2
Keras, a High-Level API for TensorFlow 2

In this chapter, we will discuss Keras, which is a high-level API for TensorFlow 2. Keras was developed by François Chollet at Google. Keras has become extremely popular for fast prototyping, for building and training deep learning models, and for research and production. Keras is a very rich API; it supports eager execution and data pipelines, and other features, as we will see.

Keras has been available for TensorFlow since 2017, but its use has been extended and further integrated into TensorFlow with the release of TensorFlow 2.0. TensorFlow 2.0 has embraced Keras as the API of choice for the majority of deep learning development work.

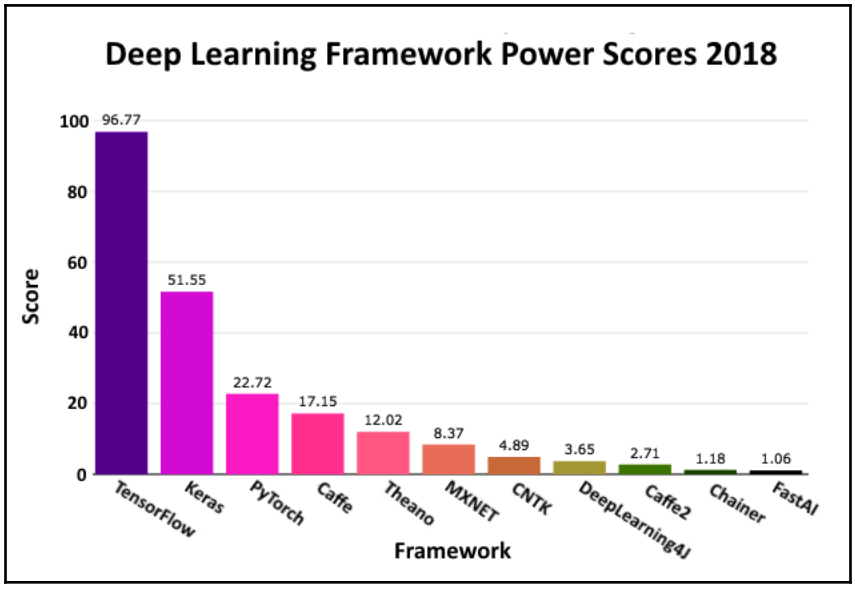
It is possible to import Keras as a standalone module, but in this book, we will concentrate on using Keras from within TensorFlow 2. The module is, thus, `tensorflow.keras`.

In this chapter, we will cover the following topics:

- The adoption and advantages of Keras
- The features of Keras
- The default Keras configuration file
- The Keras backend
- Keras data types
- Keras models
- Keras datasets

The adoption and advantages of Keras

Keras is widely used by industry and research, as shown in the following diagram. The *Power Score* rankings were devised by Jeff Hale, who used 11 data sources across 7 distinct categories to gauge framework usage, interest, and popularity. Then, he weighted and combined the data as shown in this Medium article from September 2018: `https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a`:



Keras has a number of advantages, including the following:

- It's designed for both new users and experts alike, offering consistent and simple APIs
- It's user friendly with a simple, consistent interface that is optimized for common use cases
- It provides excellent feedback for user errors that is are easily understood and often accompanied by helpful advice
- It's modular and composable; models in Keras are constructed by joining up configurable building blocks
- It's easy to extend by writing custom building blocks
- It is not necessary to import Keras as it is available as `tensorflow.keras`

The features of Keras

If you want to know which version of Keras came with your TensorFlow, use the following command:

```
import tensorflow as tf
print(tf.keras.__version__)
```

At the time of writing, this produced the following (from the alpha build of TensorFlow 2):

```
2.2.4-tf
```

Other features of Keras include built-in support for multi-GPU data parallelism, and also the fact that Keras models can be turned into TensorFlow Estimators and trained on clusters of GPUs on Google Cloud.

Keras is, perhaps, unusual in that it is has a reference implementation maintained as an independent open source project, located at `www.keras.io`.

It's maintained independently of TensorFlow, although TensorFlow does have a full implementation of Keras in the `tf.keras` module. The implementation has TensorFlow-specific augmentations, including support for eager execution, by default.

Eager execution means that execution of code is an imperative programming environment, rather than a graph-based environment, which was the only way to work in the initial offering of TensorFlow (prior to v1.5). This imperative (that is, immediate) style allows for intuitive debugging, fast development iteration, support for the TensorFlow `SavedModel` format, and built-in support for distributed training on CPUs, GPUs, and even Google's own hardware, **Tensor Processing Units** (**TPUs**).

The TensorFlow implementation also has support for `tf.data`, distribution strategies, exporting models, which can be deployed on mobile and embedded devices via TensorFlow Lite, and feature columns for representing and classifying structured data.

The default Keras configuration file

The default configuration file for Linux users is as follows:

```
$HOME/.keras/keras.json
```

For Windows users, replace `$HOME` with `%USERPROFILE%`.

It is created the first time you use Keras, and may be edited to change the defaults. Here's what that `.json` file contains:

```
{ "image_data_format": "channels_last",
"epsilon": 1e-07,
"floatx": "float32",
"backend": "tensorflow" }
```

The defaults are as follows:

- `image_data_format`: This is a string, either `"channels_last"` or `channels_first`, for the image format. Keras running on top of TensorFlow uses the default.
- `epsilon`: This is a float, being a numeric *fuzzing* constant used to avoid division by zero in some operations.
- `floatx`: This is a string and specifies the default float precision, being one of `"float16"`, `"float32"`, or `"float64"`.
- `backend`: This is a string and specifies the tool that Keras finds itself on top of, being one of `"tensorflow"`, `"theano"`, or `"cntk"`.

There are getter and setter methods, available in `keras.backend`, for all these values; see `https://keras.io/backend/`.

For example, in the following sets, the floating point type for Keras to use is `floatx`, where the `floatx` argument is one of the three precisions shown in the following command:

```
keras.backend.set_floatx(floatx)
```

The Keras backend

Due to its model-level library structure, Keras may have different tensor manipulation engines that handle low-level operations, such as convolutions, tensor products, and the like. These engines are called **backends**. Other backends are available; we will not consider them here.

The same `https://keras.io/backend/` link takes you to a wealth of `keras.backend` functions.

The canonical way to use the Keras `backend` is with the following:

```
from keras import backend as K
```

For example, here is the signature of a useful function:

```
K.constant(value, dtype=None, shape=None, name=None)
```

Here `value` is the value to be given to the constant, `dtype` is the type of the tensor that is created, `shape` is the shape of the tensor that is created, and `name` is an optional name.

