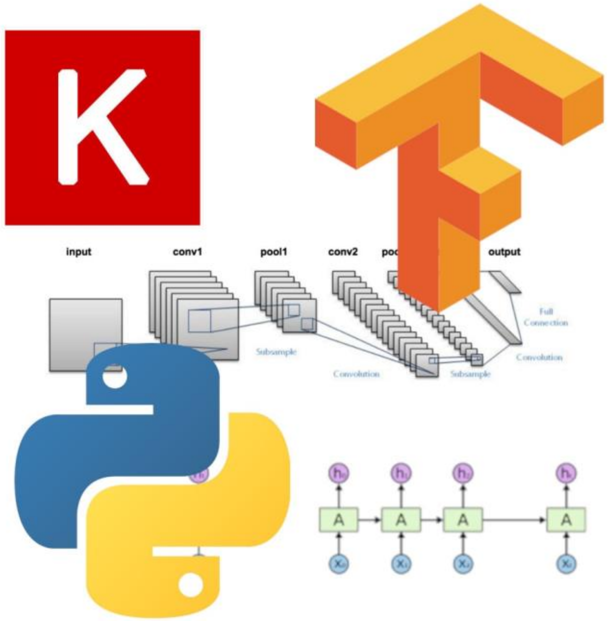
Introduction to TensorFlow in Python

Recommendation system, image classification

Black Raven (James Ng)

21 Feb 2021 · 21 min read



This is a memo to share what I have learnt in Introduction to TensorFlow in Python, capturing the learning objectives as well as my personal notes. The course is taught by Isaiah Hull from DataCamp, and it includes 4 chapters:

Chapter 1: Introduction to TensorFlow
Chapter 2: Linear models
Chapter 3: Neural Networks
Chapter 4: High Level APIs

Not long ago, cutting-edge computer vision algorithms could not differentiate between images of cats and dogs. Today, a skilled data scientist equipped with nothing more than a laptop can classify tens of thousands of objects with greater accuracy than the human eye.

In this course, you will use TensorFlow 2.3 to develop, train, and make predictions with the models that have powered major advances in recommendation systems, image classification, and FinTech. You will learn both high-level APIs, which will enable you to design and train deep learning models in 15 lines of code, and low-level APIs, which will allow you to move beyond off-the-shelf routines. You will also learn to accurately predict housing prices, credit card borrower defaults, and images of sign language gestures.

Chapter 1. Introduction to TensorFlow

Before you can build advanced models in TensorFlow 2, you will first need to understand the basics. In this chapter, you'll learn how to define constants and variables, perform tensor addition and multiplication, and compute derivatives. Knowledge of linear algebra will be helpful, but not necessary.

Constants and variables

TensorFlow is an open-source library for graph-based numerical computation

- developed by the Google Brain Team

Has both low and high level APIs

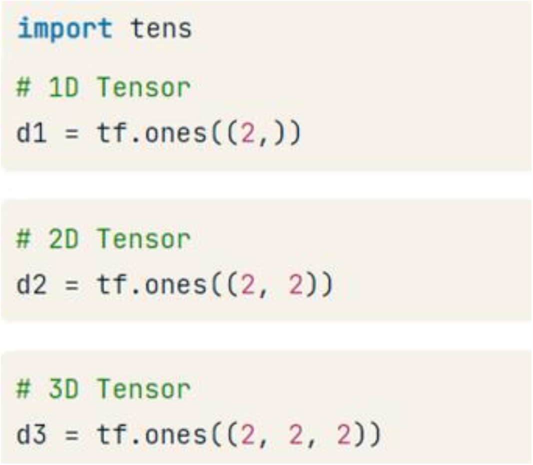
- perform addition, multiplication, and differentiation
- design and train machine learning models

Changes in TenserFlow 2.0

- Eager execution is enabled by default, which allows users to write simpler/intuitive code
- Model building with Keras and Estimators

Tensor

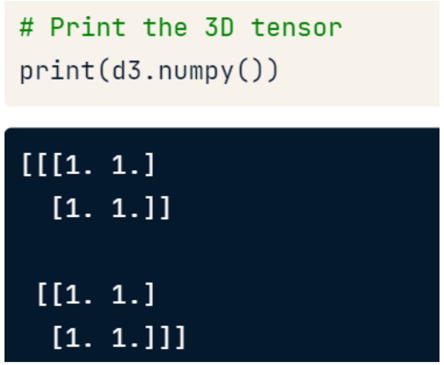
- Generalisation of vectors and matrices to potentially higher dimensions
- Collection of numbers
- Arranged in specific shape: 0-dim, 1-dim, 2-dim, 3-dimensional



```
import tens

# 1D Tensor
d1 = tf.ones((2,))
```

```
# 2D Tensor
d2 = tf.ones((2, 2))
```

```
# 3D Tensor
d3 = tf.ones((2, 2, 2))
```

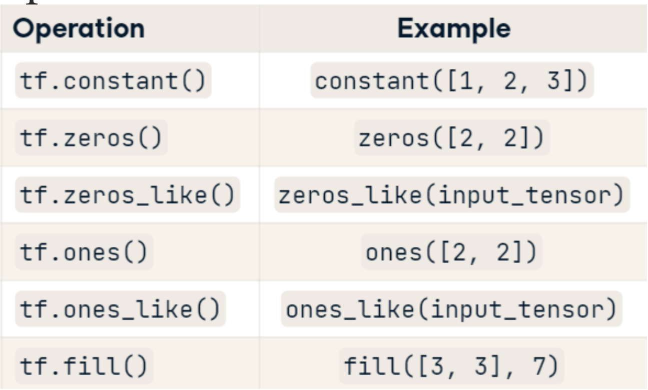


```
# Print the 3D tensor
print(d3.numpy())
```

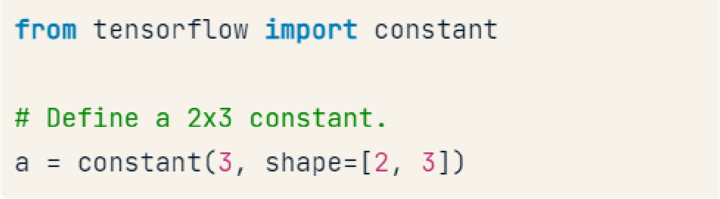
```
[[[1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]]]
```

Constant

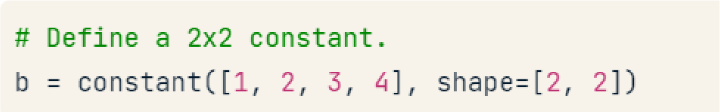
- Simplest category of tensor
- Constant cannot change, cannot be trained
- Can have any dimension
- Special tensors:



Operation	Example
`tf.constant()`	`constant([1, 2, 3])`
`tf.zeros()`	`zeros([2, 2])`
`tf.zeros_like()`	`zeros_like(input_tensor)`
`tf.ones()`	`ones([2, 2])`
`tf.ones_like()`	`ones_like(input_tensor)`
`tf.fill()`	`fill([3, 3], 7)`



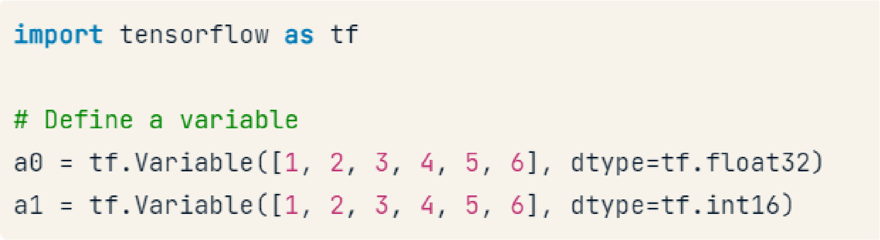
```
from tensorflow import constant

# Define a 2x3 constant.
a = constant(3, shape=[2, 3])
```



