highest scoring feature:

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
bestfeatures = SelectKBest( k=5)
fit = bestfeatures.fit(X,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
print(featureScores.nlargest(2,'Score'))  #print 2 best features
```

```
Output

                  Specs      Score
2             Variable1  58262.490
3             Variable2    321.031
```

When features are irrelevant, they should be dropped. Dropping the irrelevant features is illustrated in the following sample code:

```
#dropping the old features
dataset.drop(['Feature1','Feature2','Feature3'],axis=1,inplace=True)
```

1 Feature selection is more relevant for supervised learning models and is described in detail in the individual case studies in Chapters 5 and 6.

2 Overfitting is covered in detail in Chapter 4.

4.3. Data transformation. Many machine learning algorithms make assumptions about the data. It is a good practice to perform the data preparation in such a way that exposes the data in the best possible manner to the machine learning algorithms. This can be accomplished through data transformation.

The different data transformation approaches are as follows:

Rescaling
: When data comprises attributes with varying scales, many machine learning algorithms can benefit from *rescaling* all the attributes to the same scale. Attributes are often rescaled in the range between zero and one. This is useful for optimization algorithms used in the core of machine learning algorithms, and it also helps to speed up the calculations in an algorithm:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = pd.DataFrame(scaler.fit_transform(X))
```

Standardization
: *Standardization* is a useful technique to transform attributes to a standard normal distribution with a mean of zero and a standard deviation of one. It is most suitable for techniques that assume the input variables represent a normal distribution:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(X)
StandardisedX = pd.DataFrame(scaler.fit_transform(X))
```

Normalization
: *Normalization* refers to rescaling each observation (row) to have a length of one (called a unit norm or a vector). This preprocessing method can be useful for sparse datasets of attributes of varying scales when using algorithms that weight input values:

```
from sklearn.preprocessing import Normalizer
scaler = Normalizer().fit(X)
NormalizedX = pd.DataFrame(scaler.fit_transform(X))
```

5. Evaluate models

Once we estimate the performance of our algorithm, we can retrain the final algorithm on the entire training dataset and get it ready for operational use. The best way to do this is to evaluate the performance of the algorithm on a new dataset. Different machine learning techniques require different evaluation metrics. Other than model performance, several other factors such as simplicity, interpretability, and training time are considered when selecting a model. The details regarding these factors are covered in Chapter 4.

5.1. Training and test split. The simplest method we can use to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. We can take our original dataset and split it into two parts: train the algorithm on the first part, make predictions on the second part, and evaluate the predictions against the expected results. The size of the split can depend on the size and specifics of the dataset, although it is common to use 80% of the data for training and the remaining 20% for testing. The differences in the training and test datasets can result in meaningful differences in the estimate of accuracy. The data can easily be split into the training and test sets using the `train_test_split` function available in sklearn:

```
# split out validation dataset for the end
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Identify evaluation metrics. Choosing which metric to use to evaluate machine learning algorithms is very important. An important aspect of evaluation metrics is the capability to discriminate among model results. Different types of evaluation metrics used for different kinds of ML models are covered in detail across several chapters of this book.

5.3. Compare models and algorithms. Selecting a machine learning model or algorithm is both an art and a science. There is no one solution or approach that fits all. There are several factors over and above the model performance that can impact the decision to choose a machine learning algorithm.

Let's understand the process of model comparison with a simple example. We define two variables, *X* and *Y*, and try to build a model to predict *Y* using *X*. As a first step, the data is divided into training and test split as mentioned in the preceding section:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
validation_size = 0.2
seed = 7
X = 2 - 3 * np.random.normal(0, 1, 20)
Y = X - 2 * (X ** 2) + 0.5 * (X ** 3) + np.exp(-X)+np.random.normal(-3, 3, 20)
# transforming the data to include another axis
X = X[:, np.newaxis]
Y = Y[:, np.newaxis]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,\
test_size=validation_size, random_state=seed)
```

We have no idea which algorithms will do well on this problem. Let's design our test now. We will use two models—one linear regression and the second polynomial regression to fit *Y* against *X*. We will evaluate algorithms using the *Root Mean*

Squared Error (RMSE) metric, which is one of the measures of the model performance. RMSE will give a gross idea of how wrong all predictions are (zero is perfect):

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures

model = LinearRegression()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_train)

rmse_lin = np.sqrt(mean_squared_error(Y_train,Y_pred))
r2_lin = r2_score(Y_train,Y_pred)
print("RMSE for Linear Regression:", rmse_lin)

polynomial_features= PolynomialFeatures(degree=2)
x_poly = polynomial_features.fit_transform(X_train)

model = LinearRegression()
model.fit(x_poly, Y_train)
Y_poly_pred = model.predict(x_poly)

rmse = np.sqrt(mean_squared_error(Y_train,Y_poly_pred))
r2 = r2_score(Y_train,Y_poly_pred)
print("RMSE for Polynomial Regression:", rmse)
```

`Output`

```
RMSE for Linear Regression: 6.772942423315028
RMSE for Polynomial Regression: 6.420495127266883
```

We can see that the RMSE of the polynomial regression is slightly better than that of the linear regression.[3] With the former having the better fit, it is the preferred model in this step.