An instantiation of this is as follows:

```
from tensorflow.keras import backend as K
const = K.constant([[42,24],[11,99]], dtype=tf.float16, shape=[2,2])
const
```

This produces the following constant tensor. Notice that, because eager execution is enabled, (by default) the value of the constant is given in the output:

```
<tf.Tensor: id=1, shape=(2, 2), dtype=float16, numpy= array([[42., 24.],
[11., 99.]], dtype=float16)>
```

Were eager not to be enabled, the output would be as follows:

```
<tf.Tensor 'Const:0' shape=(2, 2) dtype=float16>
```

Keras data types

Keras **data types** (**dtypes**) are the same as TensorFlow Python data types, as shown in the following table:

Python type	Description
`tf.float16`	16-bit floating point
`tf.float32`	32-bit floating point
`tf.float64`	64-bit floating point
`tf.int8`	8-bit signed integer
`tf.int16`	16-bit signed integer
`tf.int32`	32-bit signed integer
`tf.int64`	64-bit signed integer
`tf.uint8`	8-bit unsigned integer
`tf.string`	Variable-length byte arrays

Python type	Description
`tf.bool`	Boolean
`tf.complex64`	Complex number made of two 32-bit floating points—one real and imaginary part
`tf.complex128`	Complex number made of two 64-bit floating points—one real and one imaginary part
`tf.qint8`	8-bit signed integer used in quantized Ops
`tf.qint32`	32-bit signed integer used in quantized Ops
`tf.quint8`	8-bit unsigned integer used in quantized Ops

Keras models

Keras is based on the concept of a neural network model. The predominant model is called a **Sequence**, being a linear stack of layers. There is also a system using the Keras functional API.

The Keras Sequential model

To build a Keras `Sequential` model, you *add* layers to it in the same order that you want the computations to be undertaken by the network.

After you have built your model, you *compile* it; this optimizes the computations that are to be undertaken, and is where you allocate the optimizer and the loss function you want your model to use.

The next stage is to *fit* the model to the data. This is commonly known as training the model, and is where all the computations take place. It is possible to present the data to the model either in batches, or all at once.

Next, you evaluate your model to establish its accuracy, loss, and other metrics. Finally, having trained your model, you can use it to make predictions on new data. So, the workflow is: build, compile, fit, evaluate, make predictions.

There are two ways to create a `Sequential` model. Let's take a look at each of them.

The first way to create a Sequential model

Firstly, you can pass a list of layer instances to the constructor, as in the following example.

We will have much more to say about layers in the next chapter; for now, we will just explain enough to allow you to understand what is happening here.

Acquire the data. `mnist` is a dataset of hand-drawn numerals, each on a 28 x 28 pixel grid. Every individual data point is an unsigned 8-bit integer (`uint8`), as are the labels:

```
mnist = tf.keras.datasets.mnist
(train_x,train_y), (test_x, test_y) = mnist.load_data()
```

The `epochs` variable stores the number of times we are going to present the data to the model:

```
epochs=10
batch_size = 32 # 32 is default in fit method but specify anyway
```

Next, normalize all the data points (`x`) to be in the float range zero to one, and of the `float32` type. Also, cast the labels (`y`) to `int64`, as required:

```
train_x, test_x = tf.cast(train_x/255.0, tf.float32), tf.cast(test_x/255.0,
tf.float32)
train_y, test_y = tf.cast(train_y,tf.int64),tf.cast(test_y,tf.int64)
```

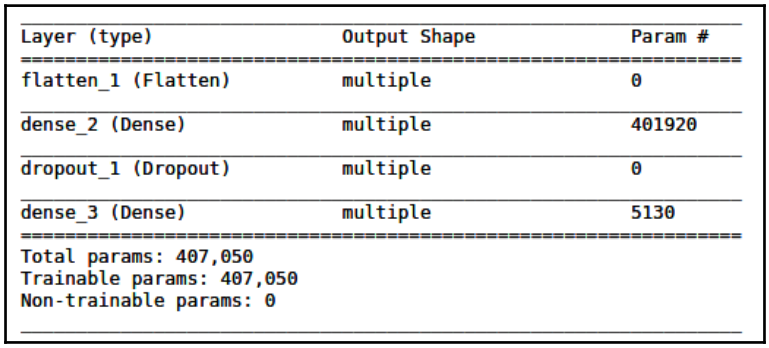
The model definition follows.

Notice in the model definition how we are passing a list of layers:

- `Flatten` takes the input of 28 x 28 (that is, 2D) pixel images and produces a 784 (that is, 1D) vector because the next (dense) layer is one-dimensional.
- `Dense` is a fully connected layer, meaning all its neurons are connected to every neuron in the previous and next layers. The following example has 512 neurons, and its inputs are passed through a ReLU (non-linear) activation function.
- `Dropout` randomly turns off a fraction (in this case, 0.2) of neurons in the previous layer. This is done to prevent any particular neuron becoming too specialized and causing *overfitting* of the model to the data, thus impacting on the accuracy metric of the model on the test data (much more on this in later chapters).
- The final `Dense` layer has a special activation function called `softmax`, which assigns probabilities to each of the possible 10 output units:

```
model1 = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(),
 tf.keras.layers.Dense(512,activation=tf.nn.relu),
 tf.keras.layers.Dropout(0.2),
 tf.keras.layers.Dense(10,activation=tf.nn.softmax)
])
```

The `model.summary()` function is a useful eponymous method and gives the following output for our model:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          multiple                  0
_________________________________________________________________
dense_2 (Dense)              multiple                  401920
_________________________________________________________________
dropout_1 (Dropout)          multiple                  0
_________________________________________________________________
dense_3 (Dense)              multiple                  5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

The figure of `401920` comes from input = 28 x 28 = 784 x 512 (`dense_2` layer) = giving *784 *512 = 401,408* together with a bias unit for each of the `dense_1` layer neurons, giving *401,408 + 512 = 401,920*.

The figure of `5130` comes in a similar way, from *512* 10 + 10 = 5,130*.

Next, we compile our model, as shown in the following code:

```
optimiser = tf.keras.optimizers.Adam()
model1.compile (optimizer= optimiser,
loss='sparse_categorical_crossentropy', metrics = ['accuracy'])
```

`optimizer` is the method by which the weights of the weighted connections in the model are adjusted to decrease the loss.

`loss` is a measure of the difference between the required output of the model and the actual output, and `metrics` is how we evaluate the model.

To train our model, we use the `fit` method next, shown as follows:

```
model1.fit(train_x, train_y, batch_size=batch_size, epochs=epochs)
```

The output from the call to `fit()` is as follows, showing the epoch training time, the loss, and the accuracy:

```
Epoch 1/10 60000/60000 [==============================] - 5s 77us/step -
loss: 0.2031 - acc: 0.9394
...
Epoch 10/10 60000/60000 [==============================] - 4s 62us/step -
loss: 0.0098 - acc: 0.9967
```

And finally, we can check our trained model for accuracy using the `evaluate` method:

```
model1.evaluate(test_x, test_y)
```

This produces the following output:

```
10000/10000 [==============================] - 0s 39us/step
[0.09151900197149189, 0.9801]
```

This represents a loss of 0.09 and an accuracy of 0.9801 on the test data. An accuracy of 0.98 means that out of 100 test data points, 98 were, on average, correctly identified by the model.