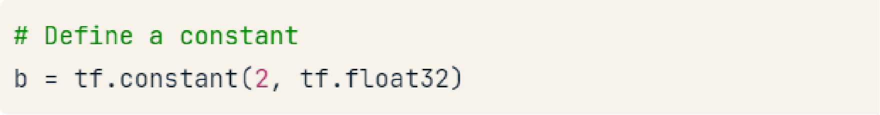
```
# Define a 2x2 constant.
b = constant([1, 2, 3, 4], shape=[2, 2])
```

Variable

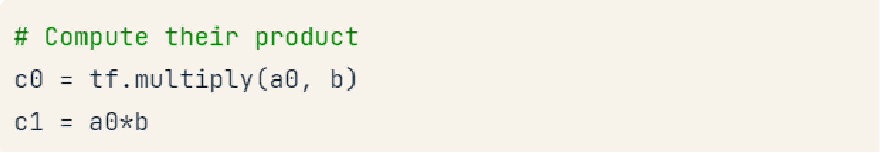
- Value can change during computation
- Value is shared, persistant, modifiable
- Fixed data type and shape



```
import tensorflow as tf

# Define a variable
a0 = tf.Variable([1, 2, 3, 4, 5, 6], dtype=tf.float32)
a1 = tf.Variable([1, 2, 3, 4, 5, 6], dtype=tf.int16)
```



```
# Define a constant
b = tf.constant(2, tf.float32)
```



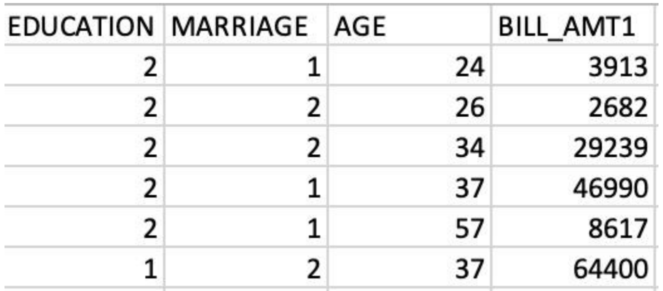
```
# Compute their product
c0 = tf.multiply(a0, b)
c1 = a0*b
```

Defining data as constants

Throughout this course, we will use `tensorflow` version 2.4 and will exclusively import the submodules needed to complete each exercise. This will usually be done for you, but you will do it in this exercise by importing `constant` from `tensorflow`.

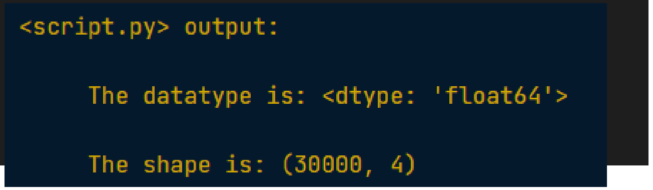
After you have imported `constant`, you will use it to transform a `numpy` array, `credit_numpy`, into a `tensorflow` constant, `credit_constant`. This array contains feature columns from a dataset on credit card holders and is previewed in the image below. We will return to this dataset in later chapters.

Note that `tensorflow` 2 allows you to use data as either a `numpy` array or a `tensorflow` `constant` object. Using a `constant` will ensure that any operations performed with that object are done in `tensorflow`.



EDUCATION	MARRIAGE	AGE	BILL_AMT1
2	1	24	3913
2	2	26	2682
2	2	34	29239
2	1	37	46990
2	1	57	8617
1	2	37	64400

```
# Import constant from TensorFlow
from tensorflow import constant

# Convert the credit_numpy array into a tensorflow constant
credit_constant = constant(credit_numpy)

# Print constant datatype
print('\n The datatype is:', credit_constant.dtype)

# Print constant shape
print('\n The shape is:', credit_constant.shape)
```



```
<script.py> output:

     The datatype is: <dtype: 'float64'>

     The shape is: (30000, 4)
```

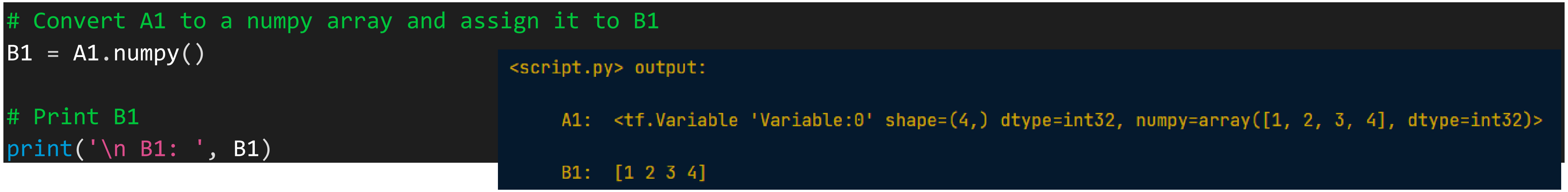
You now understand how constants are used in tensorflow. In the following exercise, you'll practice defining variables.

Defining variables

Unlike a constant, a variable's value can be modified. This will be useful when we want to train a model by updating its parameters.

Let's try defining and printing a variable. We'll then convert the variable to a numpy array, print again, and check for differences. Note that Variable(), which is used to create a variable tensor, has been imported from tensorflow and is available to use in the exercise.

```
# Define the 1-dimensional variable A1
A1 = Variable([1, 2, 3, 4])

# Print the variable A1
print('\n A1: ', A1)
```



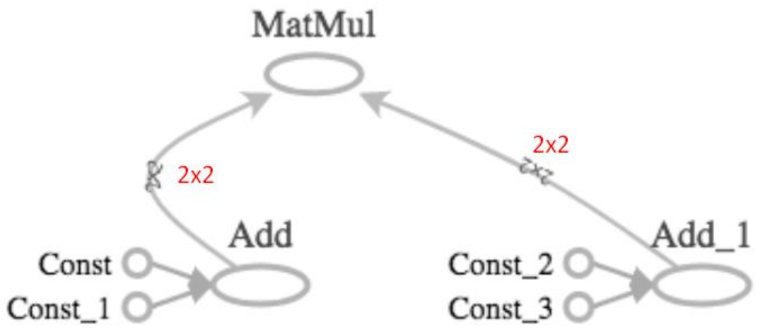
```
# Convert A1 to a numpy array and assign it to B1
B1 = A1.numpy()

# Print B1
print('\n B1: ', B1)
```

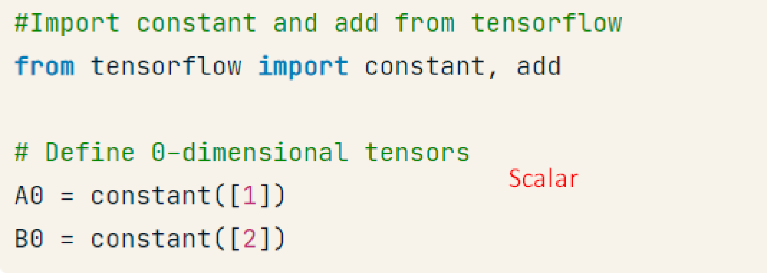
```
<script.py> output:

    A1:  <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>

    B1:  [1 2 3 4]
```

Did you notice any differences between the print statements for `A1` and `B1`? In our next exercise, we'll review how to check the properties of a tensor after it is already defined.

