CHAPTER 2

Keras in Action

In this chapter, we will explore the Keras framework and get started with hands-on exercises to learn the basics of Keras along with a bit of Python and the necessary DL topics. A word of caution, given that this a fast-track guide: we will not have the scope to talk in detail about exhaustive topics in DL. Instead, we will start with a simple topic, explore the basic idea behind it, and add references where you can dive deeper for a more foundational understanding of the topic.

Setting Up the Environment

As discussed earlier, we will be developing DL models with the Keras stack using TensorFlow as a back end in Python. Hence, to get started we need to set up our playground environment by installing Python, a few important Python packages, TensorFlow, and finally Keras.

Let's get started.

Selecting the Python Version

Python is currently available in two major versions: 2.7.x and 3.x. Although Python 3.x is the most recent version and the future of Python, there have been a series of conflicts due to backward incapability in the developer community with regard to the transition from 2.7 to 3.x. Unfortunately, many developers are still connected with the Python 2.7.x version.

J. Moolayil, *Learn Keras for Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-4240-7_2

However, for our use case, I highly recommend getting started with Python 3.x, given that it is the future. Some may be reluctant to start with Python 3, assuming there will be issues with many packages in the 3.x version, but for almost all practical use cases, we have all major DL, ML, and other useful packages already updated for 3.x.

Installing Python for Windows, Linux, or macOS

There are many distributions of Python available in the market. You could either download and install Python from the official python.org website or choose any popular distribution. For ML and DL, the most recommended distribution of Python is the Anaconda distribution from Continuum Analytics. Anaconda is a free and open source distribution of Python, especially for ML and DL large-scale processing. It simplifies the entire package management and deployment process and comes with a very easy to use virtual environment manager and a couple of additional tools for coding like Jupyter Notebooks and the Spyder IDE.

To get started with Anaconda, you can go to `www.anaconda.com/download/` and select an appropriate version based on the OS (Mac/Windows/Linux) and architecture (32 bit/64 bit) of your choice. At the time of writing this book, the most recent version of Python 3 is 3.6. By the time you read this book, there might be a newer version available. You should comfortably download and install the most updated version of Anaconda Python.

Once you have downloaded the installer, please install the application.

For Windows users, this will be a simple executable file installation. Double-click the .exe file downloaded from Anaconda's website and follow the visual onscreen guidelines to complete the installation process.

Linux users can use the following command after navigating to the downloaded folder:

```
bash Anaconda-latest-Linux-x86_64.sh
```

Mac users can install the software by double-clicking the downloaded .pkg file and then following the onscreen instructions.

The Anaconda distribution of Python eases out the process for DL and ML by installing all major Python packages required for DL.

Installing Keras and TensorFlow Back End

Now that Python is set up, we need to install TensorFlow and Keras. Installing packages in Python can be done easily using the `pip`, a package manager for Python. You can install any Python package with the command `pip install package-name` in the terminal or command prompt.

So, let's install our required packages (i.e., TensorFlow and Keras).

```
pip install keras
```

followed by

```
pip install tensorflow
```

In case you face any issues in setting Anaconda Python with TensorFlow and Keras, or you want to experiment only within a Python virtual environment, you can explore a more detailed installation guide here:

https://medium.com/@margaretmz/anaconda-jupyter-notebook-tensorflow-and-keras-b91f381405f8

Also, you might want to install TensorFlow with GPU support if your system has any NVIDIA CUDA–compatible GPUs. Here is a link to a step-by-step guide to install TensorFlow with GPU support on Windows, Mac and Linux:

www.tensorflow.org/install/

To check if your GPU is CUDA compatible, please explore the list available on NVIDIA's official website:

https://developer.nvidia.com/cuda-gpus

To write codes and develop models, you can choose the IDE provided by Anaconda (i.e., Spyder), the native terminal or command prompt, or a web-based notebook IDE called Jupyter Notebooks. For all data science–related experiments, I would highly recommend using Jupyter Notebooks for the convenience it provides in exploratory analysis and reproducibility. We will be using Jupyter Notebooks for all experiments in our book.

Jupyter Notebooks comes preinstalled with Anaconda Python; in case you are using a virtual environment, you might have to install it using the package manager or just the command

```
conda install jupyter
```

To start Jupyter Notebooks, you can use the Anaconda Navigator or just enter the command

```
jupyter notebook
```

inside your command prompt or terminal; then, Jupyter should start in your default browser on localhost. The following screenshot shows when Jupyter is running in the browser.

Click the 'New' button at the extreme right and select Python from the drop-down menu. If you have installed one or more virtual environments, all of them will show up in the drop-down; please select the Python environment of your choice.

Once selected, your Jupyter notebook should open and should be ready to get started. The following screenshot showcases a Jupyter notebook up and running in the browser.

The green highlighted cell is where you write your code, and Ctrl + Enter will execute the selected cell. You can add more cells with the '+' icon in the control bar or explore additional options from the Menu bar. If this is your first time with Jupyter, I recommend the available options in the navigation menu.

Now that we have all the required tools set up and running, let's start with simple DL building blocks with Keras.

Getting Started with DL in Keras

Let's start by studying the DNN and its logical components, understanding what each component is used for and how these building blocks are mapped in the Keras framework.