The second way to create a Sequential model

The alternative to passing a list of layers to the `Sequential` model's constructor is to use the `add` method, as follows, for the same architecture:

```
model2 = tf.keras.models.Sequential();
model2.add(tf.keras.layers.Flatten())
model2.add(tf.keras.layers.Dense(512, activation='relu'))
model2.add(tf.keras.layers.Dropout(0.2))
model2.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))
model2.compile (optimizer= tf.keras.Adam(),
loss='sparse_categorical_crossentropy',
 metrics = ['accuracy'])
```

The `fit()` method performs the training, as we have seen, fitting the inputs to the outputs using the model:

```
model2.fit(train_x, train_y, batch_size=batch_size, epochs=epochs)
```

Then, we evaluate the performance of our model using the `test` data:

```
model2.evaluate(test_x, test_y)
```

This gives us a loss of `0.07` and an accuracy of `0.981`.

Thus, this method of defining a model produces an almost identical result to the first one, as is to be expected, since it is the same architecture, albeit expressed slightly differently, with the same `optimizer` and `loss` functions. Now let's take a look at the functional API.

The Keras functional API

The functional API lets you build much more complex architectures than the simple linear stack of `Sequential` models we have seen previously. It also supports more advanced models. These models include multi-input and multi-output models, models with shared layers, and models with residual connections.

Here is a short example, with an identical architecture to the previous two, of the use of the functional API.

The setup code is the same as previously demonstrated:

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(train_x,train_y), (test_x, test_y) = mnist.load_data()
train_x, test_x = train_x/255.0, test_x/255.0
epochs=10
```

Here is the model definition.

Notice how layers are callable on `tensor` and return a tensor as output, and how these input and output tensors are then used to define a model:

```
inputs = tf.keras.Input(shape=(28,28)) # Returns a 'placeholder' tensor
x = tf.keras.layers.Flatten()(inputs)
x = tf.layers.Dense(512, activation='relu',name='d1')(x)
x = tf.keras.layers.Dropout(0.2)(x)
predictions = tf.keras.layers.Dense(10,activation=tf.nn.softmax,
name='d2')(x)

model3 = tf.keras.Model(inputs=inputs, outputs=predictions)
```

Notice how this code produces an identical architecture to that of `model1` and `model2`:

```
model3.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         (None, 28, 28)            0
_________________________________________________________________
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
d1 (Dense)                   (None, 512)               401920
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
d2 (Dense)                   (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

The **None** appears here because we haven't specified how many input items we have (that is, the batch size). It really means *not given*.

The rest of the code is the same as in the previous example:

```
optimiser = tf.keras.optimizers.Adam()
model3.compile (optimizer= optimiser,
loss='sparse_categorical_crossentropy', metrics = ['accuracy'])

model3.fit(train_x, train_y, batch_size=32, epochs=epochs)

model3.evaluate(test_x, test_y)
```

This gives a loss of `0.067` and an accuracy of `0.982`, again as expected, for an identical architecture.

Next, let's see how to subclass the Keras `model` class.

Subclassing the Keras Model class

The Keras `Model` class may be subclassed as shown in the code that follows. Google states that a *pure* functional style (as in the preceding example) is to be preferred over the subclassing style (we have included it here for completeness, and because it is interesting).

Firstly, notice how the layers are individually declared and named in the constructor (`.__init__()`).

Then, notice how the layers are chained together in the functional style in the `call()` method. This method encapsulates what is referred to as the *forward pass*:

```
class MyModel(tf.keras.Model):
 def __init__(self, num_classes=10):
  super(MyModel, self).__init__()
 # Define your layers here.
   inputs = tf.keras.Input(shape=(28,28)) # Returns a placeholder tensor
   self.x0 = tf.keras.layers.Flatten()
   self.x1 = tf.keras.layers.Dense(512, activation='relu',name='d1')
   self.x2 = tf.keras.layers.Dropout(0.2)
   self.predictions = tf.keras.layers.Dense(10,activation=tf.nn.softmax,
name='d2')

 def call(self, inputs):
 # This is where to define your forward pass
 # using the layers previously defined in `__init__`
   x = self.x0(inputs)
```

```
        x = self.x1(x)
        x = self.x2(x)
        return self.predictions(x)

model4 = MyModel()
```

This definition may be used in place of any of the earlier model definitions in this chapter with identical supporting code for data download, and similar code for training/evaluation. This last example is shown in the following code:

```
model4 = MyModel()
batch_size = 32
steps_per_epoch = len(train_x.numpy())//batch_size
print(steps_per_epoch)

model4.compile (optimizer= tf.keras.Adam(),
loss='sparse_categorical_crossentropy',
 metrics = ['accuracy'])

model4.fit(train_x, train_y, batch_size=batch_size, epochs=epochs)

 model4.evaluate(test_x, test_y)
```

The results of this are a loss of `0.068` and accuracy is `0.982`; again, virtually identical to the results produced by the other three model-building styles in this chapter.

Using data pipelines

Data may also be passed into the `fit` method as a `tf.data.Dataset()` iterator, using the following code (the data acquisition code is identical to that previously described). The `from_tensor_slices()` method converts the NumPy arrays into a dataset. Notice the `batch()` and `shuffle()` methods chained together. Next, the `map()` method invokes a method on the input images, `x`, that randomly flips one in two of them across the *y*-axis, effectively increasing the size of the image set. The labels, `y`, are left unaltered here. Finally, the `repeat()` method means that the dataset will be re-fed from the beginning when its end is reached (continuously):

```
batch_size = 32
buffer_size = 10000

train_dataset = tf.data.Dataset.from_tensor_slices((train_x,
train_y)).batch(32).shuffle(10000)

train_dataset = train_dataset.map(lambda x, y:
```

```
(tf.image.random_flip_left_right(x), y))
train_dataset = train_dataset.repeat()
```

The code for the `test` set is similar except that no flipping is done:

```
test_dataset = tf.data.Dataset.from_tensor_slices((test_x,
test_y)).batch(batch_size).shuffle(10000)

test_dataset = train_dataset.repeat()
```

Now in the `fit()` function, we can pass the dataset directly in, as follows:

```
steps_per_epoch = len(train_x)//batch_size # required because of the repeat
on the dataset
optimiser = tf.keras.optimizers.Adam()
model5.compile (optimizer= optimiser,
loss='sparse_categorical_crossentropy', metrics = ['accuracy'])
model.fit(train_dataset, batch_size=batch_size, epochs=epochs,
steps_per_epoch=steps_per_epoch)
```

The compile and evaluate code is similar to that previously seen.

The advantages of using `data.Dataset` iterators are that the pipeline takes care of much of the plumbing that normally goes into preparing data, such as batching and shuffling, as we have seen. The various operations can be chained together, as we have also seen.

Saving and loading Keras models

The Keras API in TensorFlow has the ability to save and restore models easily. This is done as follows, and saves the model in the current directory. Of course, a longer path may be passed here:

```
model.save('./model_name.h5')
```

This will save the model architecture, its weights, its training state (`loss`, `optimizer`), and the state of the optimizer, so that you can carry on training the model from where you left off.

Loading a saved model is done as follows. Note that if you have compiled your model, the load will compile your model using the saved training configuration:

```
from tensorflow.keras.models import load_model
new_model = load_model('./model_name.h5')
```

It is also possible to save just the model weights and load them with this (in which case, you must build your architecture to load the weights into):

```
model.save_weights('./model_weights.h5')
```

Then use the following to load it:

```
model.load_weights('./model_weights.h5')
```

Keras datasets

The following datasets are available from within Keras: `boston_housing`, `cifar10`, `cifar100`, `fashion_mnist`, `imdb`, `mnist`, and `reuters`.