Basic operations

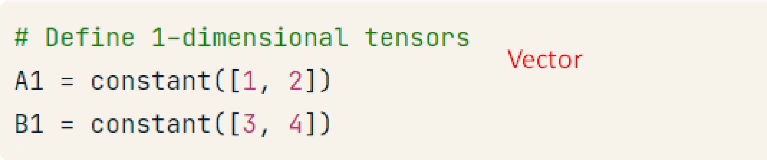
TensorFlow has a model of computation that revolves around the use of graphs.
TensorFlow **graph** contains edges and nodes, where the edges are tensors and the nodes are operations.





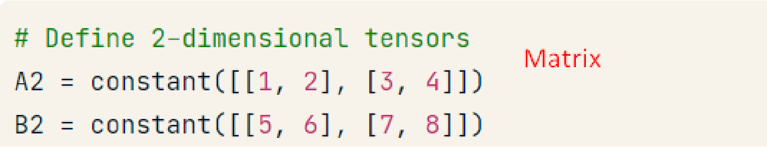
```
#Import constant and add from tensorflow
from tensorflow import constant, add

# Define 0-dimensional tensors
A0 = constant([1])
B0 = constant([2])
```

Scalar



```
# Define 1-dimensional tensors
A1 = constant([1, 2])
B1 = constant([3, 4])
```

Vector

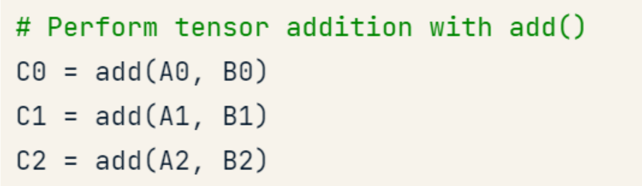


```
# Define 2-dimensional tensors
A2 = constant([[1, 2], [3, 4]])
B2 = constant([[5, 6], [7, 8]])
```

Matrix

Addition operator

- The `add()` operation performs element-wise addition with two tensors
- Each pair of tensors added must have the same shape
- the add operator is overloaded, ie, we can also perform addition using the plus symbol



```
# Perform tensor addition with add()
C0 = add(A0, B0)
C1 = add(A1, B1)
C2 = add(A2, B2)
```

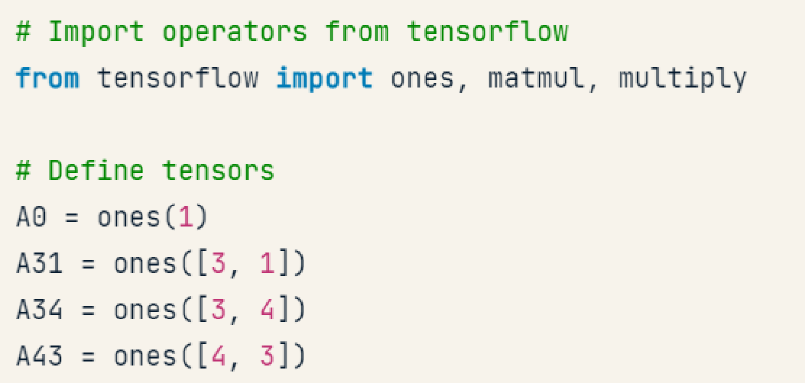
Multiplication operator

Element-wise multiplication performed using multiply() operator

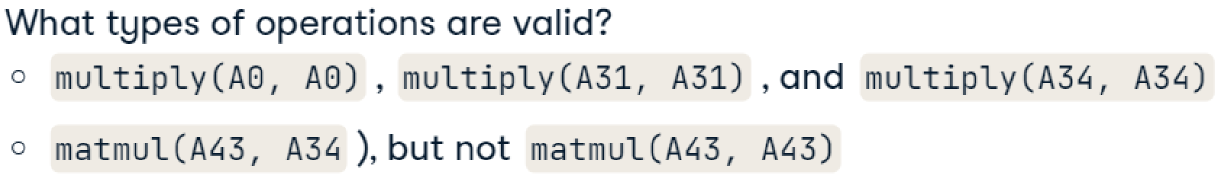
- tensors must have the same shape

Matrix multiplication performed using matmul() operator

- matmul(A,B) operation multiplies A by B
- the number of columns of A must equal the number of rows of B



```python
# Import operators from tensorflow
from tensorflow import ones, matmul, multiply

# Define tensors
A0 = ones(1)
A31 = ones([3, 1])
A34 = ones([3, 4])
A43 = ones([4, 3])
```



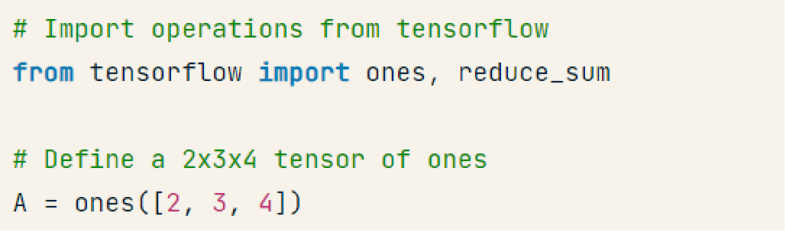
What types of operations are valid?

- ○ `multiply(A0, A0)` , `multiply(A31, A31)` , and `multiply(A34, A34)`
- ○ `matmul(A43, A34`), but not `matmul(A43, A43)`

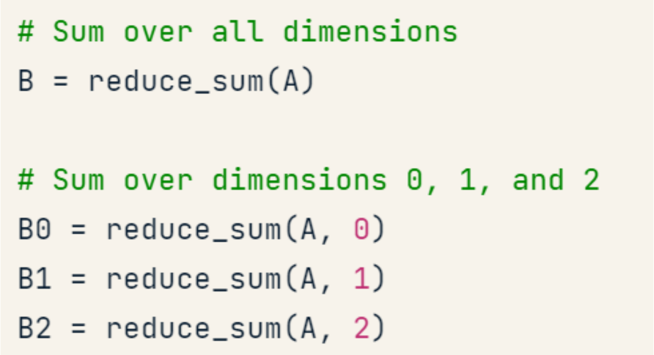
Summing over tensor dimensions

The reduce_sum() operator sums over the dimensions of a tensor

- reduce_sum(A) sums over all dimensions of A
- reduce_sum(A, i) sums over dimension i



```python
# Import operations from tensorflow
from tensorflow import ones, reduce_sum

# Define a 2x3x4 tensor of ones
A = ones([2, 3, 4])
```



```python
# Sum over all dimensions
B = reduce_sum(A)

# Sum over dimensions 0, 1, and 2
B0 = reduce_sum(A, 0)
B1 = reduce_sum(A, 1)
B2 = reduce_sum(A, 2)
```

B = 24

B0 = 3x4 matrix of ‘2’s
B1 = 2x4 matrix of ‘3’s
B2 = 2x3 matrix of ‘4’s

Performing element-wise multiplication

Element-wise multiplication in TensorFlow is performed using two tensors with identical shapes. This is because the operation multiplies elements in corresponding positions in the two tensors. An example of an element-wise multiplication, denoted by the $\odot$ symbol, is shown below:

$$\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix} \odot \begin{bmatrix}3 & 1\\ 2 & 5\end{bmatrix} = \begin{bmatrix}3 & 2\\ 4 & 5\end{bmatrix}$$

In this exercise, you will perform element-wise multiplication, paying careful attention to the shape of the tensors you multiply. Note that `multiply()`, `constant()`, and `ones_like()` have been imported for you.

```
# Define tensors A1 and A23 as constants
A1 = constant([1, 2, 3, 4])
A23 = constant([[1, 2, 3], [1, 6, 4]])

# Define B1 and B23 to have the correct shape
B1 = ones_like(A1)
B23 = ones_like(A23)

# Perform element-wise multiplication
C1 = multiply(A1, B1)
C23 = multiply(A23, B23)

# Print the tensors C1 and C23
print('\n C1: {}'.format(C1.numpy()))
print('\n C23: {}'.format(C23.numpy()))
```

```
<script.py> output:

     C1: [1 2 3 4]

     C23: [[1 2 3]
     [1 6 4]]
```

Notice how performing element-wise multiplication with tensors of ones leaves the original tensors unchanged.