6. Model tuning

Finding the best combination of hyperparameters of a model can be treated as a search problem.[4] This searching exercise is often known as *model tuning* and is one of the most important steps of model development. It is achieved by searching for the best parameters of the model by using techniques such as a *grid search*. In a grid search, you create a grid of all possible hyperparameter combinations and train the model using each one of them. Besides a grid search, there are several other

3 It should be noted that the difference in RMSE is small in this case and may not replicate with a different split of the train/test data.

4 Hyperparameters are the external characteristics of the model, can be considered the model's settings, and are not estimated based on data-like model parameters.

techniques for model tuning, including randomized search, Bayesian optimization, and hyperbrand.

In the case studies presented in this book, we focus primarily on grid search for model tuning.

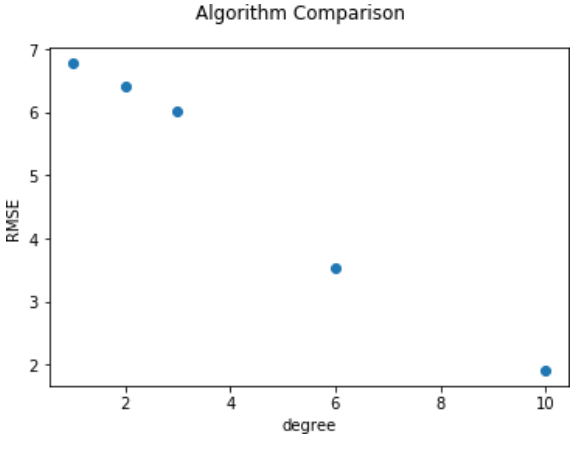
Continuing on from the preceding example, with the polynomial as the best model: next, run a grid search for the model, refitting the polynomial regression with different degrees. We compare the RMSE results for all the models:

```
Deg= [1,2,3,6,10]
results=[]
names=[]
for deg in Deg:
    polynomial_features= PolynomialFeatures(degree=deg)
    x_poly = polynomial_features.fit_transform(X_train)

    model = LinearRegression()
    model.fit(x_poly, Y_train)
    Y_poly_pred = model.predict(x_poly)

    rmse = np.sqrt(mean_squared_error(Y_train,Y_poly_pred))
    r2 = r2_score(Y_train,Y_poly_pred)
    results.append(rmse)
    names.append(deg)
plt.plot(names, results,'o')
plt.suptitle('Algorithm Comparison')
```

Output



The RMSE decreases when the degree increases, and the lowest RMSE is for the model with degree 10. However, models with degrees lower than 10 performed very well, and the test set will be used to finalize the best model.

While the generic set of input parameters for each algorithm provides a starting point for analysis, it may not have the optimal configurations for the particular dataset and business problem.

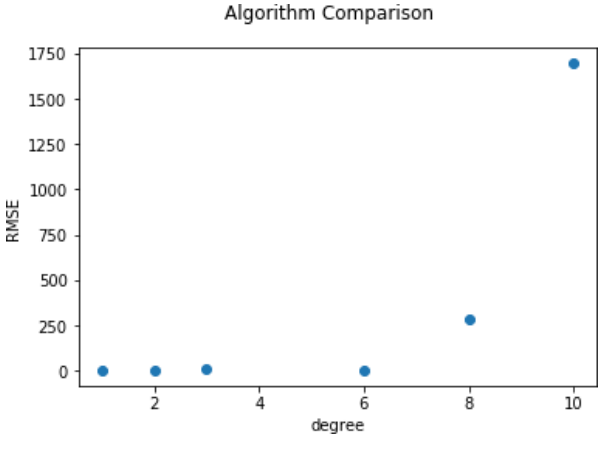
7. Finalize the model

Here, we perform the final steps for selecting the model. First, we run predictions on the test dataset with the trained model. Then we try to understand the model intuition and save it for further usage.

7.1. Performance on the test set. The model selected during the training steps is further evaluated on the test set. The test set allows us to compare different models in an unbiased way, by basing the comparisons in data that were not used in any part of the training. The test results for the model developed in the previous step are shown in the following example:

```
Deg= [1,2,3,6,8,10]
for deg in Deg:
    polynomial_features= PolynomialFeatures(degree=deg)
    x_poly = polynomial_features.fit_transform(X_train)
    model = LinearRegression()
    model.fit(x_poly, Y_train)
    x_poly_test = polynomial_features.fit_transform(X_test)
    Y_poly_pred_test = model.predict(x_poly_test)
    rmse = np.sqrt(mean_squared_error(Y_test,Y_poly_pred_test))
    r2 = r2_score(Y_test,Y_poly_pred_test)
    results_test.append(rmse)
    names_test.append(deg)
plt.plot(names_test, results_test,'o')
plt.suptitle('Algorithm Comparison')
```

Output



In the training set we saw that the RMSE decreases with an increase in the degree of polynomial model, and the polynomial of degree 10 had the lowest RMSE. However, as shown in the preceding output for the polynomial of degree 10, although the training set had the best results, the results in the test set are poor. For the polynomial of degree 8, the RMSE in the test set is relatively higher. The polynomial of degree 6 shows the best result in the test set (although the difference is small compared to other lower-degree polynomials in the test set) as well as good results in the training set. For these reasons, this is the preferred model.