They are all accessed with the `load_data()` function. For example, to load the `fashion_mnist` dataset, use the following:

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.fashion_mnist.load_data()
```

Further details may be found at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/datasets/`.

Summary

In this chapter, we explored the Keras API with general comments and insights followed by the same basic architecture expressed in four different ways, for training with the `mnist` dataset.

In the next chapter, we will start to use TensorFlow in earnest by exploring a number of supervised learning scenarios, including linear regression, logistic regression, and k-nearest neighbors.

3
ANN Technologies Using TensorFlow 2

In this chapter, we will discuss and exemplify those parts of TensorFlow 2 that are needed for the construction, training, and evaluation of artificial neural networks and their utilization purposes for inference. Initially, we will not present a complete application. Instead, we will focus on individual concepts and techniques before putting them all together and presenting full models in subsequent chapters.

In this chapter, we will cover the following topics:

- Presenting data to an **Artificial Neural Network** (**ANN**)
- ANN layers
- Gradient calculations for gradient descent algorithms
- Loss functions

Presenting data to an ANN

The canonical way to present data to a TensorFlow ANN, as recommended by Google, is via a data pipeline composed of a `tf.data.Dataset` object and a `tf.data.Iterator` method. A `tf.data.Dataset` object consists of a sequence of elements in which each element contains one or more tensor objects. The `tf.data.Iterator` is a method used to loop over a dataset so that successive individual elements in it may be accessed.

We will look at two important ways of constructing a data pipeline, firstly, from in-memory **NumPy** arrays, and, secondly, from **Comma-Separated Value** (**CSV**) files. We will also look at a binary TFRecord format.

Using NumPy arrays with datasets

Let's look at some straightforward examples first. Here is a NumPy array:

```
import tensorflow as tf
import numpy as np

num_items = 11
num_list1 = np.arange(num_items)
num_list2 = np.arange(num_items,num_items*2)
```

This is how to create datasets, using the `from_tensor_slices()` method:

```
num_list1_dataset = tf.data.Dataset.from_tensor_slices(num_list1)
```

This is how to create an `iterator` on it using the `make_one_shot_iterator()` method:

```
iterator = tf.compat.v1.data.make_one_shot_iterator(num_list1_dataset)
```

And this is how to use them together, using the `get_next` method:

```
for item in num_list1_dataset:
    num = iterator1.get_next().numpy()
    print(num)
```

Note that executing this code twice in the same program run will raise an error because we are using a **one-shot** iterator.

It's also possible to access the data in batches with the `batch` method. Note that the first argument is the number of elements to put in each batch and the second is the self-explanatory `drop_remainder` argument:

```
num_list1_dataset = tf.data.Dataset.from_tensor_slices(num_list1).batch(3,
drop_remainder = False)
iterator = tf.compat.v1.data.make_one_shot_iterator(num_list1_dataset)
for item in num_list1_dataset:
    num = iterator.get_next().numpy()
    print(num)
```

There is also a `zip` method, which is useful for presenting features and labels together:

```
dataset1 = [1,2,3,4,5]
dataset2 = ['a','e','i','o','u']
dataset1 = tf.data.Dataset.from_tensor_slices(dataset1)
dataset2 = tf.data.Dataset.from_tensor_slices(dataset2)
zipped_datasets = tf.data.Dataset.zip((dataset1, dataset2))
iterator = tf.compat.v1.data.make_one_shot_iterator(zipped_datasets)
for item in zipped_datasets:
    num = iterator.get_next()
    print(num)
```

We can concatenate two datasets as follows, using the `concatenate` method:

```
ds1 = tf.data.Dataset.from_tensor_slices([1,2,3,5,7,11,13,17])
ds2 = tf.data.Dataset.from_tensor_slices([19,23,29,31,37,41])
ds3 = ds1.concatenate(ds2)
print(ds3)
iterator = tf.compat.v1.data.make_one_shot_iterator(ds3)
for i in range(14):
  num = iterator.get_next()
  print(num)
```

We can also do away with iterators altogether, as shown here:

```
epochs=2
for e in range(epochs):
  for item in ds3:
    print(item)
```

Note that the outer loop here does not raise an error, and so would be the preferred method to use in most circumstances.

Using comma-separated value (CSV) files with datasets

CSV files are a very popular method of storing data. TensorFlow 2 contains flexible methods for dealing with them. The main method here is `tf.data.experimental.CsvDataset`.

CSV example 1

With the following arguments, our dataset will consist of two items taken from each row of the `filename` file, both of the float type, with the first line of the file ignored and columns 1 and 2 used (column numbering is, of course, 0-based):

```
filename = ["./size_1000.csv"]
record_defaults = [tf.float32] * 2 # two required float columns
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header=True, select_cols=[1,2])
for item in dataset:
  print(item)
```

CSV example 2

In this example, and with the following arguments, our dataset will consist of one required float, one optional float with a default value of `0.0`, and an `int`, where there is no header in the CSV file and only columns 1, 2, and 3 are imported:

```
#file Chapter_2.ipynb
filename = "mycsvfile.txt"
record_defaults = [tf.float32, tf.constant([0.0], dtype=tf.float32),
tf.int32,]
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header=False, select_cols=[1,2,3])
for item in dataset:
  print(item)
```

CSV example 3

For our final example, our `dataset` will consist of two required floats and a required string, where the CSV file has a `header` variable:

```
filename = "file1.txt"
record_defaults = [tf.float32, tf.float32, tf.string ,]
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header=False)
or item in dataset:
    print(item[0].numpy(), item[1].numpy(),item[2].numpy().decode() )
# decode as string is in binary format.
```

TFRecords

Another popular choice for storing data is the TFRecord format. This is a binary file format. For large files, it is a good choice because binary files take up less disc space, take less time to copy, and can be read very efficiently from the disc. All this can have a significant effect on the efficiency of your data pipeline and, thus, the training time of your model. The format is also optimized in a variety of ways for use with TensorFlow. It is a little complex because data has to be converted into the binary format prior to storage and decoded when read back.

TFRecord example 1

The first example we show here will demonstrate the bare bones of the techniques. (The file is `TFRecords.ipynb`).

Because a TFRecord file is a sequence of binary strings, its structure must be specified prior to saving so that it can be properly written and subsequently read back. TensorFlow has two structures for this, `tf.train.Example` and `tf.train.SequenceExample`. What you have to do is store each sample of your data in one of these structures, then serialize it, and use `tf.python_io.TFRecordWriter` to save it to disk.

In the following example, the float array, `data`, is converted to the binary format and then saved to disc. A `feature` is a dictionary containing the data that is passed to `tf.train.Example` prior to serialization and saving. A more elaborate example of this is shown in *TFRecord example 2*:

The byte data types supported by TFRecords are `FloatList`, `Int64List`, and `BytesList`.