Making predictions with matrix multiplication

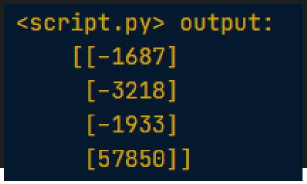
In later chapters, you will learn to train linear regression models. This process will yield a vector of parameters that can be multiplied by the input data to generate predictions. In this exercise, you will use input data, `features`, and a target vector, `bill`, which are taken from a credit card dataset we will use later in the course.

$$features = \begin{bmatrix} 2 & 24 \\ 2 & 26 \\ 2 & 57 \\ 1 & 37 \end{bmatrix}, bill = \begin{bmatrix} 3913 \\ 2682 \\ 8617 \\ 64400 \end{bmatrix}, params = \begin{bmatrix} 1000 \\ 150 \end{bmatrix}$$

The matrix of input data, `features`, contains two columns: education level and age. The target vector, `bill`, is the size of the credit card borrower's bill.

Since we have not trained the model, you will enter a guess for the values of the parameter vector, `params`. You will then use `matmul()` to perform matrix multiplication of `features` by `params` to generate predictions, `billpred`, which you will compare with `bill`. Note that we have imported `matmul()` and `constant()`.

```
# Define features, params, and bill as constants
features = constant([[2, 24], [2, 26], [2, 57], [1, 37]])
params = constant([[1000], [150]])
bill = constant([[3913], [2682], [8617], [64400]])

# Compute billpred using features and params
billpred = matmul(features, params)

# Compute and print the error
error = bill - billpred
print(error.numpy())
```



```
<script.py> output:
    [[-1687]
     [-3218]
     [-1933]
     [57850]]
```

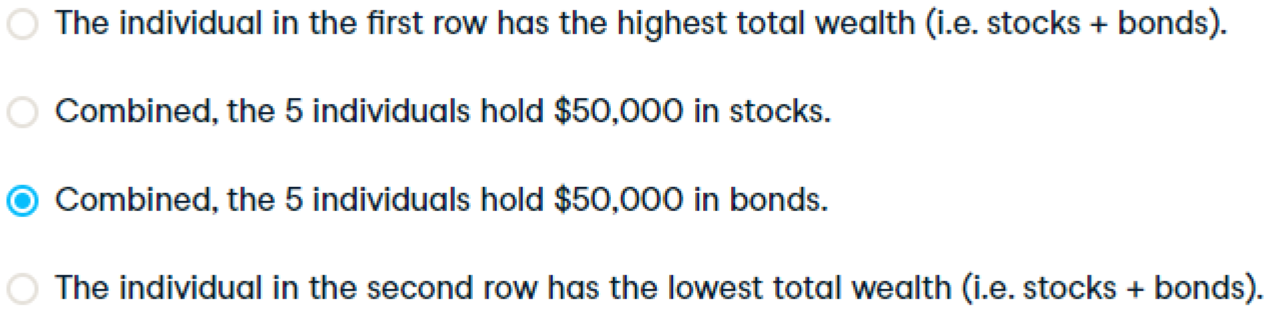
Understanding matrix multiplication will make things simpler when we start making predictions with linear models.

Summing over tensor dimensions

You've been given a matrix, `wealth`. This contains the value of bond and stock wealth for five individuals in thousands of dollars.
The first column corresponds to bonds and the second corresponds to stocks. Each row gives the bond and stock wealth for a single individual. Use `wealth`, `reduce_sum()`, and `.numpy()` to determine which statements are correct about `wealth`.

$$\texttt{wealth} = \begin{bmatrix} 11 & 50 \\ 7 & 2 \\ 4 & 60 \\ 3 & 0 \\ 25 & 10 \end{bmatrix}$$

Possible Answers



○ The individual in the first row has the highest total wealth (i.e. stocks + bonds).

○ Combined, the 5 individuals hold $50,000 in stocks.

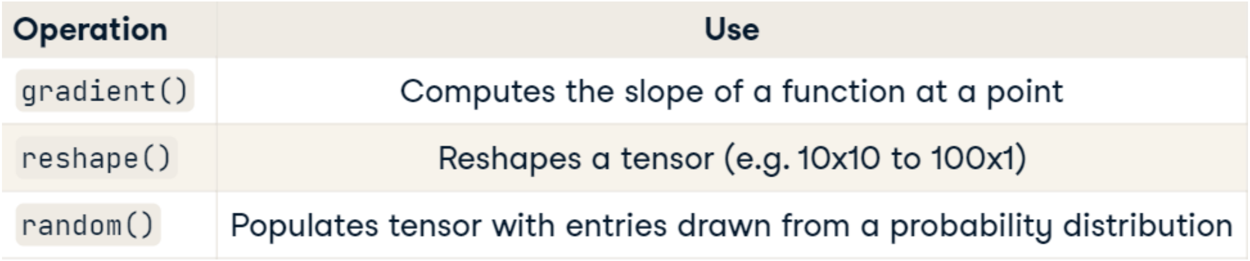
● Combined, the 5 individuals hold $50,000 in bonds.

○ The individual in the second row has the lowest total wealth (i.e. stocks + bonds).

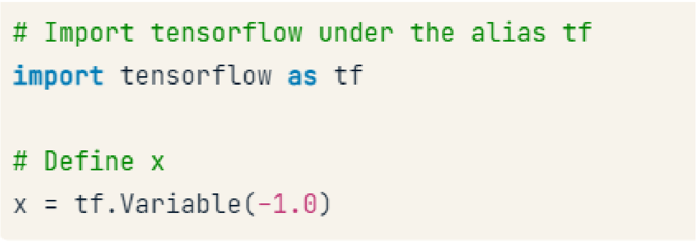
Understanding how to sum over tensor dimensions will be helpful when preparing datasets and training models.

Advanced operations

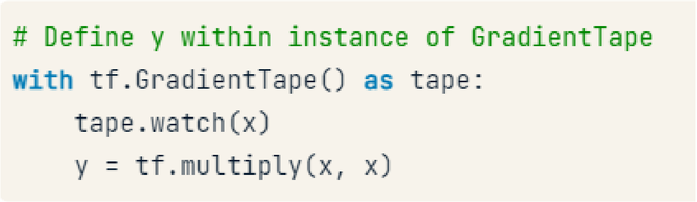
Understanding advanced operations will help you to gain inutition about complex machine learning routines.



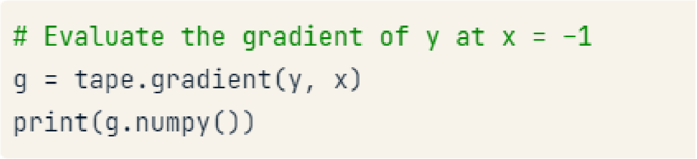
Operation	Use
`gradient()`	Computes the slope of a function at a point
`reshape()`	Reshapes a tensor (e.g. 10x10 to 100x1)
`random()`	Populates tensor with entries drawn from a probability distribution



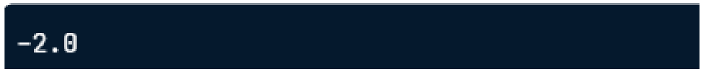
```python
# Import tensorflow under the alias tf
import tensorflow as tf

# Define x
x = tf.Variable(-1.0)
```



```python
# Define y within instance of GradientTape
with tf.GradientTape() as tape:
    tape.watch(x)
    y = tf.multiply(x, x)
```