In addition to the model performance, there are several other factors to consider when selecting a model, such as simplicity, interpretability, and training time. These factors will be covered in the upcoming chapters.

7.2. Model/variable intuition. This step involves considering a holistic view of the approach taken to solve the problem, including the model's limitations as it relates to the desired outcome, the variables used, and the selected model parameters. Details on model and variable intuition regarding different types of machine learning models are presented in the subsequent chapters and case studies.

7.3. Save/deploy. After finding an accurate machine learning model, it must be saved and loaded in order to ensure its usage later.

Pickle is one of the packages for saving and loading a trained model in Python. Using pickle operations, trained machine learning models can be saved in the *serialized* format to a file. Later, this serialized file can be loaded to *de-serialize* the model for its usage. The following sample code demonstrates how to save the model to a file and load it to make predictions on new data:

```
# Save Model Using Pickle
from pickle import dump
from pickle import load
# save the model to disk
filename = 'finalized_model.sav'
dump(model, open(filename, 'wb'))
# load the model from disk
loaded_model = load(filename)
```

In recent years, frameworks such as AutoML have been built to automate the maximum number of steps in a machine learning model development process. Such frameworks allow the model developers to build ML models with high scale, efficiency, and productivity. Readers are encouraged to explore such frameworks.

Chapter Summary

Given its popularity, rate of adoption, and flexibility, Python is often the preferred language for machine learning development. There are many available Python packages to perform numerous tasks, including data cleaning, visualization, and model development. Some of these key packages are Scikit-learn and Keras.

The seven steps of model development mentioned in this chapter can be leveraged while developing any machine learning–based model in finance.

Next Steps

In the next chapter, we will cover the key algorithm for machine learning—the artificial neural network. The artificial neural network is another building block of machine learning in finance and is used across all types of machine learning and deep learning algorithms.

CHAPTER 3
Artificial Neural Networks

There are many different types of models used in machine learning. However, one class of machine learning models that stands out is artificial neural networks (ANNs). Given that artificial neural networks are used across all machine learning types, this chapter will cover the basics of ANNs.

ANNs are computing systems based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

Deep learning involves the study of complex ANN-related algorithms. The complexity is attributed to elaborate patterns of how information flows throughout the model. Deep learning has the ability to represent the world as a nested hierarchy of concepts, with each concept defined in relation to a simpler concept. Deep learning techniques are extensively used in reinforcement learning and natural language processing applications that we will look at in Chapters 9 and 10.

We will review detailed terminology and processes used in the field of ANNs[1] and cover the following topics:

- Architecture of ANNs: Neurons and layers
- Training an ANN: Forward propagation, backpropagation and gradient descent
- Hyperparameters of ANNs: Number of layers and nodes, activation function, loss function, learning rate, etc.
- Defining and training a deep neural network–based model in Python
- Improving the training speed of ANNs and deep learning models

ANNs: Architecture, Training, and Hyperparameters

ANNs contain multiple neurons arranged in layers. An ANN goes through a training phase by comparing the modeled output to the desired output, where it learns to recognize patterns in data. Let us go through the components of ANNs.

Architecture

An ANN architecture comprises neurons, layers, and weights.

Neurons

The building blocks for ANNs are neurons (also known as artificial neurons, nodes, or perceptrons). Neurons have one or more inputs and one output. It is possible to build a network of neurons to compute complex logical propositions. Activation functions in these neurons create complicated, nonlinear functional mappings between the inputs and the output.[2]

As shown in Figure 3-1, a neuron takes an input (x_1, x_2…x_n), applies the learning parameters to generate a weighted sum (z), and then passes that sum to an activation function (f) that computes the output $f(z)$.

1 Readers are encouraged to refer to the book *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio (MIT Press) for more details on ANN and deep learning.