```
# file: TFRecords.ipynb
import tensorflow as tf
import numpy as np

data=np.array([10.,11.,12.,13.,14.,15.])

def npy_to_tfrecords(fname,data):
    writer = tf.io.TFRecordWriter(fname)
    feature={}
    feature['data'] =
tf.train.Feature(float_list=tf.train.FloatList(value=data))
    example = tf.train.Example(features=tf.train.Features(feature=feature))
```

```
        serialized = example.SerializeToString()
        writer.write(serialized)
        writer.close()

npy_to_tfrecords("./myfile.tfrecords",data)
```

The code to read the record back is as follows. A `parse_function` function is constructed that decodes the dataset read back from the file. This requires a dictionary (`keys_to_features`) with the same name and structure as the saved data:

```
dataset = tf.data.TFRecordDataset("./myfile.tfrecords")

def parse_function(example_proto):
 keys_to_features = {'data':tf.io.FixedLenSequenceFeature([], dtype =
tf.float32, allow_missing = True) }
    parsed_features = tf.io.parse_single_example(serialized=example_proto,
features=keys_to_features)
    return parsed_features['data']

dataset = dataset.map(parse_function)
iterator = tf.compat.v1.data.make_one_shot_iterator(dataset)
# array is retrieved as one item
item = iterator.get_next()
print(item)
print(item.numpy())
print(item[2].numpy())
```

TFRecord example 2

In this example, we look at a more complicated record structure given by this dictionary:

```
filename = './students.tfrecords'
data = {
            'ID': 61553,
            'Name': ['Jones', 'Felicity'],
            'Scores': [45.6, 97.2]
        }
```

Using this, we can construct a `tf.train.Example` class, again using the `Feature()` method. Note how we have to encode our string:

```
ID = tf.train.Feature(int64_list=tf.train.Int64List(value=[data['ID']]))

Name =
tf.train.Feature(bytes_list=tf.train.BytesList(value=[n.encode('utf-8') for
n in data['Name']]))
```

```
Scores =
tf.train.Feature(float_list=tf.train.FloatList(value=data['Scores']))

example = tf.train.Example(features=tf.train.Features(feature={'ID': ID,
'Name': Name, 'Scores': Scores }))
```

Serializing and writing this record to disc is the same as *TFRecord example 1*:

```
writer = tf.io.TFRecordWriter(filename)
writer.write(example.SerializeToString())
writer.close()
```

To read this back, we just need to construct our `parse_function` function to reflect the structure of the record:

```
dataset = tf.data.TFRecordDataset("./students.tfrecords")

def parse_function(example_proto):
    keys_to_features = {'ID':tf.io.FixedLenFeature([], dtype = tf.int64),
                        'Name':tf.io.VarLenFeature(dtype = tf.string),
                         'Scores':tf.io.VarLenFeature(dtype = tf.float32)
                        }
    parsed_features = tf.io.parse_single_example(serialized=example_proto,
features=keys_to_features)
    return parsed_features["ID"],
parsed_features["Name"],parsed_features["Scores"]
```

The next step is the same as before:

```
dataset = dataset.map(parse_function)

iterator = tf.compat.v1.data.make_one_shot_iterator(dataset)
item = iterator.get_next()
# record is retrieved as one item
print(item)
```

The output is as follows:

```
(<tf.Tensor: id=264, shape=(), dtype=int64, numpy=61553>,
<tensorflow.python.framework.sparse_tensor.SparseTensor object at
0x7f1bfc7567b8>, <tensorflow.python.framework.sparse_tensor.SparseTensor
object at 0x7f1bfc771e80>)
```

Now we can extract our data from `item` (note that the string must be decoded (from bytes) where the default for our Python 3 is `utf8`). Note also that the string and the array of floats are returned as sparse arrays, and to extract them from the record, we use the sparse array `value` method:

```
print("ID: ",item[0].numpy())
name = item[1].values.numpy()
name1= name[0].decode()returned
name2 = name[1].decode('utf8')
print("Name:",name1,",",name2)
print("Scores: ",item[2].values.numpy())
```

One-hot encoding

One-hot encoding (**OHE**) is where a tensor is constructed from the data labels with a 1 in each of the elements corresponding to a label's value, and 0 everywhere else; that is, one of the bits in the tensor is hot (1).

OHE example 1

In this example, we are converting a decimal value of `5` to a one-hot encoded value of `0000100000` using the `tf.one_hot()` method:

```
y = 5
y_train_ohe = tf.one_hot(y, depth=10).numpy()
print(y, "is ",y_train_ohe,"when one-hot encoded with a depth of 10")
# 5 is 00000100000 when one-hot encoded with a depth of 10
```

OHE example 2

This is also nicely shown in the following example using the sample code that imports from the fashion MNIST dataset.

The original labels are integers from 0 to 9, so, for example, a label of `2` becomes `0010000000` when one-hot encoded, but note the difference between the index and the label stored at that index:

```
import tensorflow as tf
from tensorflow.python.keras.datasets import fashion_mnist
tf.enable_eager_execution()
width, height, = 28,28
```

```
n_classes = 10

# load the dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
split = 50000
#split feature training set into training and validation sets
(y_train, y_valid) = y_train[:split], y_train[split:]

# one-hot encode the labels using TensorFlow.
# then convert back to numpy for display
y_train_ohe = tf.one_hot(y_train, depth=n_classes).numpy()
y_valid_ohe = tf.one_hot(y_valid, depth=n_classes).numpy()
y_test_ohe = tf.one_hot(y_test, depth=n_classes).numpy()