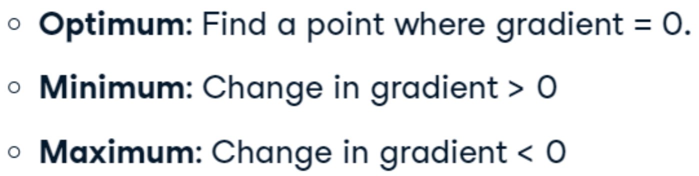


```python
# Evaluate the gradient of y at x = -1
g = tape.gradient(y, x)
print(g.numpy())
```



```
-2.0
```

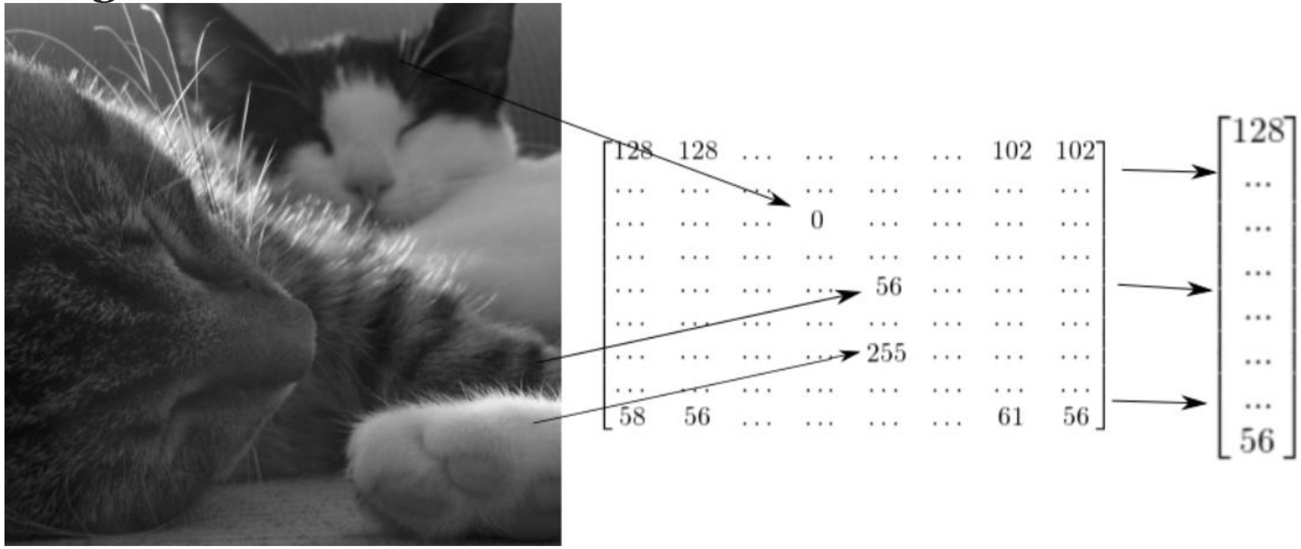
gradient() operation to find



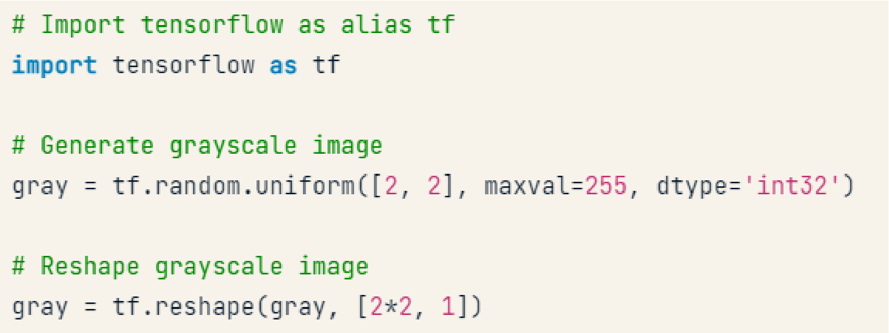
- **Optimum:** Find a point where gradient = 0.
- **Minimum:** Change in gradient > 0
- **Maximum:** Change in gradient < 0

Much of the differentiation you do in deep learning models will be handled by high level APIs. However, gradient tape remains an invaluable tool for building advanced and custom models.

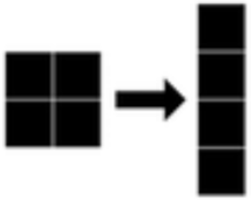
Images as tensors

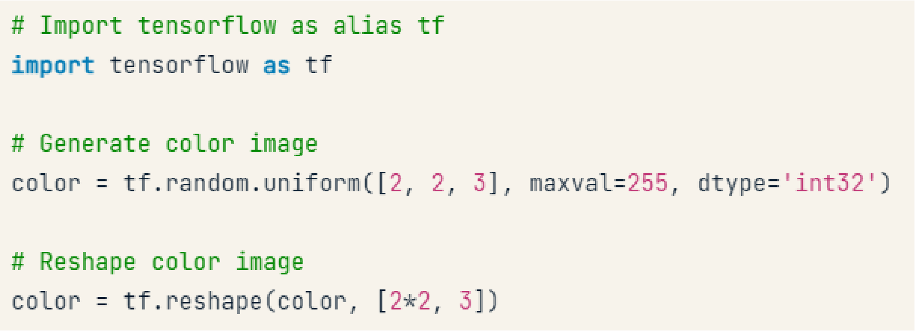


Some algorithms need to **reshape** matrices into vectors before using them as inputs

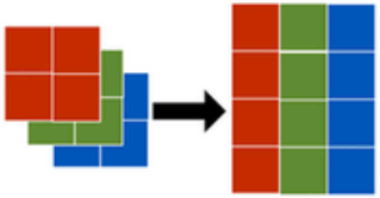


```
# Import tensorflow as alias tf
import tensorflow as tf

# Generate grayscale image
gray = tf.random.uniform([2, 2], maxval=255, dtype='int32')

# Reshape grayscale image
gray = tf.reshape(gray, [2*2, 1])
```





```
# Import tensorflow as alias tf
import tensorflow as tf

# Generate color image
color = tf.random.uniform([2, 2, 3], maxval=255, dtype='int32')

# Reshape color image
color = tf.reshape(color, [2*2, 3])
```



Reshaping tensors

Later in the course, you will classify images of sign language letters using a neural network. In some cases, the network will take 1-dimensional tensors as inputs, but your data will come in the form of images, which will either be either 2- or 3-dimensional tensors, depending on whether they are grayscale or color images.

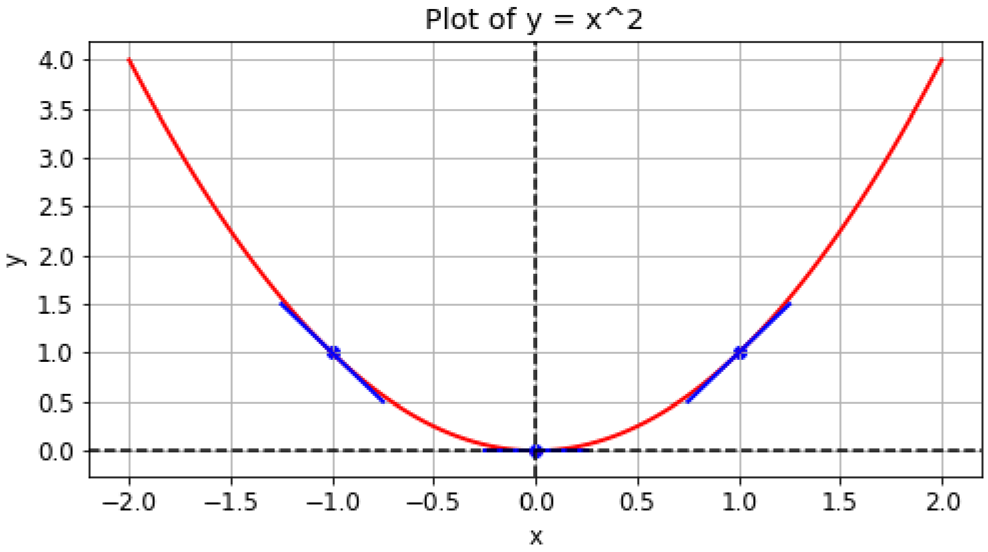
The figure below shows grayscale and color images of the sign language letter A. The two images have been imported for you and converted to the numpy arrays `gray_tensor` and `color_tensor`. Reshape these arrays into 1-dimensional vectors using the `reshape` operation, which has been imported for you from `tensorflow`. Note that the shape of `gray_tensor` is 28x28 and the shape of `color_tensor` is 28x28x3.

```
# Reshape the grayscale image tensor into a vector
gray_vector = reshape(gray_tensor, (784, 1))

# Reshape the color image tensor into a vector
color_vector = reshape(color_tensor, (2352, 1))
```