2 Activation functions are described in detail later in this chapter.

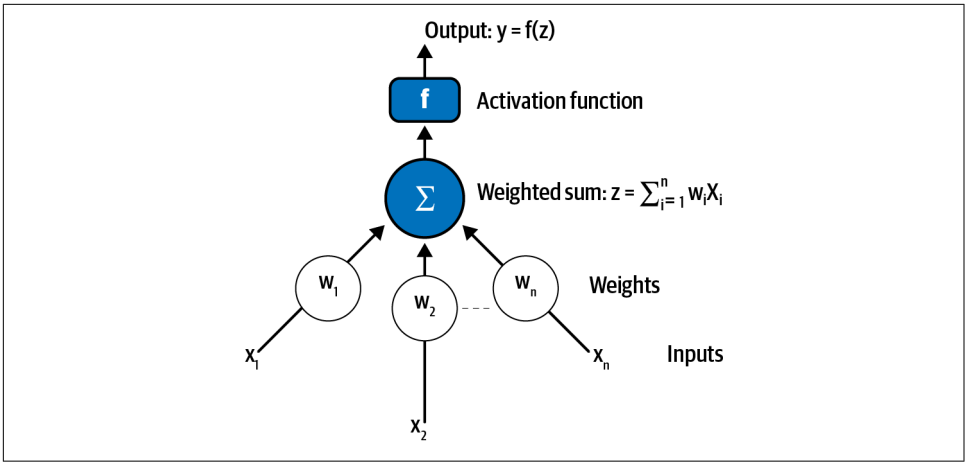


Figure 3-1. An artificial neuron

Layers

The output *f(z)* from a single neuron (as shown in Figure 3-1) will not be able to model complex tasks. So, in order to handle more complex structures, we have multiple layers of such neurons. As we keep stacking neurons horizontally and vertically, the class of functions we can get becomes increasing complex. Figure 3-2 shows an architecture of an ANN with an input layer, an output layer, and a hidden layer.

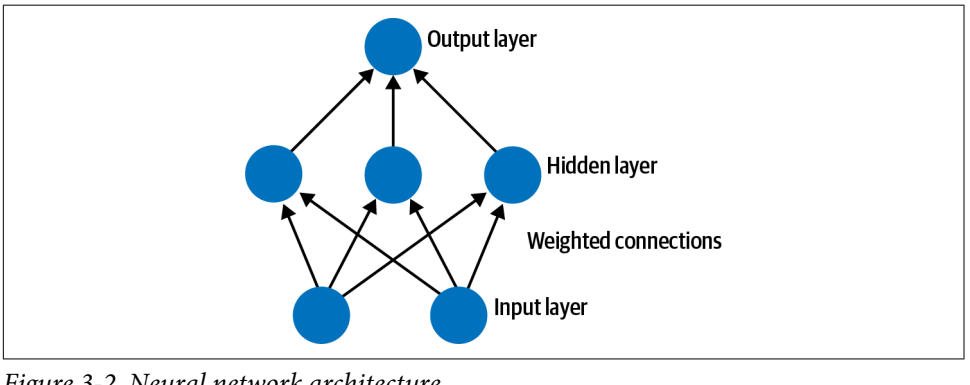


Figure 3-2. Neural network architecture

Input layer. The input layer takes input from the dataset and is the exposed part of the network. A neural network is often drawn with an input layer of one neuron per input value (or column) in the dataset. The neurons in the input layer simply pass the input value though to the next layer.

Hidden layers. Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.

A multilayer ANN is capable of solving more complex machine learning–related tasks due to its hidden layer(s). Given increases in computing power and efficient libraries, neural networks with many layers can be constructed. ANNs with many hidden layers (more than three) are known as *deep neural networks*. Multiple hidden layers allow deep neural networks to learn features of the data in a so-called feature hierarchy, because simple features recombine from one layer to the next to form more complex features. ANNs with many layers pass input data (features) through more mathematical operations than do ANNs with few layers and are therefore more computationally intensive to train.

Output layer. The final layer is called the output layer; it is responsible for outputting a value or vector of values that correspond to the format required to solve the problem.

Neuron weights

A neuron weight represents the strength of the connection between units and measures the influence the input will have on the output. If the weight from neuron one to neuron two has greater magnitude, it means that neuron one has a greater influence over neuron two. Weights near zero mean changing this input will not change the output. Negative weights mean increasing this input will decrease the output.

Training

Training a neural network basically means calibrating all of the weights in the ANN. This optimization is performed using an iterative approach involving forward propagation and backpropagation steps.

Forward propagation

Forward propagation is a process of feeding input values to the neural network and getting an output, which we call *predicted value*. When we feed the input values to the neural network's first layer, it goes without any operations. The second layer takes values from the first layer and applies multiplication, addition, and activation operations before passing this value to the next layer. The same process repeats for any subsequent layers until an output value from the last layer is received.

Backpropagation

After forward propagation, we get a predicted value from the ANN. Suppose the desired output of a network is Y and the predicted value of the network from forward propagation is Y'. The difference between the predicted output and the desired output ($Y–Y'$) is converted into the loss (or cost) function $J(w)$, where w represents the weights in ANN.[3] The goal is to optimize the loss function (i.e., make the loss as small as possible) over the training set.