# show difference between the original label and a one-hot-encoded label

i=5
print(y_train[i]) # 'ordinary' number value of label at index i=5 is 2
# 2
# note the difference between the index of 5 and the label at that index
which is 2
print(y_train_ohe[i]) #
# 0. 0. 1. 0. 0.0 .0 .0. 0. 0.
```

Next, we will examine the fundamental data structure of a neural network: the **layer** of neurons.

Layers

The fundamental data structure used by ANNs is the **layer**, and many interconnected layers make up a complete ANN. A layer can be envisaged as an array of neurons, although the use of the word *neuron* can be misleading, since there is only a marginal correspondence between human brain neurons and the artificial neurons that make up a layer. Bearing that in mind, we will use the term *neuron* in what follows. As with any computer processing unit, a neuron is characterized by its inputs and its outputs. In general, a neuron has many inputs and one output value. Each input connection carries a weight, $w_{i.}$

The following diagram shows a neuron. It is important to note that the activation function, **f**, is non-linear for anything other than trivial ANNs. A general neuron in the network receives inputs from other neurons and each of these carries a weight, $\mathbf{w_i}$, as shown, and the network *learns* by adjusting these weights so that the input generates the required output:

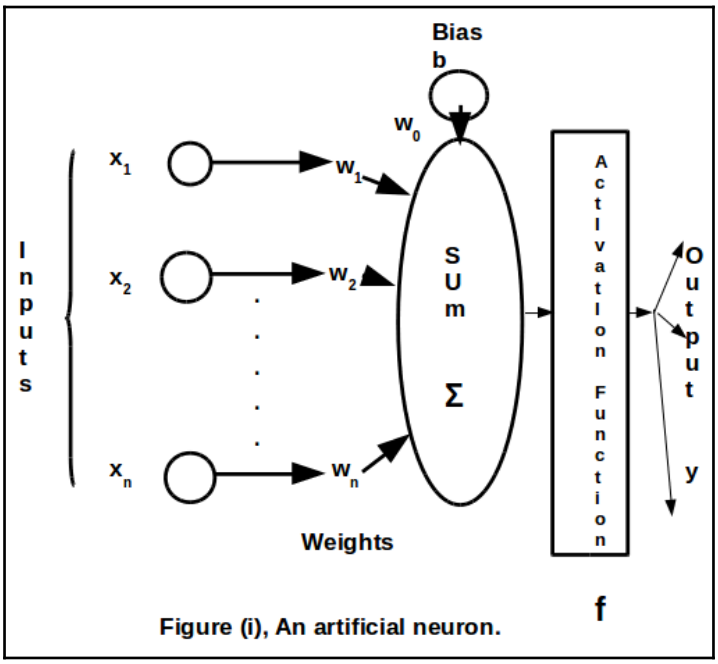


Figure 1: An artificial neuron

The output of a neuron is given by totaling the inputs multiplied by the weights, adding the bias multiplied by its weight, and then applying the activation function (refer to the following diagram).

The following diagram shows how individual artificial neurons and layers are configured to create an ANN:

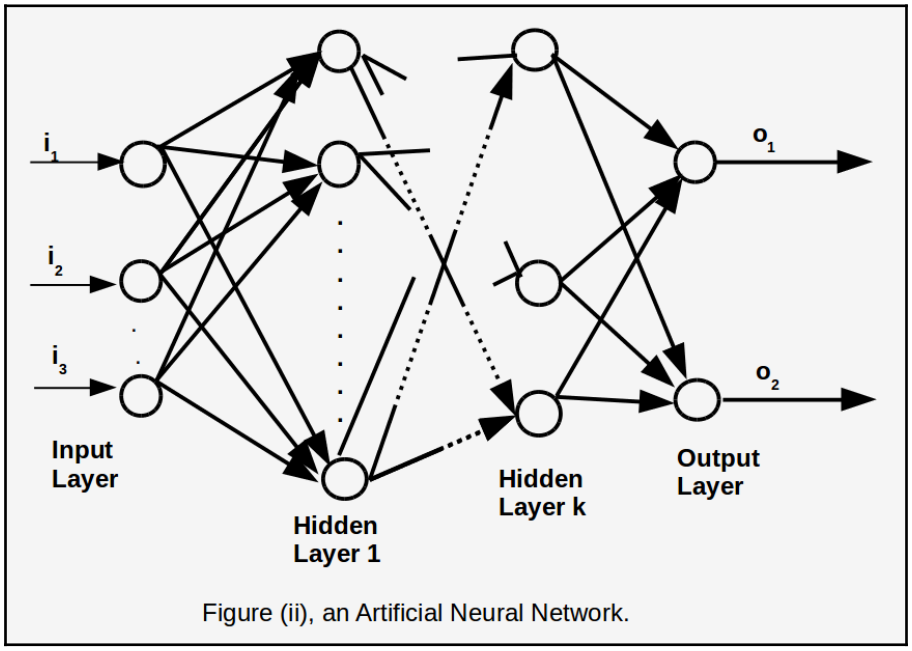


Figure 2: An artificial neural network

The output of a layer is given by the following formula:

$$output = f(\sum_{1}^{n} W.X + bias)$$

Here, W is the weights of the input, X is the input vector, and f is the non-linear activation function.

There are many types of layers, supporting a large variety of ANN model structures. A very comprehensive list may be found at `https://www.tensorflow.org/api_docs/python/tf/keras/layers`.

Here, we will examine a number of the more popular ones and how TensorFlow implements them.

Dense (fully connected) layer

A **dense layer** is a fully connected layer. This means that all neurons in the previous layer are connected to all neurons in the next layer. In a dense network, all layers are dense. (If a network has three or more hidden layers, it is known as a **deep network**).

A dense layer is constructed by the `layer = tf.keras.layers.Dense(n)` line, where `n` is the number of output units.

Note that a dense layer is one-dimensional. Please refer to the section on *Models*.

Convolutional layer

A **convolutional layer** is a layer where the neurons in a layer are grouped into small patches by the use of a filter, which is usually square, and created by sliding the filter over the layer. Each patch is *convolved*, that is, multiplied and summed by the filter. **Convolutional nets** or **ConvNets** for short, have proved themselves to be very good at image recognition and manipulation.

For images, a convolutional layer has the partial signature `tf.keras.layers.Conv2D(filters, kernel_size, strides=1, padding='valid')`.

So, in the following example, this first layer has one filter of size (1,1) and its padding is valid. The other padding possibility is *same*.

The difference is that, with the *same* padding, the layer must be padded, often with zeros, around the outside so that after the convolution has taken place, the output size is the same as the layer size. With valid padding, no padding is done and there will be some truncation of the layer if the combination of the stride and the kernel size won't fit exactly on to the layer. The output size is smaller than the layer that is being convolved:

```
seqtial_Net = tf.keras.Sequential([tf.keras.layers.Conv2D(   1, (1, 1),
strides = 1, padding='valid')
```

Max pooling layer

A **max pooling layer** takes the maximum value within its window as the window slides over the layer, in much the same way as a convolution takes place.

The max pooling signature for spatial data, that is, images, is as follows:

```
tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None,
padding='valid', data_format=None)
```

So, to use the defaults, you would simply have the following:

```
layer = tf.keras.maxPooling2D()
```

Batch normalization layer and dropout layer

Batch normalization is a layer that takes its inputs and outputs the same number of outputs with activations that have zero mean and unit variance, as this has been found to be beneficial to learning. Batch normalization regulates the activations so that they neither become vanishingly small nor explosively big, both of which situations prevent the network from learning.

The signature of the `BatchNormalization` layer is as follows:

```
tf.keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001,
center=True, scale=True, beta_initializer='zeros',
gamma_initializer='ones', moving_mean_initializer='zeros',
moving_variance_initializer='ones', beta_regularizer=None,
gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

Hence, to use the defaults, you would simply use the following command:

```
layer = tf.keras.layers.BatchNormalization()
```

Dropout layer is a layer where a certain percentage of the neurons are randomly turned off during training (not during inference). This forces the network to become better at generalizing since individual neurons are discouraged from becoming specialized with respect to their inputs.

The signature of the `Dropout` layer is as follows:

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

The `rate` argument is the fraction of neurons that are turned off.

So, to use this, you would have the following, for example:

```
layer = tf.keras.layers.Dropout(rate = 0.5)
```

Fifty percent of neurons, randomly chosen, would be turned off.

Softmax layer

A **softmax layer** is a layer where the activation of each output unit corresponds to the probability that the output unit matches a given label. The output neuron with the highest activation value is, therefore, the prediction of the net. It is used when the classes being learned are mutually exclusive, so that the probabilities output by the softmax layer total 1.

It is implemented as an activation on a dense layer.

Hence, for example, we have the following:

```
model2.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))
```

This would add a dense softmax layer with 10 neurons where the activations of the neurons would total 1.

Next, we will talk a little more about activation functions.