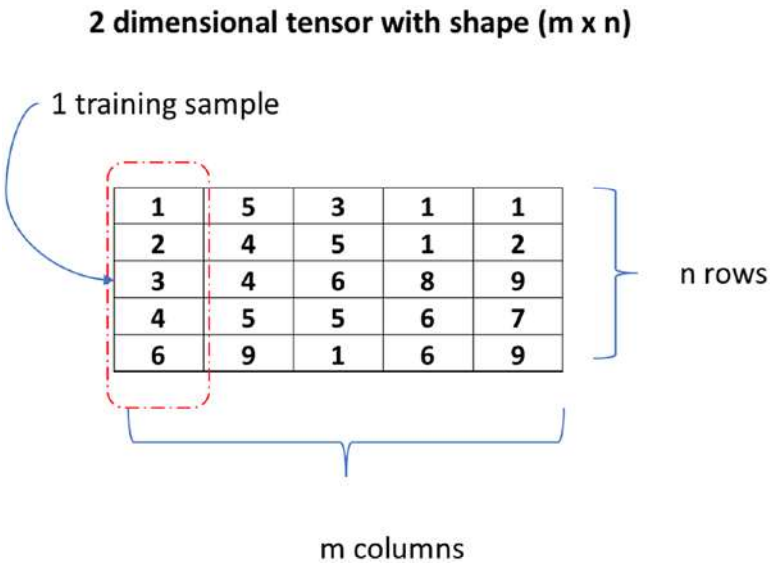
If you recall the topic "Decomposing a DL Model" from Chapter 1, we had defined the logical components in a DNN as input data, neurons, activation functions, layer (i.e., group of neurons), connections between neurons or edges, a learning procedure (i.e., the backpropagation algorithm), and the output layer.

Let's look at at these logical components one by one.

Input Data

Input data for a DL algorithm can be of a variety of types. Essentially, the model understands data as "tensors". Tensors are nothing but a generic form for vectors, or in computer engineering terms, a simple n-dimensional matrix. Data of any form is finally represented as a

homogeneous numeric matrix. So, if the data is tabular, it will be a two-dimensional tensor where each column represents one training sample and the entire table/matrix will be m samples. To understand this better, have a look at the following visual illustration.



You could also reverse the representation of training samples (i.e., each row could be one training sample), so in the context of the student passing/failing in the test example, one row would indicate all the attributes of one student (his marks, age, etc.). And for n rows, we would have a dataset with n training samples. But in DL experiments, it is common notation to use one training sample in a column. Thus, m columns would denote m samples.

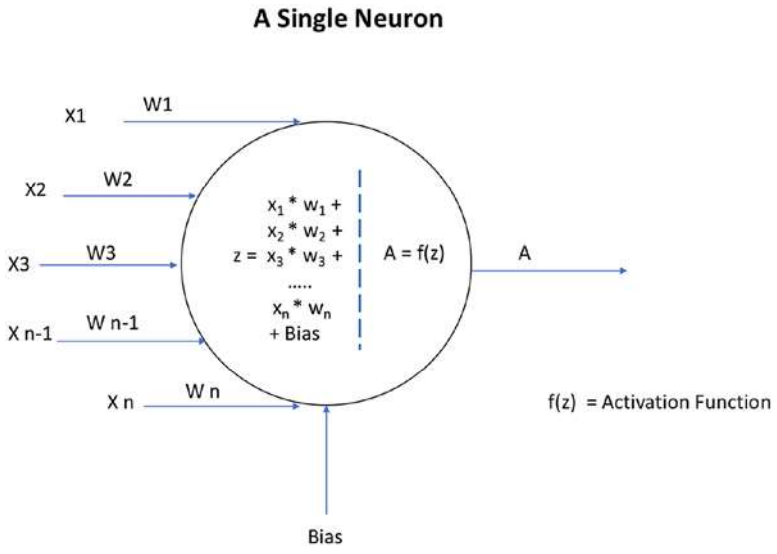
Additionally, DL models can interpret only numeric data. If the dataset has any categorical data like "gender" with values of "male" and "female," we will need to convert them to one-hot encoded variables (i.e., simply representing the columns with a value of 0 or 1, where 0 would represent "male" and 1 would represent "female" or vice versa).

Image data also needs to be transformed into an n-dimensional tensor. We will not cover DL models for image data in this book but I do want to keep you aware of its representation as input data. An image is stored in data as a three-dimensional tensor where two dimensions define the pixel values on a 2D plane and a third dimension defines the values for RGB color channels. So essentially, one image becomes a three-dimensional tensor and n images will be a four-dimensional tensor, where the fourth dimension will stack a 3D tensor image as a training sample. Therefore, if we have 100 images with a 512 × 512-pixel resolution, they will be represented as a 4D tensor with shape 512 × 512 × 3 × 100.

Lastly, it is a good practice to normalize, standardize, or scale the input values before training. Normalizing the values will bring all values in the input tensor into a 0–1 range, and standardization will bring the values into a range where the mean is 0 and the standard deviation is 1. This helps to reduce computation, as the learning improves by a great margin and so does performance, as the activation functions (covered in the following) behave more appropriately.

Neuron

At the core of the DNN, we have neurons where computation for an output is executed. A neuron receives one or more inputs from the neurons in the previous layer. If the neurons are in the first hidden layer, they will receive the data from the input data stream. In the biological neuron, an electric signal is given as an output when it receives an input with a higher influence. To map that functionality in the mathematical neuron, we need to have a function that operates on the sum of input multiplied by the corresponding weights (denoted as f(z) in the following visual) and responds with an appropriate value based on the input. If a higher-influence input is received, the output should be higher, and vice versa. It is in a way analogous to the activation signal (i.e., higher influence -> then activate, otherwise deactivate). The function that works on the computed input data is called the activation function.



Activation Function

An activation function is the function that takes the combined input z as shown in the preceding illustration, applies a function on it, and passes the output value, thus trying to mimic the activate/deactivate function. The activation function, therefore, determines the state of a neuron by computing the activation function on the combined input.

A quick thought crossing your mind might be as follows: why do we really need an activation function to compute the combined output z, when we could just pass the value of z as the final output? There are several problems here. Firstly, the range of the output value would be -Infinity to + Infinity, where we won't have a clear way of defining a threshold where activation should happen. Secondly, the network will in a way be

rendered useless, as it won't really learn. This is where a bit of calculus and derivatives come into the picture. To simplify the story, we can say that if your activation function is a linear function (basically no activation), then the derivative of that function becomes 0; this becomes a big issue because training with the backpropagation algorithm helps give feedback to the network about wrong classifications and thereby helps a neuron to adjust its weights by using a derivative of the function. If that becomes 0, the network loses out on this learning ability. To put it another way, we can say there is really no point of having the DNN, as the output of having just one layer would be similar to having n layers. To keep things simple, we would always need a nonlinear activation function (at least in all hidden layers) to get the network to learn properly.