The optimization method used is *gradient descent*. The goal of the gradient descent method is to find the gradient of $J(w)$ with respect to w at the current point and take a small step in the direction of the negative gradient until the minimum value is reached, as shown in Figure 3-3.

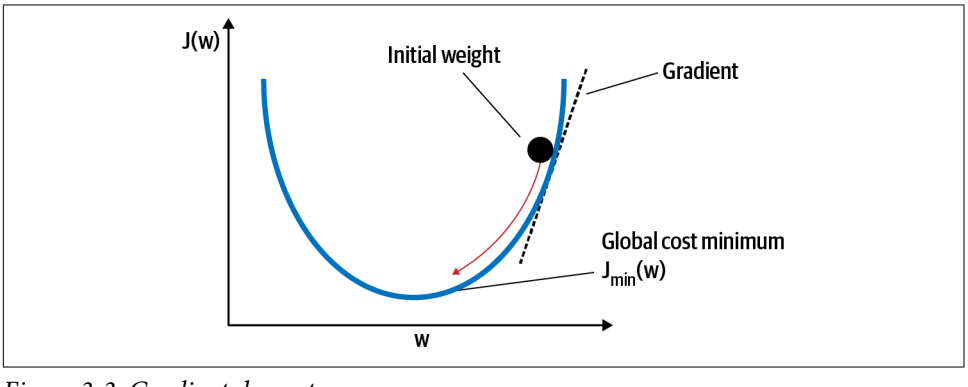


Figure 3-3. Gradient descent

In an ANN, the function $J(w)$ is essentially a composition of multiple layers, as explained in the preceding text. So, if layer one is represented as function $p()$, layer two as $q()$, and layer three as $r()$, then the overall function is $J(w)=r(q(p()))$. w consists of all weights in all three layers. We want to find the gradient of $J(w)$ with respect to each component of w.

Skipping the mathematical details, the above essentially implies that the gradient of a component w in the first layer would depend on the gradients in the second and third layers. Similarly, the gradients in the second layer will depend on the gradients in the third layer. Therefore, we start computing the derivatives in the reverse direction, starting with the last layer, and use backpropagation to compute gradients of the previous layer.

3 There are many available loss functions discussed in the next section. The nature of our problem dictates our choice of loss function.

Overall, in the process of backpropagation, the model error (difference between predicted and desired output) is propagated back through the network, one layer at a time, and the weights are updated according to the amount they contributed to the error.

Almost all ANNs use gradient descent and backpropagation. Backpropagation is one of the cleanest and most efficient ways to find the gradient.

Hyperparameters

Hyperparameters are the variables that are set before the training process, and they cannot be learned during training. ANNs have a large number of hyperparameters, which makes them quite flexible. However, this flexibility makes the model tuning process difficult. Understanding the hyperparameters and the intuition behind them helps give an idea of what values are reasonable for each hyperparameter so we can restrict the search space. Let's start with the number of hidden layers and nodes.

Number of hidden layers and nodes

More hidden layers or nodes per layer means more parameters in the ANN, allowing the model to fit more complex functions. To have a trained network that generalizes well, we need to pick an optimal number of hidden layers, as well as of the nodes in each hidden layer. Too few nodes and layers will lead to high errors for the system, as the predictive factors might be too complex for a small number of nodes to capture. Too many nodes and layers will overfit to the training data and not generalize well.

There is no hard-and-fast rule to decide the number of layers and nodes.

The number of hidden layers primarily depends on the complexity of the task. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and a huge amount of training data. For the majority of the problems, we can start with just one or two hidden layers and then gradually ramp up the number of hidden layers until we start overfitting the training set.

The number of hidden nodes should have a relationship to the number of input and output nodes, the amount of training data available, and the complexity of the function being modeled. As a rule of thumb, the number of hidden nodes in each layer should be somewhere between the size of the input layer and the size of the output layer, ideally the mean. The number of hidden nodes shouldn't exceed twice the number of input nodes in order to avoid overfitting.

Learning rate

When we train ANNs, we use many iterations of forward propagation and backpropagation to optimize the weights. At each iteration we calculate the derivative of the

loss function with respect to each weight and subtract it from that weight. The learning rate determines how quickly or slowly we want to update our weight (parameter) values. This learning rate should be high enough so that it converges in a reasonable amount of time. Yet it should be low enough so that it finds the minimum value of the loss function.

Activation functions

Activation functions (as shown in Figure 3-1) refer to the functions used over the weighted sum of inputs in ANNs to get the desired output. Activation functions allow the network to combine the inputs in more complex ways, and they provide a richer capability in the relationship they can model and the output they can produce. They decide which neurons will be activated—that is, what information is passed to further layers.

Without activation functions, ANNs lose a bulk of their representation learning power. There are several activation functions. The most widely used are as follows:

Linear (identity) function
: Represented by the equation of a straight line (i.e., $f(x) = mx + c$), where activation is proportional to the input. If we have many layers, and all the layers are linear in nature, then the final activation function of the last layer is the same as the linear function of the first layer. The range of a linear function is $-inf$ to $+inf$.