Activation functions

It is important to note that neural nets have non-linear activation functions, that is, the functions that are applied to the sum of the weighted inputs to a neuron. Linear activation units are not able to map input layers to output layers in all but trivial neural net models.

There are a number of activation functions in common use, including sigmoid, tanh, ReLU, and leaky ReLU. A good summary, together with diagrams of these functions, may be found here: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

Creating the model

There are four methods for creating our ANN model using Keras:

- **Method 1**: Arguments passed to `tf.keras.Sequential`
- **Method 2**: Use of the `.add` method of `tf.keras.Sequential`
- **Method 3**: Use of the Keras functional API
- **Method 4**: By subclassing the `tf.keras.Model` object

Please refer to `Chapter 2`, *Keras, a High-Level API for TensorFlow 2*, for details regarding these four methods.

Gradient calculations for gradient descent algorithms

One of TenorFlow's great strengths is its ability to automatically compute gradients for use in gradient descent algorithms, which, of course, are a vital part of most machine learning models. TensorFlow offers a number of methods for gradient calculations.

There are four ways to automatically compute gradients when eager execution is enabled (they also work in graph mode):

1. `tf.GradientTape`: Context records computations so that you can call `tf.gradient()` to get the gradients of any tensor computed while recording with respect to any trainable variable
2. `tfe.gradients_function()`: Takes a function (say `f()`) and returns a gradient function (say `fg()`) that can compute the gradients of the outputs of `f()` with respect to the parameters of `f()` or a subset of them
3. `tfe.implicit_gradients()`: This is very similar, but `fg()` computes the gradients of the outputs of `f()` with regard to all trainable variables that these outputs depend on
4. `tfe.implicit_value_and_gradients()`: This is almost identical, but `fg()` also returns the output of the function, `f()`

We will look at the most popular of these, `tf.GradientTape`. Again, within its context, as a calculation takes place, a record (tape) is made of those calculations so that the tape can be replayed with `tf.gradient()` and the appropriate automatic differentiation is be implemented.

In the following code, when the `sum` method is calculated, the tape records the calculations within the `tf.GradientTape()` context so that the automatic differentiation can be found by calling `tape.gradient()`.

Note how lists are used in this example in `[weight1_grad] = tape.gradient(sum, [weight1])`.

By default, only one call to `tape.gradient()` may be made:

```
# by default, you can only call tape.gradient once in a GradientTape
context
weight1 = tf.Variable(2.0)
def weighted_sum(x1):
   return weight1 * x1
with tf.GradientTape() as tape:
   sum = weighted_sum(7.)
   [weight1_grad] = tape.gradient(sum, [weight1])
print(weight1_grad.numpy()) # 7 , weight1*x diff w.r.t. weight1 is x, 7.0,
also see below.
```

In this next example, note that the argument, `persistent=True`, has been passed to `tf.GradientTape()`. This allows us to call `tape.gradient()` more than once. Again, we compute a weighted sum inside the `tf.GradientTape` context and then call `tape.gradient()` to calculate the derivatives of each term with respect to its `weight` variable:

```
# if you need to call tape.gradient() more than once
# use GradientTape(persistent=True)
weight1 = tf.Variable(2.0)
weight2 = tf.Variable(3.0)
weight3 = tf.Variable(5.0)

def weighted_sum(x1, x2, x3):
    return weight1*x1 + weight2*x2 + weight3*x3

with tf.GradientTape(persistent=True) as tape:
   sum = weighted_sum(7.,5.,6.)

[weight1_grad] = tape.gradient(sum, [weight1])
[weight2_grad] = tape.gradient(sum, [weight2])
[weight3_grad] = tape.gradient(sum, [weight3])
```

```
print(weight1_grad.numpy()) #7.0
print(weight2_grad.numpy()) #5.0
print(weight3_grad.numpy()) #6.0
```

Next, we will examine loss functions. These are functions that are optimized during the training of a neural network model.

Loss functions

A `loss` function (that is, an error measurement) is a necessary part of the training of an ANN. It is a measure of the extent to which the calculated output of a network during training differs from its required output. By differentiating the `loss` function, we can find a quantity with which to adjust the weights of the connections between the layers so as to make the calculated output of the ANN more closely match the required output.

The simplest `loss` function is the mean squared error:

$$(1/n)\sum_{n}(y_n - \hat{y}_n)^2$$

,

Here, y is the actual label value, and $\hat{y}$ is the predicted label value.

Of particular note is the categorical cross-entropy `loss` function, which is given by the following equation:

$$-\sum_{n}(y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n))$$

This `loss` function is used when only one class is correct out of all the possible ones and so is used when the `softmax` function is used as the output of the final layer of an ANN.

Note that both of these functions differentiate nicely, as required by backpropagation.

Summary

In this chapter, we looked at a number of technologies that support the creation and use of neural networks.

We covered data presentation to an ANN, layers of an ANN, creating the model, gradient calculations for gradient descent algorithms, loss functions, and saving and restoring models. These topics are important precursors to the concepts and techniques that we will encounter in subsequent chapters when we develop neural network models.

Indeed, in the next chapter, we will start to use TensorFlow in earnest by exploring a number of supervised learning scenarios, including linear regression, logistic regression, and k-nearest neighbors.

2
Section 2: Supervised and Unsupervised Learning in TensorFlow 2.00 Alpha

In this section, we will first see a number of applications of TensorFlow in supervised machine learning, to include linear regression, logistic regression, and clustering. We will then look at unsupervised learning, in particular, at autoencoding, as applied to data compression and denoising.

This section contains the following chapters:

- `Chapter 4`, *Supervised Machine Learning Using TensorFlow 2*
- `Chapter 5`, *Unsupervised Learning Using Tensorflow 2*

4
Supervised Machine Learning Using TensorFlow 2

In this chapter, we will discuss and exemplify the use of TensorFlow 2 for supervised machine learning problems for the following situations: linear regression, logistic regression, and **k-Nearest Neighbors** (**KNN**).

In this chapter, we will look at the following topics:

- Supervised learning
- Linear regression
- Our first linear regression example
- The Boston housing dataset
- Logistic regression (classification)
- **k-Nearest Neighbors** (**KNN**)

Supervised learning

Supervised learning is the machine learning scenario in which one or more data points from a set of data points is/are associated with a label. The model then *learns* to predict the labels for unseen data points. For our purposes, each data point will normally be a tensor and will be associated with a label. Supervised learning problems abound in computer vision; for example, an algorithm is shown many pictures of ripe and unripe tomatoes, together with a categorical label indicating whether or not they are ripe, and when the training has concluded, the model is able to predict the status of tomatoes that weren't in its training set. This could have a very direct application in a physical sorting mechanism for tomatoes; or an algorithm that could learn to predict the gender and age of a new face after it has been shown many examples, together with their genders and ages. Furthermore, it could be beneficial if a model could learn to predict the type of a tree from its image, having been trained on many tree images and their type labels.

Linear regression

A linear regression problem is one where you have to predict the value of one *continuous* variable, given the value of one or more other variables (data points); for example, predicting the selling price of a house, given its floor space. You can plot the known features with their associated labels on a simple linear graph in these examples, as in the familiar x, y scatter plots, and plot a line that best fits the data. This is known as a **line of best fit**. You can then read off the label corresponding to any value of your feature that lies within the x range of the plot.