There are a variety of choices available to use as an activation function. The most common ones are the sigmoid function and the ReLU (rectified linear unit).

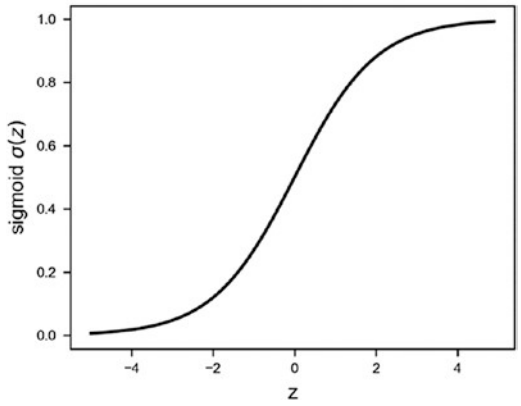
Sigmoid Activation Function

A sigmoid function is defined as $\frac{1}{\left(1+e^{-z}\right)}$, which renders the output between 0 and 1 as shown in the following illustration. The nonlinear output (s shaped as shown) improves the learning process very well, as it closely resembles the following principle—*lower influence: low output* and *higher influence: higher output*—and also confines the output within the 0-to-1 range.

In Keras, the sigmoid activation function is available as keras.activations.sigmoid(x).

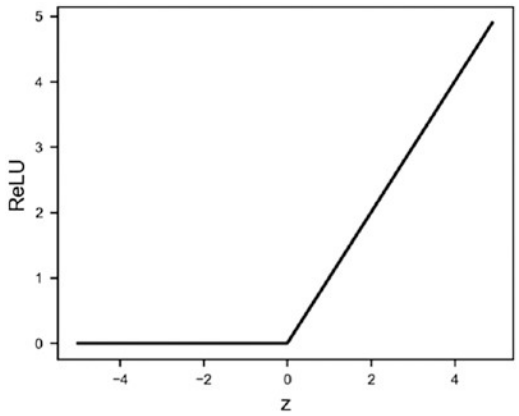
We can import this into Python simply with the `import` command:

```
import keras.activations.sigmoid
```



ReLU Activation Function

Similarly, the ReLU uses the function **f(z) = max(0,z),** which means that if the output is positive it would output the same value, otherwise it would output 0. The function's output range is shown in the following visual.



Keras provides ReLU as

```
keras.activations.relu(x, alpha=0.0, max_value=None)
```

The function may look linear, but it isn't. ReLU is a valid nonlinear function and in fact works really well as an activation function. It not only improves the performance but significantly helps the number of

computations to be reduced during the training phase. This is a direct result of the 0 value in the output when z is negative, thereby deactivating the neuron.

But because of the horizontal line with 0 as the output, we can face serious issues sometimes. For instance, in the previous section we discussed a horizontal line, which is a constant with a derivative of 0 and therefore may become a bottleneck during training, as the weights will not easily get updated. To circumvent the problem, there was a new activation function proposed: Leaky ReLU, where the negative value outputs a slightly slanting line instead of a horizontal line, which helps in updating the weights through backpropagation effectively.

Leaky ReLU is defined as

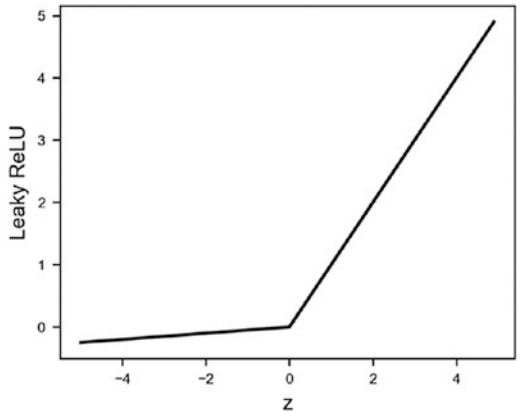
$f(z) = z$; when $z > 0$

$f(z) = \propto z$; when $z < 0$ and where $\propto$ is a parameter that is defined as a small constant, say 0.005

Keras provides Leaky ReLU as follows:

```
keras.layers.LeakyReLU(X, alpha=0.0, max_value=None).
```

We can directly use the activation function by setting the value of alpha with a small constant.



There are many more activation functions that can be used in a DNN and are available in Keras. A few other popular ones are tanh (hyperbolic tan activation), swish activation, elu (exponential linear unit), selu (scaled elu), and so on.

Model

The overall structure of a DNN is developed using the model object in Keras. This provides a simple way to create a stack of layers by adding new layers one after the other.

The easiest way to define a model is by using the sequential model, which allows easy creation of a linear stack of layers.

The following example showcases the creation of a simple sequential model with one layer followed by an activation. The layer would have 10 neurons and would receive an input with 15 neurons and be activated with the ReLU activation function.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(10, input_dim=15))
model.add(Activation('relu'))
```

Layers

A layer in the DNN is defined as a group of neurons or a logically separated group in a hierarchical network structure. As DL became more and more popular, there were several experiments conducted with network architectures to improve performance for a variety of use cases. The use cases centered around regular supervised algorithms like classification and regression, computer vision experiments, extending DL for natural language processing and understanding, speech recognition, and

combinations of different domains. To simplify the model development process, Keras provides us with several types of layers and various means to connect them. Discussing all of them would be beyond the scope of the book. However, we will take a close look at a few layers and also glance through some important layers for other advanced use cases, which you can explore later.

Core Layers

There are a few important layers that we will be using in most use cases.

Dense Layer

A dense layer is a regular DNN layer that connects every neuron in the defined layer to every neuron in the previous layer. For instance, if Layer 1 has 5 neurons and Layer 2 (dense layer) has 3 neurons, the total number of connections between Layer 1 and Layer 2 would be 15 (5×3). Since it accommodates every possible connection between the layers, it is called a "dense" layer.

Keras offers the dense layer with the following default parameters.