Sigmoid function
: Refers to a function that is projected as an S-shaped graph (as shown in Figure 3-4). It is represented by the mathematical equation $f(x) = 1 / (1 + e^{-x})$ and ranges from 0 to 1. A large positive input results in a large positive output; a large negative input results in a large negative output. It is also referred to as logistic activation function.

Tanh function
: Similar to sigmoid activation function with a mathematical equation $Tanh(x) = 2Sigmoid(2x) - 1$, where *Sigmoid* represents the `sigmoid` function discussed above. The output of this function ranges from –1 to 1, with an equal mass on both sides of the zero-axis, as shown in Figure 3-4.

ReLU function
: ReLU stands for the Rectified Linear Unit and is represented as $f(x) = max(x, 0)$. So, if the input is a positive number, the function returns the number itself, and if the input is a negative number, then the function returns zero. It is the most commonly used function because of its simplicity.

Figure 3-4 shows a summary of the activation functions discussed in this section.

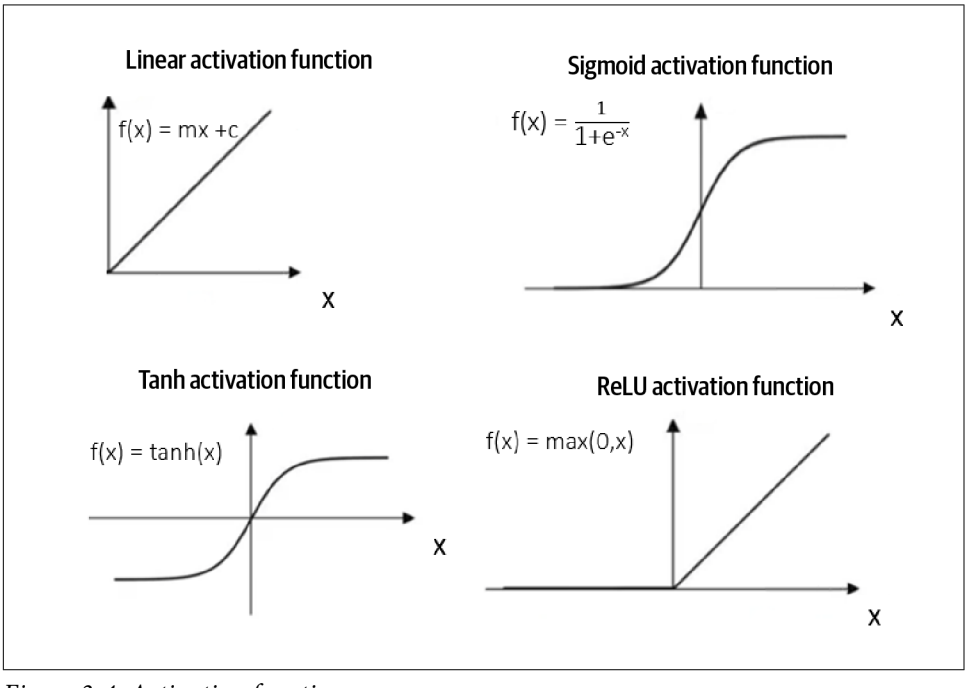


Figure 3-4. Activation functions

There is no hard-and-fast rule for activation function selection. The decision completely relies on the properties of the problem and the relationships being modeled. We can try different activation functions and select the one that helps provide faster convergence and a more efficient training process. The choice of activation function in the output layer is strongly constrained by the type of problem that is modeled.[4]

Cost functions

Cost functions (also known as loss functions) are a measure of the ANN performance, measuring how well the ANN fits empirical data. The two most common cost functions are:

Mean squared error (MSE)
: This is the cost function used primarily for regression problems, where output is a continuous value. MSE is measured as the average of the squared difference

4 Deriving a regression or classification output by changing the activation function of the output layer is described further in Chapter 4.

between predictions and actual observation. MSE is described further in Chapter 4.

Cross-entropy (or log loss)
: This cost function is used primarily for classification problems, where output is a probability value between zero and one. Cross-entropy loss increases as the predicted probability diverges from the actual label. A perfect model would have a cross-entropy of zero.

Optimizers

Optimizers update the weight parameters to minimize the loss function.[5] Cost function acts as a guide to the terrain, telling the optimizer if it is moving in the right direction to reach the global minimum. Some of the common optimizers are as follows:

Momentum
: The *momentum optimizer* looks at previous gradients in addition to the current step. It will take larger steps if the previous updates and the current update move the weights in the same direction (gaining momentum). It will take smaller steps if the direction of the gradient is opposite. A clever way to visualize this is to think of a ball rolling down a valley—it will gain momentum as it approaches the valley bottom.

AdaGrad (Adaptive Gradient Algorithm)
: *AdaGrad* adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features.