However, linear regression problems may involve several features in which the terminology **multiple** or **multivariate linear regression** is used. In this case, it is not a line that best fits the data, but a plane (two features) or a hyperplane (more than two features). In the house price example, we could add the number of rooms and the length of the garden to the features. There is a famous dataset, known as the Boston housing dataset, that involves 13 features (for more info, see `https://www.kaggle.com/c/ml210-boston`). The regression problem here is to predict the median value of a home in the Boston suburbs, given these 13 features.

A word on nomenclature: features are also known as predictor variables or independent variables. Labels are also known as response variables or dependent variables.

Our first linear regression example

We will start with a simple, artificial, linear regression problem to set the scene. In this problem, we construct an artificial dataset where we first create, and hence, know, the line to which we are fitting, but then we'll use TensorFlow to find this line.

We do this as follows—after our imports and initialization, we enter a loop. Inside this loop, we calculate the overall loss (defined as the mean squared error over our dataset, y, of points). We then take the derivative of this loss with respect to our weights and bias. This produces values that we can use to adjust our weights and bias to lower the loss; this is known as gradient descent. By repeating this loop a number of times (technically called **epochs**), we can lower our loss to the point where it is as low as it can go, and we can use our trained model to make predictions.

First, we import the required modules (recall that eager execution is the default):

```
import tensorflow as tf
import numpy as np
```

Next, we initialize important constants, as follows:

```
n_examples = 1000 # number of training examples
training_steps = 1000 # number of steps we are going to train for
display_step = 100 # after multiples of this, we display the loss
learning_rate = 0.01 # multiplying factor on gradients
m, c = 6, -5 # gradient and y-intercept of our line, edit these for a
different linear problem
```

A function to calculate our predicted `y`, given `weight` and `bias` (`m` and `c`):

```
def train_data(n, m, c):
    x = tf.random.normal([n]) # n values taken from a normal distribution,
    noise = tf.random.normal([n])# n values taken from a normal
distribution
    y = m*x + c + noise # our scatter plot
    return x, y
def prediction(x, weight, bias):
    return weight*x + bias # our predicted (learned) m and c, expression is
like y = m*x + c
```

A function to take the initial, or predicted, weights and biases and calculate the mean-squared loss (deviation) from `y`:

```
def loss(x, y, weights, biases):
    error = prediction(x, weights, biases) - y # how 'wrong' our predicted
(learned) y is
    squared_error = tf.square(error)
    return tf.reduce_mean(input_tensor=squared_error) # overall mean of
squared error, scalar value.
```

This is where TensorFlow comes into its own. Using a class called `GradientTape()`, we can write a function to calculate the derivatives (gradients) of our loss with respect to our `weights` and `bias`:

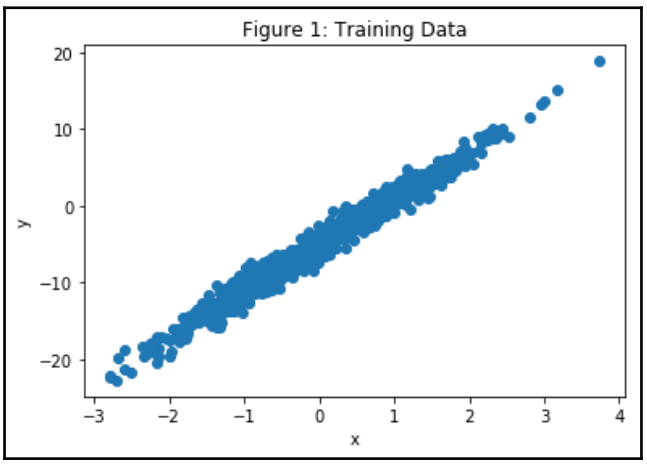
```
def grad(x, y, weights, biases):
    with tf.GradientTape() as tape:
         loss_ = loss(x, y, weights, biases)
    return tape.gradient(loss, [weights, bias]) # direction and value of
the gradient of our weights and biases
```

Set up our regressor for the training loop and display the initial loss as follows:

```
x, y = train_data(n_examples,m,c) # our training values x and y
plt.scatter(x,y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Figure 1: Training Data")
W = tf.Variable(np.random.randn()) # initial, random, value for predicted
weight (m)
B = tf.Variable(np.random.randn()) # initial, random, value for predicted
bias (c)

print("Initial loss: {:.3f}".format(loss(x, y, W, B)))
```

The output is shown as follows:



Next, our main training loop. The idea here is to adjust our `weights` and `bias` by small amounts, factored by our `learning_rate`, to successively lower the loss to the points on which we have converged on our line of best fit:

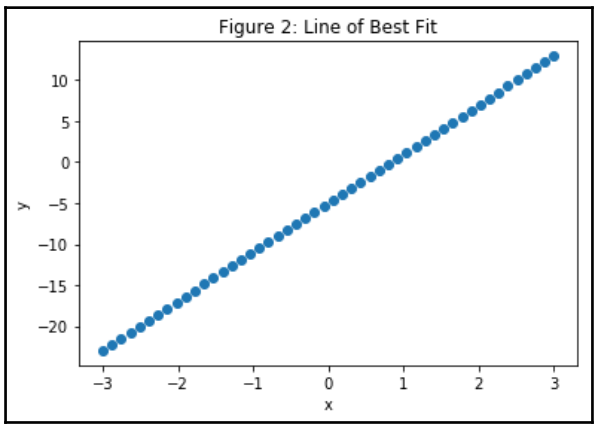
```
for step in range(training_steps): #iterate for each training step
     deltaW, deltaB = grad(x, y, W, B) # direction(sign) and value of the
gradients of our loss
   # with respect to our weights and bias
     change_W = deltaW * learning_rate # adjustment amount for weight
     change_B = deltaB * learning_rate # adjustment amount for bias
     W.assign_sub(change_W) # subract change_W from W
     B.assign_sub(change_B) # subract change_B from B
     if step==0 or step % display_step == 0:
```

```
    # print(deltaW.numpy(), deltaB.numpy()) # uncomment if you want to see
the gradients
  print("Loss at step {:02d}: {:.6f}".format(step, loss(x, y, W, B)))
```

The final results are as follows:

```
print("Final loss: {:.3f}".format(loss(x, y, W, B)))
print("W = {}, B = {}".format(W.numpy(), B.numpy()))
print("Compared with m = {:.3f}, c = {:.3f}".format(m, c)," of the original
line")
xs = np.linspace(-3, 4, 50)
ys = W.numpy()*xs + B.numpy()
plt.scatter(xs,ys)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Figure 2: Line of Best Fit")
```

You should see that the values found for `W` and `B` are very close to the values we used for `m` and `c`, as is to be expected:



The Boston housing dataset

Next, we will apply a similar regression technique to the Boston housing dataset.

The main difference between this and our previous artificial dataset, which had just one feature, is that the Boston housing dataset is real data and has 13 features. This is a regression problem because house prices—the label—we take as being continuously valued.