```
keras.layers.Dense(units, activation=None, use_bias=True,
                   kernel_initializer='glorot_uniform',
                   bias_initializer='zeros',
                   kernel_regularizer=None,
                   bias_regularizer=None,
                   activity_regularizer=None,
                   kernel_constraint=None,
                   bias_constraint=None)
```

It offers a lot of customization for any given layer. We can specify the number of units (i.e., neurons for the layer), the activation type, the type initialization for kernel and bias, and other constraints. Most often, we just use parameters like units and activation. The rest can be left to the defaults

for simplicity. These additional parameters become important when we are working in specialized use cases where the importance of using specific types of constraints and initializers for a given layer is paramount.

We also need to define the input shape for the Keras layer. The input shape needs to be defined for only the first layer. Subsequent layers just need the number of neurons defined. We can use the `input_dim` attribute to define how many dimensions the input has. For instance, if we have a table with 10 features and 1000 samples, we need to provide the `input_dim` as 10 for the layer to understand the shape of input data.

Example: A network with one hidden layer and the output layer for simple binary classification.

Layer 1 has 5 neurons and expects an input with 10 features; therefore, `input_dim =10`. The final layer is the output, which has one neuron.

```
model = Sequential()
model.add(Dense(5,input_dim=10,activation = "sigmoid"))
model.add(Dense(1,activation = "sigmoid"))
```

Dropout Layer

The dropout layer in DL helps reduce overfitting by introducing regularization and generalization capabilities into the model. In the literal sense, the dropout layer drops out a few neurons or sets them to 0 and reduces computation in the training process. The process of arbitrarily dropping neurons works quite well in reducing overfitting. We will take up this topic in more depth and understand the rationale behind overfitting, model generalization in Chapter 5.

Keras offers a dropout layer with the following default parameters:

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

We add the dropout layer after a regular layer in the DL model architecture. The following codes show a sample:

```
model = Sequential()
model.add(Dense(5,input_dim=10,activation = "sigmoid"))
model.add(Dropout(rate = 0.1,seed=100))
model.add(Dense(1,activation = "sigmoid"))
```

Other Important Layers

Considering the diversity of use cases, Keras has inbuilt defined layers for most. In computer vision use cases, the input is usually an image. There are special layers to extract features from images; they are called convolutional layers. Similarly, for natural language processing and similar use cases, there is an advanced DNN called recurrent neural network (RNN). Keras has provided several different types of recurrent layers for its development.

The list is quite long, and we won't cover the other advanced layers now. However, in order to keep you updated, here are some of the other important layers in Keras that will be handy for you for advanced use cases in the future:

- Embedding layers - https://keras.io/layers/embeddings/
- Convolutional layers - https://keras.io/layers/convolutional/
- Pooling layers - https://keras.io/layers/pooling/
- Merge layers - https://keras.io/layers/merge/
- Recurrent layers - https://keras.io/layers/recurrent/
- Normalization layers and many more - https://keras.io/layers/normalization/

You can also write your own layers in Keras for a different type of use case. More details can be explored here: https://keras.io/layers/writing-your-own-keras-layers/

The Loss Function

The loss function is the metric that helps a network understand whether it is learning in the right direction. To frame the loss function in simple words, consider it as the test score you achieve in an examination. Say you appeared for several tests on the same subject: what metric would you use to understand your performance on each test? Obviously, the test score. Assume you scored 56, 60, 78, 90, and 96 out of 100 in five consecutive language tests. You would clearly see that the improving test scores are an indication of how well you are performing. Had the test scores been decreasing, then the verdict would be that your performance is decreasing and you would need to change your studying methods or materials to improve.

Similarly, how does a network understand whether it is improving its learning process in each iteration? It uses the loss function, which is analogous to the test score. The loss function essentially measures the loss from the target. Say you are developing a model to predict whether a student will pass or fail and the chance of passing or failing is defined by the probability. So, 1 would indicate that he will pass with 100% certainty and 0 would indicate that he will definitely fail.

The model learns from the data and predicts a score of 0.87 for the student to pass. So, the actual loss here would be 1.00 – 0.87 = 0.13. If it repeats the exercise with some parameter updates in order to improve and now achieves a loss of 0.40, it would understand that the changes it has made are not helping the network to appropriately learn. Alternatively, a new loss of 0.05 would indicate that the updates or changes from the learning are in the right direction.

Based on the type of data outcome, we have several standard loss functions defined in ML and DL. For regression use cases (i.e., where the end prediction would be a continuous number like the marks scored by a

student, the number of product units sold by a shop, the number of calls received from customers in a contact center, etc.), here are some popular loss functions available:

- Mean Squared Error - Average squared difference between the actual and predicted value. The squared difference makes it easy to penalize the model more for a higher difference. So, a difference of 3 would result in a loss of 9, but difference of 9 would return a loss of 81.
  - The mathematical equivalent would be

    $$\sum_{n=1}^{k} \frac{(Actual - Predicted)^2}{k}$$

  - Keras equivalent

    ```
    keras.losses.mean_squared_error(y_actual,
    y_pred)
    ```

- Mean Absolute Error – The average absolute error between actual and predicted.
  - The mathematical equivalent would be

    $$\sum_{n=1}^{k} |Actual - Predicted|$$

  - Keras equivalent

    ```
    keras.losses.mean_absolute_error
    (y_actual, y_pred)
    ```

- Similarly, few other variants are
  - MAPE – Mean absolute percentage error

    ```
    keras.losses.mean_absolute_percentage_error
    ```

  - MSLE – Mean square logarithmic error

    ```
    keras.losses.mean_squared_logarithmic_error
    ```

For categorical outcomes, your prediction would be for a class, like whether a student will pass (1) or fail (0), whether the customer will make a purchase or not, whether the customer will default on payment or not, and so on. Some use cases may have multiple classes as an outcome, like classifying types of disease (Type A, B, or C), classifying images as cats, dogs, cars, horses, landscapes, and so on.

In such cases, the losses defined in the preceding cannot be used due to obvious reasons. We would need to quantify the outcome of the class as probability and define losses based on the probability estimates as predictions.

A few popular choices for losses for categorical outcomes in Keras are as follows:

- **Binary cross-entropy:** Defines the loss when the categorical outcomes is a binary variable, that is, with two possible outcomes: (Pass/Fail) or (Yes/No)
  - The mathematical form would be

    $$\text{Loss} = -[\, y * \log(p) + (1-y) * \log(1-p) \,]$$

  - Keras equivalent

    ```
    keras.losses.binary_crossentropy(y_
    actual, y_predicted)
    ```

- **Categorical cross-entropy:** Defines the loss when the categorical outcomes is a nonbinary, that is, >2 possible outcomes: (Yes/No/Maybe) or (Type 1/ Type 2/… Type n)
  - The mathematical form would be

    $$\text{Loss} = -\sum_{i}^{n} y_i' log_2\, y_i$$

  - Keras equivalent

    ```
    keras.losses.categorical_crossentropy
    (y_actual, y_predicted)
    ```

Optimizers

The most important part of the model training is the optimizer. Up to this point, we have addressed the process of giving feedback to the model through an algorithm called backpropagation; this is actually an optimization algorithm.

To add more context, imagine the model structure that you have defined to classify whether a student will pass or fail. The structure created by defining the sequence of layers with the number of neurons, the activation functions, and the input and output shape is initialized with random weights in the beginning. The weights that determined the influence of a neuron on the next neuron or the final output are updated during the learning process by the network.

In a nutshell, a network with randomized weights and a defined structure is the starting point for a model. The model can make a prediction at this point, but it would almost always be of no value. The network takes one training sample and uses its values as inputs to the neurons in the first layer, which then produces an output with the defined activation function. The output now becomes an input for the next layer, and so on. The output of the final layer would be the prediction for the training sample. This is where the loss function comes into the picture. The loss function helps the network understand how well or poorly the current set of weights has performed on the training sample. The next step for the model is to reduce the loss. How does it know what steps or updates it should perform on the weights to reduce the loss? The optimizer function helps it understand this step. The optimizer function is a mathematical algorithm that uses derivatives, partial derivatives, and the chain rule in calculus to understand how much change the network will see in the loss function by making a small change in the weight of the neurons. The change in the loss function, which would be an increase or decrease, helps

in determining the direction of the change required in the weight of the connection. The computation of one training sample from the input layer to the output layer is called a pass. Usually, training would be done in batches due to memory constraints in the system. A batch is a collection of training samples from the entire input. The network updates its weights after processing all samples in a batch. This is called an iteration (i.e., a successful pass of all samples in a batch followed by a weight update in the network). The computing of all training samples provided in the input data with batch-by-batch weight updates is called an epoch. In each iteration, the network leverages the optimizer function to make a small change to its weight parameters (which were randomly initialized at the beginning) to improve the end prediction by reducing the loss function. Step by step, with several iterations and then several epochs, the network updates its weights and learns to make a correct prediction for the given training samples.

The mathematical explanation for the functioning of the optimizer function was abstracted in a simple way for you to understand and appreciate the background operations that happen in the DNN during the training process. The in-depth math equations and the reasoning for the optimization process are beyond the scope of this book. In case you are supercurious about learning math and the actual process of the optimization algorithm, I would recommend reading a chapter from the book *Pro Deep Learning with TensorFlow* by Santanu Pattanayak (Apress, 2017). The book does an amazing job of explaining the math behind DL with a very intuitive approach. I highly recommend this book to all PhD students exploring DL.

Given that you have a fair understanding of the overall optimization process, I would like to take a minute to discuss various optimization algorithms available in Keras.

Stochastic Gradient Descent (SGD)

SGD performs an iteration with each training sample (i.e., after the pass of every training sample, it calculates the loss and updates the weight). Since the weights are updated too frequently, the overall loss curve would be very noisy. However, the optimization is relatively fast compared to others.

The formula for weight updates can be expressed in a simple way as follows:

> Weights = Weights – learning rate * Loss
>
> Where learning rate is a parameter we define in the network architecture.
>
> Say, for learning rate =0.01

Keras provides SGD with

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0,
nesterov=False)
```

For updates with every training sample, we would need to use batch_size=1 in the model training function.

To reduce high fluctuations in the SGD optimizations, a better approach would be to reduce the number of iterations by providing a minibatch, which would then enable averaging the loss for all samples in a batch and updating the weights at the end of the batch. This approach has been more successful and results in a smoother training process. Batch size is usually set in powers of 2 (i.e., 32, 64, 128, etc.).

Adam

Adam, which stands for Adaptive Moment Estimation, is by far the most popular and widely used optimizer in DL. In most cases, you can blindly choose the Adam optimizer and forget about the optimization alternatives. This optimization technique computes an adaptive learning rate for each

parameter. It defines momentum and variance of the gradient of the loss and leverages a combined effect to update the weight parameters. The momentum and variance together help smooth the learning curve and effectively improve the learning process.