RMSProp
: *RMSProp* stands for Root Mean Square Propagation. In RMSProp, the learning rate gets adjusted automatically, and it chooses a different learning rate for each parameter.

Adam (Adaptive Moment Estimation)
: *Adam* combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization and is one of the most popular gradient descent optimization algorithms.

5 Refer to *https://oreil.ly/FSt-8* for more details on optimization.

Epoch

One round of updating the network for the entire training dataset is called an *epoch*. A network may be trained for tens, hundreds, or many thousands of epochs depending on the data size and computational constraints.

Batch size

The batch size is the number of training examples in one forward/backward pass. A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated. The higher the batch size, the more memory space is needed.

Creating an Artificial Neural Network Model in Python

In Chapter 2 we discussed the steps for end-to-end model development in Python. In this section, we dig deeper into the steps involved in building an ANN-based model in Python.

Our first step will be to look at Keras, the Python package specifically built for ANN and deep learning.

Installing Keras and Machine Learning Packages

There are several Python libraries that allow building ANN and deep learning models easily and quickly without getting into the details of underlying algorithms. Keras is one of the most user-friendly packages that enables an efficient numerical computation related to ANNs. Using Keras, complex deep learning models can be defined and implemented in a few lines of code. We will primarily be using Keras packages for implementing deep learning models in several of the book's case studies.

Keras is simply a wrapper around more complex numerical computation engines such as TensorFlow and Theano. In order to install Keras, TensorFlow or Theano needs to be installed first.

This section describes the steps to define and compile an ANN-based model in Keras, with a focus on the following steps.[6]

Importing the packages

Before you can start to build an ANN model, you need to import two modules from the Keras package: `Sequential` and `Dense`:

6 The steps and Python code related to implementing deep learning models using Keras, as demonstrated in this section, are used in several case studies in the subsequent chapters.

```
from Keras.models import Sequential
from Keras.layers import Dense
import numpy as np
```

Loading data

This example makes use of the `random` module of NumPy to quickly generate some data and labels to be used by ANN that we build in the next step. Specifically, an array with size *(1000,10)* is first constructed. Next, we create a labels array that consists of zeros and ones with a size *(1000,1)*:

```
data = np.random.random((1000,10))
Y = np.random.randint(2,size= (1000,1))
model = Sequential()
```

Model construction—defining the neural network architecture

A quick way to get started is to use the Keras Sequential model, which is a linear stack of layers. We create a Sequential model and add layers one at a time until the network topology is finalized. The first thing to get right is to ensure the input layer has the right number of inputs. We can specify this when creating the first layer. We then select a dense or fully connected layer to indicate that we are dealing with an input layer by using the argument `input_dim`.

We add a layer to the model with the `add()` function, and the number of nodes in each layer is specified. Finally, another dense layer is added as an output layer.

The architecture for the model shown in Figure 3-5 is as follows:

- The model expects rows of data with 10 variables (`input_dim_=10` argument).
- The first hidden layer has 32 nodes and uses the `relu` activation function.
- The second hidden layer has 32 nodes and uses the `relu` activation function.
- The output layer has one node and uses the `sigmoid` activation function.

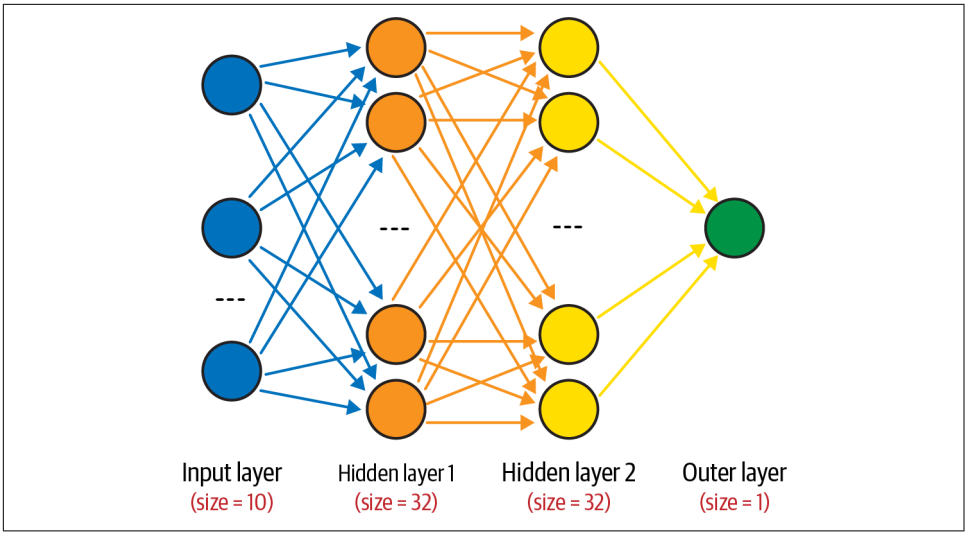


Figure 3-5. An ANN architecture

The Python code for the network in Figure 3-5 is shown below:

```
model = Sequential()
model.add(Dense(32, input_dim=10, activation= 'relu' ))
model.add(Dense(32, activation= 'relu' ))
model.add(Dense(1, activation= 'sigmoid'))
```

Compiling the model

With the model constructed, it can be compiled with the help of the `compile()` function. Compiling the model leverages the efficient numerical libraries in the Theano or TensorFlow packages. When compiling, it is important to specify the additional properties required when training the network. Training a network means finding the best set of weights to make predictions for the problem at hand. So we must specify the loss function used to evaluate a set of weights, the optimizer used to search through different weights for the network, and any optional metrics we would like to collect and report during training.