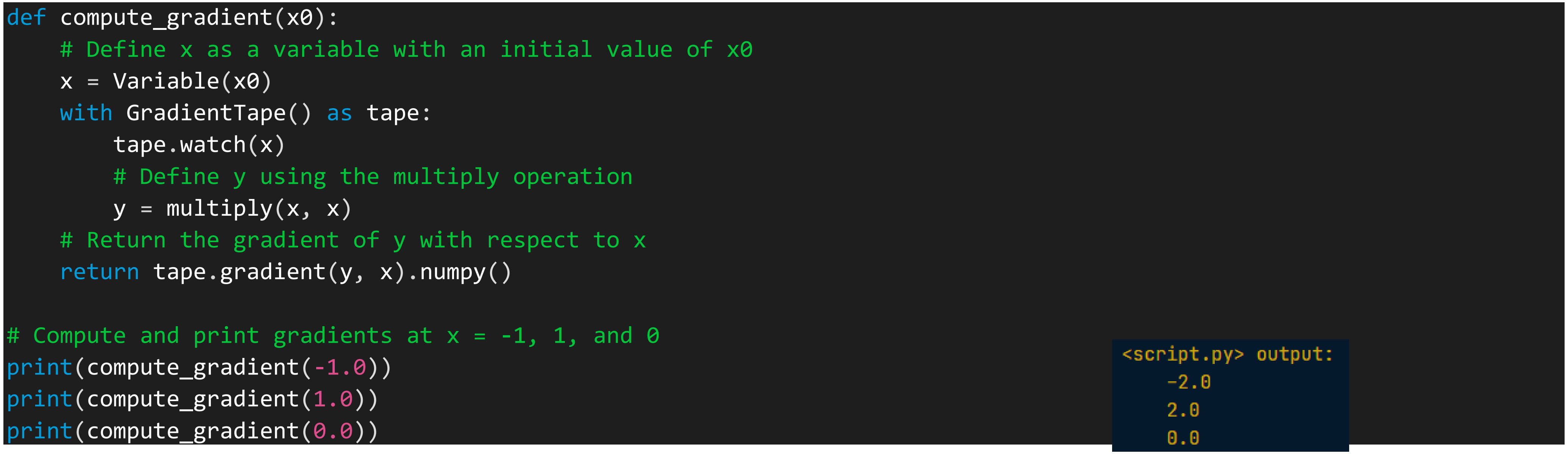
Notice that there are 3 times as many elements in `color_vector` as there are in `gray_vector`, since `color_tensor` has 3 color channels.

Optimizing with gradients

You are given a loss function, y=x2, which you want to minimize. You can do this by computing the slope using the `GradientTape()` operation at different values of `x`. If the slope is positive, you can decrease the loss by lowering `x`. If it is negative, you can decrease it by increasing `x`. This is how gradient descent works.



In practice, you will use a high level `tensorflow` operation to perform gradient descent automatically. In this exercise, however, you will compute the slope at `x` values of -1, 1, and 0. The following operations are available: `GradientTape()`, `multiply()`, and `Variable()`.



```
def compute_gradient(x0):
    # Define x as a variable with an initial value of x0
    x = Variable(x0)
    with GradientTape() as tape:
        tape.watch(x)
        # Define y using the multiply operation
        y = multiply(x, x)
    # Return the gradient of y with respect to x
    return tape.gradient(y, x).numpy()

# Compute and print gradients at x = -1, 1, and 0
print(compute_gradient(-1.0))
print(compute_gradient(1.0))
print(compute_gradient(0.0))
```

```
<script.py> output:
    -2.0
    2.0
    0.0
```

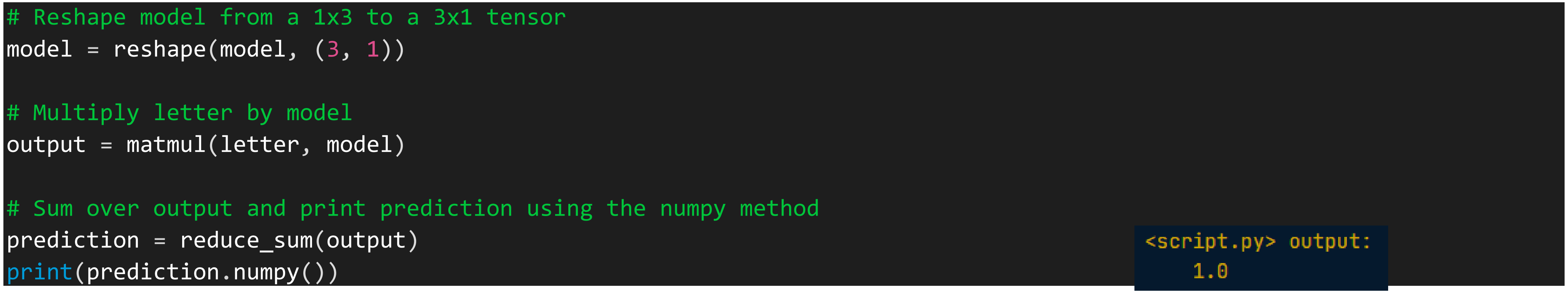
Notice that the slope is positive at `x` = 1, which means that we can lower the loss by reducing `x`. The slope is negative at `x` = -1, which means that we can lower the loss by increasing `x`. The slope at `x` = 0 is 0, which means that we cannot lower the loss by either increasing or decreasing `x`. This is because the loss is minimized at `x` = 0.

Working with image data

You are given a black-and-white image of a letter, which has been encoded as a tensor, `letter`. You want to determine whether the letter is an X or a K. You don't have a trained neural network, but you do have a simple model, `model`, which can be used to classify `letter`.

The 3x3 tensor, `letter`, and the 1x3 tensor, `model`, are available in the Python shell. You can determine whether `letter` is a K by multiplying `letter` by `model`, summing over the result, and then checking if it is equal to 1. As with more complicated models, such as neural networks, `model` is a collection of weights, arranged in a tensor.

Note that the functions `reshape()`, `matmul()`, and `reduce_sum()` have been imported from `tensorflow` and are available for use.



```python
# Reshape model from a 1x3 to a 3x1 tensor
model = reshape(model, (3, 1))

# Multiply letter by model
output = matmul(letter, model)

# Sum over output and print prediction using the numpy method
prediction = reduce_sum(output)
print(prediction.numpy())
```

```
<script.py> output:
    1.0
```

Your model found that `prediction`=1.0 and correctly classified the letter as a K. In the coming chapters, you will use data to train a model, `model`, and then combine this with matrix multiplication, `matmul(letter, model)`, as we have done here, to make predictions about the classes of objects.