The math representation can be simplified in the following way:

> Weights = Weights – (Momentum and Variance combined)

Keras provides the Adam optimizer as

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
```

The parameters beta_1 and beta_2 are used in computing the momentum and variance, respectively. The default values work quite effectively and doesn't need to be changed for most use cases.

Other Important Optimizers

There are many other popular optimizers that can also be used for different DL models. Discussing all of them would be beyond the scope of this book. In the interest of keeping you well informed about the available options, I would like to list a few of the other popular optimization alternatives used and available within Keras:

- Adagrad
- Adadelta
- RMSProp
- Adamax
- Nadam

Each of the optimization techniques has its own pros and cons. A major problem which we often face in DL is the vanishing gradient and saddle point problem. You can explore these problems in more detail while choosing the best optimizer for your problem. But for most use cases, Adam always works fine.

Metrics

Similar to the loss function, we also define metrics for the model in Keras. In a simple way, metrics can be understood as the function used to judge the performance of the model on a different unseen dataset, also called the validation dataset. The only difference between metrics and the loss function is that the results from metrics are not used in training the model with respect to optimization. They are only used to validate the test results while reporting.

A few available options for metrics in Keras are as follows:

- Binary Accuracy - keras.metrics.binary_accuracy
- Categorical Accuracy - keras.metrics.caetogrical_accuracy
- Sparse Categorical Accuracy - keras.metrics.sparse_categorical_accuracy

You can also define custom functions for your model metrics. Keras provides you with the ability to easily configure a model with user-defined metrics.

Model Configuration

Now that we understand the most fundamental building blocks of a DNN in Keras, we can take a look at the final model configuration step, which orchestrates all the preceding components together.

Once you have designed your network, Keras provides you with an easy one-step model configuration process with the 'compile' command. To compile a model, we need to provide three parameters: an optimization function, a loss function, and a metric for the model to measure performance on the validation dataset.

The following example builds a DNN with two hidden layers, with 32 and 16 neurons, respectively, with a ReLU activation function. The final output is for a binary categorical numeric output using a sigmoid activation. We compile the model with the Adam optimizer and define binary cross-entropy as the loss function and "accuracy" as the metric for validation.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(32, input_dim=10,activation = "relu"))
model.add(Dense(16,activation = "relu"))
model.add(Dense(1,activation = "sigmoid"))

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

Model Training

Once we configure a model, we have all the required pieces for the model ready. We can now go ahead and train the model with the data. While training, it is always a good practice to provide a validation dataset for us to evaluate whether the model is performing as desired after each epoch. The model leverages the training data to train itself and learn the patterns, and at the end of each epoch, it will use the unseen validation data to make predictions and compute metrics. The performance on the validation dataset is a good cue for the overall performance.

For validation data, it is a common practice to divide your available data into three parts with a 60:20:20 ratio. We use 60% for training, 20% for validation, and the last 20% for testing. This ratio is not a mandate. You have the flexibility to change the ratio as per your choice. In general, when you have really large training datasets, say n>1MN samples, it is fine to take 95% for training, 2% for validation, and 3% for testing. Again, the ratio is a choice you make based on your judgment and available data.

Keras provides a fit function for the model object to train with the provided training data.

Here is a sample model invoking its fit method. At this point, it is assumed that you have the model architecture defined and configured (compiled) as discussed in the preceding.

```
model.fit(x_train, y_train, batch_size=64, epochs=3,
validation_data=(x_val, y_val))
```

We have a model being trained on a training dataset named x_train with the actual labels in y_train. We choose a batch size of 64. Therefore, if there were 500 training samples, the model would intake and process 64 samples at a time in a batch before it updates the model weights. The last batch may have <64 training sample if unavailable. We have set the number of epochs to three; therefore, the whole process of training 500 sample in batches of 64 will be repeated thrice. Also, we have provided the validation dataset as x_val and y_val. At the end of each epoch, the model would use the validation data to make predictions and compute the performance metrics as defined in the metrics parameter of the model configuration.

Now that we have all the pieces required for the model to be designed, configured, and trained, let's put all pieces of the puzzle together and see it in action.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
```

```
# Generate dummy training dataset
np.random.seed(2018)
x_train = np.random.random((6000,10))
y_train = np.random.randint(2, size=(6000, 1))

# Generate dummy validation dataset
x_val = np.random.random((2000,10))
y_val = np.random.randint(2, size=(2000, 1))

# Generate dummy test dataset
x_test = np.random.random((2000,10))
y_test = np.random.randint(2, size=(2000, 1))

#Define the model architecture
model = Sequential()
model.add(Dense(64, input_dim=10,activation = "relu")) #Layer 1
model.add(Dense(32,activation = "relu"))               #Layer 2
model.add(Dense(16,activation = "relu"))               #Layer 3
model.add(Dense(8,activation = "relu"))                #Layer 4
model.add(Dense(4,activation = "relu"))                #Layer 5
model.add(Dense(1,activation = "sigmoid"))             #Output
                                                       Layer

#Configure the model
model.compile(optimizer='Adam',loss='binary_crossentropy',metri
cs=['accuracy'])

#Train the model
model.fit(x_train, y_train, batch_size=64, epochs=3,
validation_data=(x_val,y_val))
```

The output while training the model is showcased in the following:

```
Train on 6000 samples, validate on 2000 samples
Epoch 1/3
6000/6000 [==============================] - 2s 363us/step - loss: 0.6934 - acc: 0.4972 - val_loss: 0.6932 - val_acc:
0.4945
Epoch 2/3
6000/6000 [==============================] - 0s 58us/step - loss: 0.6933 - acc: 0.4993 - val_loss: 0.6934 - val_acc:
0.4945
Epoch 3/3
6000/6000 [==============================] - 0s 55us/step - loss: 0.6931 - acc: 0.5045 - val_loss: 0.6933 - val_acc:
0.5110
```

We can see that after every epoch, the model prints the mean training loss and accuracy as well as the validation loss and accuracy. We can use these intermediate results to make a judgment on the model performance. In most large DL use cases, we would have several epochs for training. It is a good practice to keep a track of the model performance with the metrics we have configured at intervals to see the results after a few epochs. If the results don't seem in your favor, it might be a good idea to stop the training and revisit the model architecture and configuration.