In the following example, we use `cross-entropy` loss function, which is defined in Keras as `binary_crossentropy`. We will also use the adam optimizer, which is the default option. Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.[7] The Python code follows:

```
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , \
  metrics=[ 'accuracy' ])
```

7 A detailed discussion of the evaluation metrics for classification models is presented in Chapter 4.

Fitting the model

With our model defined and compiled, it is time to execute it on data. We can train or fit our model on our loaded data by calling the `fit()` function on the model.

The training process will run for a fixed number of iterations (epochs) through the dataset, specified using the `nb_epoch` argument. We can also set the number of instances that are evaluated before a weight update in the network is performed. This is set using the `batch_size` argument. For this problem we will run a small number of epochs (10) and use a relatively small batch size of 32. Again, these can be chosen experimentally through trial and error. The Python code follows:

```
model.fit(data, Y, nb_epoch=10, batch_size=32)
```

Evaluating the model

We have trained our neural network on the entire dataset and can evaluate the performance of the network on the same dataset. This will give us an idea of how well we have modeled the dataset (e.g., training accuracy) but will not provide insight on how well the algorithm will perform on new data. For this, we separate the data into training and test datasets. The model is evaluated on the training dataset using the `evaluation()` function. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics configured, such as accuracy. The Python code follows:

```
scores = model.evaluate(X_test, Y_test)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Running an ANN Model Faster: GPU and Cloud Services

For training ANNs (especially deep neural networks with many layers), a large amount of computation power is required. Available CPUs, or Central Processing Units, are responsible for processing and executing instructions on a local machine. Since CPUs are limited in the number of cores and take up the job sequentially, they cannot do rapid matrix computations for the large number of matrices required for training deep learning models. Hence, the training of deep learning models can be extremely slow on the CPUs.

The following alternatives are useful for running ANNs that generally require a significant amount of time to run on a CPU:

- Running notebooks locally on a GPU.
- Running notebooks on Kaggle Kernels or Google Colaboratory.
- Using Amazon Web Services.

GPU

A GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. Running ANNs and deep learning models can be accelerated by the use of GPUs.

GPUs are particularly adept at processing complex matrix operations. The GPU cores are highly specialized, and they massively accelerate processes such as deep learning training by offloading the processing from CPUs to the cores in the GPU subsystem.

All the Python packages related to machine learning, including Tensorflow, Theano, and Keras, can be configured for the use of GPUs.

Cloud services such as Kaggle and Google Colab

If you have a GPU-enabled computer, you can run ANNs locally. If you do not, we recommend you use a service such as Kaggle Kernels, Google Colab, or AWS:

Kaggle
: A popular data science website owned by Google that hosts Jupyter service and is also referred to as Kaggle Kernels. Kaggle Kernels are free to use and come with the most frequently used packages preinstalled. You can connect a kernel to any dataset hosted on Kaggle, or alternatively, you can just upload a new dataset on the fly.

Google Colaboratory
: A free Jupyter Notebook environment provided by Google where you can use free GPUs. The features of Google Colaboratory are the same as Kaggle.

Amazon Web Services (AWS)
: AWS Deep Learning provides an infrastructure to accelerate deep learning in the cloud, at any scale. You can quickly launch AWS server instances preinstalled with popular deep learning frameworks and interfaces to train sophisticated, custom AI models, experiment with new algorithms, or learn new skills and techniques. These web servers can run longer than Kaggle Kernels. So for big projects, it might be worth using an AWS instead of a kernel.

Chapter Summary

ANNs comprise a family of algorithms used across all types of machine learning. These models are inspired by the biological neural networks containing neurons and layers of neurons that constitute animal brains. ANNs with many layers are referred to as deep neural networks. Several steps, including forward propagation and back-propagation, are required for training these ANNs. Python packages such as Keras make the training of these ANNs possible in a few lines of code. The training of these deep neural networks require more computational power, and CPUs alone might not be enough. Alternatives include using a GPU or cloud service such as Kaggle Kernels, Google Colaboratory, or Amazon Web Services for training deep neural networks.

Next Steps

As a next step, we will be going into the details of the machine learning concepts for supervised learning, followed by case studies using the concepts covered in this chapter.