Chapter 2. Linear models

In this chapter, you will learn how to build, solve, and make predictions with models in TensorFlow 2. You will focus on a simple class of models – the linear regression model – and will try to predict housing prices. By the end of the chapter, you will know how to load and manipulate data, construct loss functions, perform minimization, make predictions, and reduce resource use with batch training.

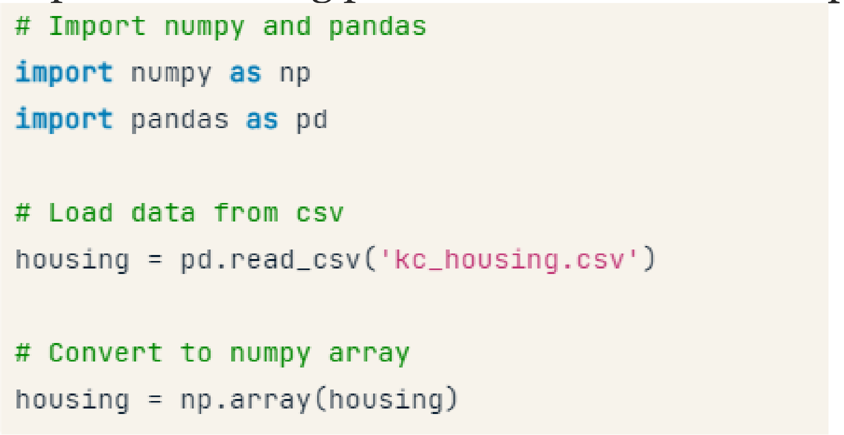
Input data

Training a linear model with TensorFlow

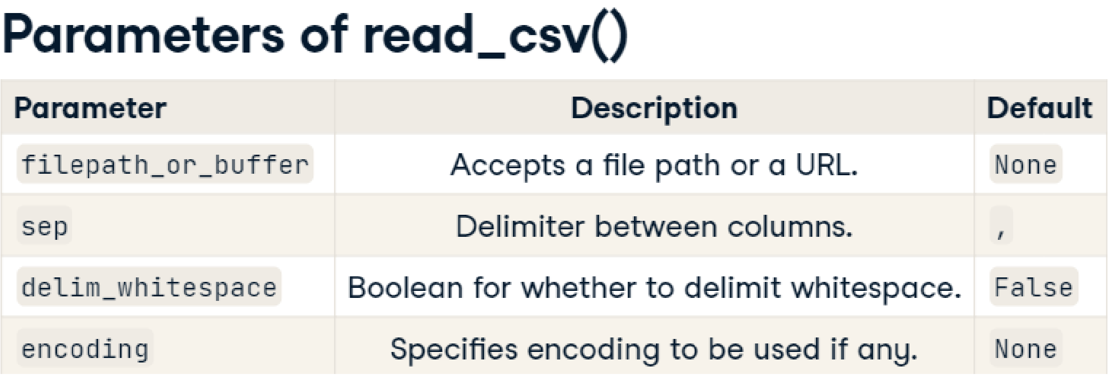
Import data from external source using TensorFlow:

- numeric data -> assign type
- image/text data -> convert to usable format
- useful for managing complex pipelines

Import data using pandas -> convert to numpy array -> use in tensorflow without modification



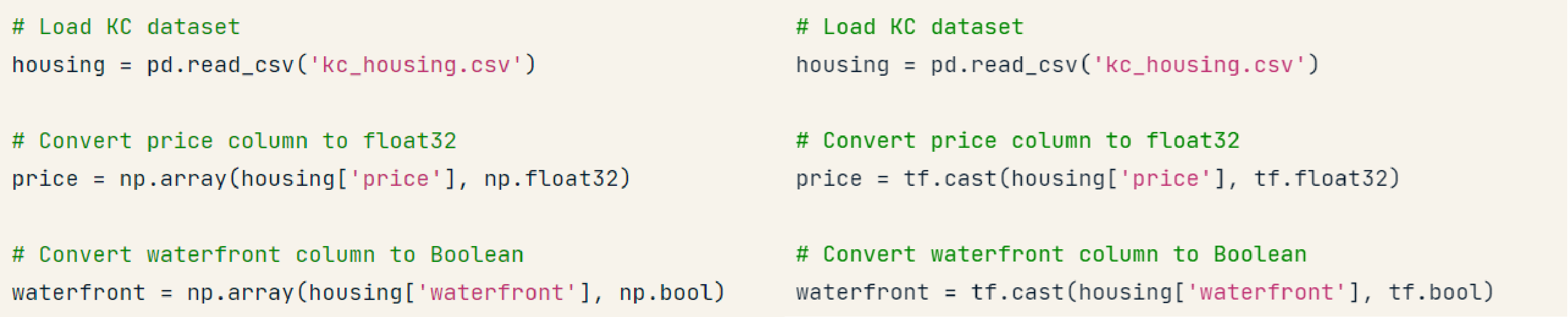
```
# Import numpy and pandas
import numpy as np
import pandas as pd

# Load data from csv
housing = pd.read_csv('kc_housing.csv')

# Convert to numpy array
housing = np.array(housing)
```



Parameters of read_csv()

Parameter	Description	Default
filepath_or_buffer	Accepts a file path or a URL.	None
sep	Delimiter between columns.	,
delim_whitespace	Boolean for whether to delimit whitespace.	False
encoding	Specifies encoding to be used if any.	None

Transform imported data for use in TensorFlow



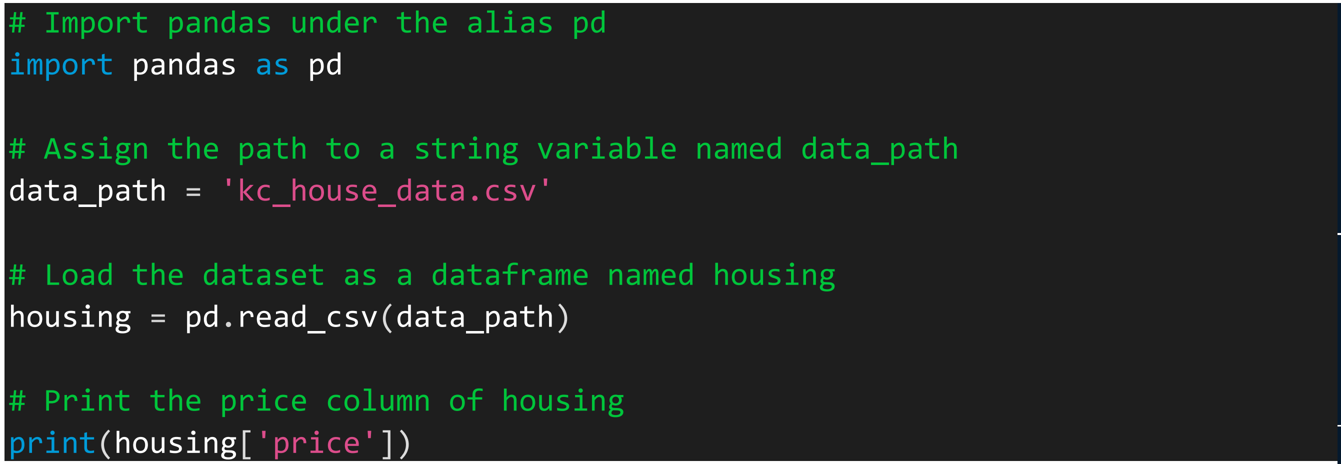
```python
# Load KC dataset
housing = pd.read_csv('kc_housing.csv')

# Convert price column to float32
price = np.array(housing['price'], np.float32)

# Convert waterfront column to Boolean
waterfront = np.array(housing['waterfront'], np.bool)
```

```python
# Load KC dataset
housing = pd.read_csv('kc_housing.csv')

# Convert price column to float32
price = tf.cast(housing['price'], tf.float32)

# Convert waterfront column to Boolean
waterfront = tf.cast(housing['waterfront'], tf.bool)
```