Model Evaluation

In all of the preceding examples, we have looked into a specific portion of the model development step or we have concluded with model training. We haven't discussed model performance so far. Understanding how effectively your model is performing on an unseen test dataset is of paramount importance.

Keras provides the model object equipped with inbuilt model evaluation and another function to predict the outcome from a test dataset. Let's have a look at both of these using the trained model and dummy test data generated in the preceding example.

The method provided by Keras for the sequential model is as shown in the following:

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_
weight=None, steps=None)
```

We provide the test data and the test labels in the parameters x and y. In cases where the test data is also huge and expected to consume a significant amount of memory, you can use the batch size to tell the Keras model to make predictions batch-wise and then consolidate all results.

```
print(model.evaluate(x_test,y_test))
[0.6925005965232849, 0.521]
```

In the evaluate method, the model returns the loss value and all metrics defined in the model configuration. These metric labels are available in the model property metrics_names.

```
print(model.metrics_names)
['loss', 'acc']
```

We can therefore see that the model has an overall accuracy of 52% on the test dataset. This is definitely not a good model result, but it was expected given that we used just a dummy dataset.

Alternatively, you could use the predict method of the model and leverage the actual predictions that would be probabilities (for this use case, since binary classification):

```
#Make predictions on the test dataset and print the first 10
predictions
pred = model.predict(x_test)
pred[:10]
```

Output

```
array([[0.4989694 ],
       [0.5111768 ],
       [0.4981183 ],
       [0.50972915],
       [0.5059872 ],
       [0.50466985],
       [0.5042962 ],
       [0.5179587 ],
       [0.5002746 ],
       [0.5066786 ]], dtype=float32)
```

This output can be used to make even more refined final predictions. A simple example is that the model would use 0.5 as the threshold for the predictions. Therefore, any predicted value above 0.5 is classified as 1 (say, Pass), and others as 0 (Fail).

Depending on your use case, you might want to slightly tweak your prediction for more aggressive correct prediction for 1 (Pass), so you might choose a threshold at 0.6 instead of 0.5, or vice versa.

Putting All the Building Blocks Together

I hope you can now make sense of the first DNN model we saw in the last section of Chapter 1. Before understanding all the basic building blocks, it would have been overwhelming to grasp the reasoning for the code used in the model development.

Now that we have all the basic necessary ingredients ready, let's look at more tangible use case before we conclude this chapter. To do so, let's take a better dataset and see what things look like. Keras also provides a few datasets to play with. These are real datasets and are usually used by most beginners during their initial experiments with ML and DL.

For our experiment, let's select a popular Keras dataset for developing a model. We can start with the Boston House Prices dataset. It is taken from the StatLib library, which is maintained at Carnegie Mellon University. The data is present in an Amazon S2 bucket, which we can download by using simple Keras commands provided exclusively for the datasets.

```
#Download the data using Keras; this will need an active
internet connection
from keras.datasets import boston_housing
(x_train, y_train), (x_test, y_test) = boston_housing.load_
data()
```

The dataset is directly downloaded into the Python environment and is ready to use. Let's have a look at what the data looks like. We will use basic Python commands to look at the type of data, its length and breadth, and a preview of the content.

```
#Explore the data structure using basic python commands
print("Type of the Dataset:",type(y_train))
print("Shape of training data :",x_train.shape)
print("Shape of training labels :",y_train.shape)
print("Shape of testing data :",type(x_test))
print("Shape of testing labels :",y_test.shape)
```

Output

```
Type of the Dataset: <class 'numpy.ndarray'>
Shape of training data : (404, 13)
Shape of training labels : (404,)
Shape of testing data : <class 'numpy.ndarray'>
Shape of testing labels : (102,)
```

We can see that the training and test datasets are Python numpy arrays. Numpy is a Python library to handle large multidimensional arrays. We have 404 rows of data with 13 features in the training dataset and 102 rows with the same number of features in the test dataset. Overall, it's approximately an 80:20 ratio between train and test. We can further divide the 402 rows of training data into 300 for training and 102 for validation.

Alright, the data structure and its shape look great. Let's have a quick look at the contents of the dataset. The preceding code showcased that we have 13 columns in the data. To understand the actual column names, we would need to refer to the data dictionary provided by CMU. You can find more details about the dataset here: http://lib.stat.cmu.edu/datasets/boston.

The description for the features in the data is showcased in the following list. The last row in the list refers to the label or the actual house price in our use case.

Column Name	Description
CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq. ft.
INDUS	proportion of nonretail business acres per town
CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX	nitric oxide concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centers
RAD	index of accessibility to radial highways
TAX	full-value property tax rate per $10,000
PTRATIO	pupil-teacher ratio by town
B	1000(Bk – 0.63)^2, where Bk is the proportion of blacks by town
LSTAT	% lower status of the population
MEDV	median value of owner-occupied homes in $1000's

To look at the contents of the training dataset, we can use the index-slicing option provided by Python's numpy library for the numpy n-dimensional arrays.

```
x_train[:3,:]
```

Output

```
array([[1.23247e+00, 0.00000e+00, 8.14000e+00, 0.00000e+00,
        5.38000e-01, 6.14200e+00, 9.17000e+01, 3.97690e+00,
        4.00000e+00, 3.07000e+02, 2.10000e+01, 3.96900e+02,
        1.87200e+01],
       [2.17700e-02, 8.25000e+01, 2.03000e+00, 0.00000e+00,
        4.15000e-01, 7.61000e+00, 1.57000e+01, 6.27000e+00,
        2.00000e+00, 3.48000e+02, 1.47000e+01, 3.95380e+02,
        3.11000e+00],
       [4.89822e+00, 0.00000e+00, 1.81000e+01, 0.00000e+00,
        6.31000e-01, 4.97000e+00, 1.00000e+02, 1.33250e+00,
        2.40000e+01, 6.66000e+02, 2.02000e+01, 3.75520e+02,
        3.26000e+00]])
```

All columns have numeric values, so there is no need for data transformation. Usually, once we have imported the dataset, we will need to extensively explore the data and will almost always clean, process, and augment it before we can start developing the models.

But for now, we will directly go ahead with a simple model and see what the results look like.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation

#Extract the last 100 rows from the training data to create the
validation datasets.
x_val = x_train[300:,]
y_val = y_train[300:,]
```

```
#Define the model architecture
model = Sequential()
model.add(Dense(13, input_dim=13, kernel_initializer='normal',
activation='relu'))
model.add(Dense(6, kernel_initializer='normal',
activation='relu'))
model.add(Dense(1, kernel_initializer='normal'))

# Compile model
model.compile(loss='mean_squared_error', optimizer='adam',
metrics=['mean_absolute_percentage_error'])

#Train the model
model.fit(x_train, y_train, batch_size=32, epochs=3,
validation_data=(x_val,y_val))
```

Output

```
Train on 404 samples, validate on 104 samples
Epoch 1/3
404/404 [==============================] - 2s 4ms/step - loss:
598.8595 - mean_absolute_percentage_error: 101.7889 - val_loss:
681.4912 - val_mean_absolute_percentage_error: 100.0789
Epoch 2/3
404/404 [==============================] - 0s 81us/step - loss:
583.6991 - mean_absolute_percentage_error: 99.7594 - val_loss:
674.8345 - val_mean_absolute_percentage_error: 99.2616
Epoch 3/3
404/404 [==============================] - 0s 94us/step - loss:
573.6101 - mean_absolute_percentage_error: 98.3180 - val_loss:
654.3787 - val_mean_absolute_percentage_error: 96.9662
```

We have created a simple two-hidden-layer model for the regression use case. We have chosen MAPE as the metric. Generally, this is not the best metric to choose for studying model performance, but its advantage is simplicity in terms of comprehending the results. It gives a simple percentage value for the error, say 10% error. So, if you know the average range of your prediction, you can easily estimate what the predictions are going to look like.

Let's now train the model and use the evaluate function to study the results of the model.

```
results = model.evaluate(x_test, y_test)

for i in range(len(model.metrics_names)):
    print(model.metrics_names[i]," : ", results[i])
```

Output

```
102/102 [==============================] - 0s 87us/step
loss  :  589.7658882889093
mean_absolute_percentage_error  :  96.48218611174939
```

We can see that MAPE is around 96%, which is actually not a great number to have for model performance. This would translate into our model predictions at around 96% error. So, in general, if a house was priced at 10K, our model would have predicted ~20K.

In DL, the model updates weight after every iteration and evaluates after every epoch. Since the updates are quite small, it usually takes a fairly higher number of epochs for a generic model to learn appropriately. To test the performance once again, let's increase the number of epochs to 30 instead of 3. This would increase the computation significantly and might take a while to execute. But since this is a fairly small dataset, training with 30 epochs should not be a problem. It should execute in ~1 min on your system.

```
#Train the model
model.fit(x_train, y_train, batch_size=32, epochs=30,
validation_data=(x_val,y_val))
```

Output

```
Train on 404 samples, validate on 104 samples
Epoch 1/1000
404/404 [==============================] - 0s 114us/step -
loss: 536.6662 - mean_absolute_percentage_error: 93.4381 - val_
loss: 580.3155 - val_mean_absolute_percentage_error: 88.6968
Epoch 2/1000
404/404 [==============================] - 0s 143us/step -
loss: 431.7025 - mean_absolute_percentage_error: 79.0697 - val_
loss: 413.4064 - val_mean_absolute_percentage_error: 67.0769
```

Skipping the output for in-between epochs.

(Adding output for only the last three epochs, i.e., 28 to 30)

```
Epoch 28/30
404/404 [==============================] - 0s 111us/step -
loss: 6.0758 - mean_absolute_percentage_error: 9.5185 - val_
loss: 5.2524 - val_mean_absolute_percentage_error: 8.3853
Epoch 29/30
404/404 [==============================] - 0s 100us/step -
loss: 6.2895 - mean_absolute_percentage_error: 10.1037 - val_
loss: 6.0818 - val_mean_absolute_percentage_error: 8.9386
Epoch 30/30
404/404 [==============================] - 0s 111us/step -
loss: 6.0761 - mean_absolute_percentage_error: 9.8201 - val_
loss: 7.3844 - val_mean_absolute_percentage_error: 8.9812
```

If we take a closer look at the loss and MAPE for the validation datasets, we can see a significant improvement. It has reduced from 96% in the previous example to 8.9% now.

Let's have a look at the test results.

```
results = model.evaluate(x_test, y_test)

for i in range(len(model.metrics_names)):
    print(model.metrics_names[i]," : ", results[i])
```

Output

```
102/102 [==============================] - 0s 92us/step
loss  :  22.09559840782016
mean_absolute_percentage_error  :  16.22196163850672
```

We can see that the results have improved significantly, but there still seems to be a significant gap between the MAPE for validation dataset and the test dataset. As discussed earlier, this gap is an indicator that the model has overfit, or in simple terms, has overcomplicated the process of learning. We will look in detail at the steps to reduce overfitting in DNNs in the next chapter for a bigger and better use case. For now, we have successfully explored Keras on a real dataset (though a small one) and used our learnings on the building blocks of DL in Keras.

Summary

In this chapter, we explored Keras in depth with hands-on exercises as well as contextual depth of topics. We studied the basic building blocks of DL and its implementation in Keras. We looked at how we can combine the different building blocks together in using Keras to develop DNN models. In the next chapter, we will start exploring a real use case step by step by exploring, cleaning, extracting, and applying the necessary transformations to get the data ready for developing DL models.