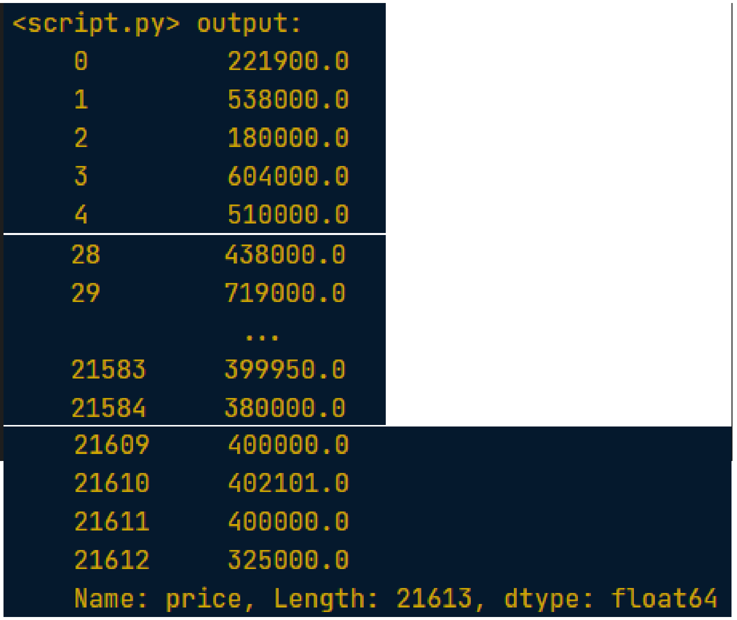
Load data using pandas

Before you can train a machine learning model, you must first import data. There are several valid ways to do this, but for now, we will use a simple one-liner from `pandas`: `pd.read_csv()`. Recall from the video that the first argument specifies the path or URL. All other arguments are optional.

In this exercise, you will import the King County housing dataset, which we will use to train a linear model later in the chapter.



```python
# Import pandas under the alias pd
import pandas as pd

# Assign the path to a string variable named data_path
data_path = 'kc_house_data.csv'

# Load the dataset as a dataframe named housing
housing = pd.read_csv(data_path)

# Print the price column of housing
print(housing['price'])
```

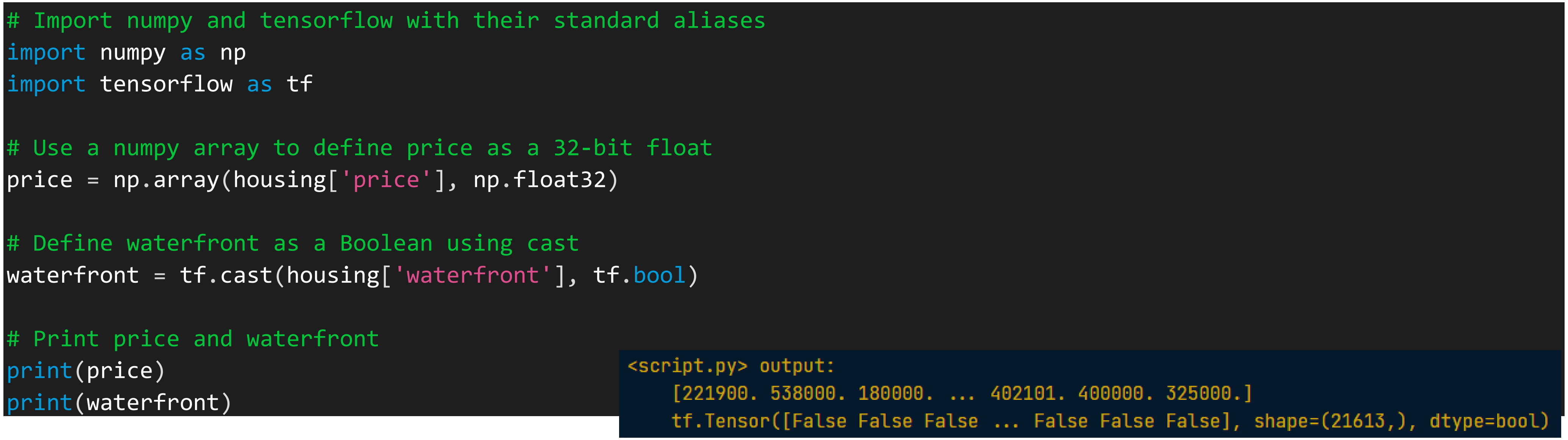


```
<script.py> output:
    0          221900.0
    1          538000.0
    2          180000.0
    3          604000.0
    4          510000.0
    28         438000.0
    29         719000.0
                 ...
    21583      399950.0
    21584      380000.0
    21609      400000.0
    21610      402101.0
    21611      400000.0
    21612      325000.0
    Name: price, Length: 21613, dtype: float64
```

Notice that you did not have to specify a delimiter with the `sep` parameter, since the dataset was stored in the default comma-separated format.

Setting the data type

In this exercise, you will both load data and set its type. Note that `housing` is available and `pandas` has been imported as `pd`. You will import `numpy` and `tensorflow`, and define tensors that are usable in `tensorflow` using columns in `housing` with a given data type. Recall that you can select the `price` column, for instance, from `housing` using `housing['price']`.



```
# Import numpy and tensorflow with their standard aliases
import numpy as np
import tensorflow as tf

# Use a numpy array to define price as a 32-bit float
price = np.array(housing['price'], np.float32)

# Define waterfront as a Boolean using cast
waterfront = tf.cast(housing['waterfront'], tf.bool)

# Print price and waterfront
print(price)
print(waterfront)
```

```
<script.py> output:
    [221900. 538000. 180000. ... 402101. 400000. 325000.]
    tf.Tensor([False False False ... False False False], shape=(21613,), dtype=bool)
```

Notice that printing `price` yielded a `numpy` array; whereas printing `waterfront` yielded a `tf.Tensor()`.

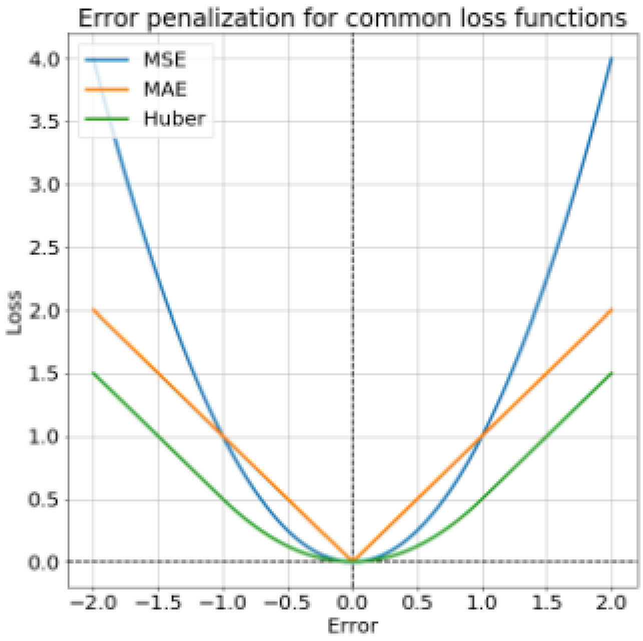
Loss functions

Fundamental TensorFlow operation

- used to train a model
- measure of model fit, ie, how well the model explains the data
- minimise the loss function: high value -> worse fit

TensorFlow has operations for common loss functions

- mean squared error (MSE) using `tf.keras.losses.mse()`
- mean absolute error (MAE) using `tf.keras.losses.mae()`
- Huber error using `tf.keras.losses.Huber()`



MSE

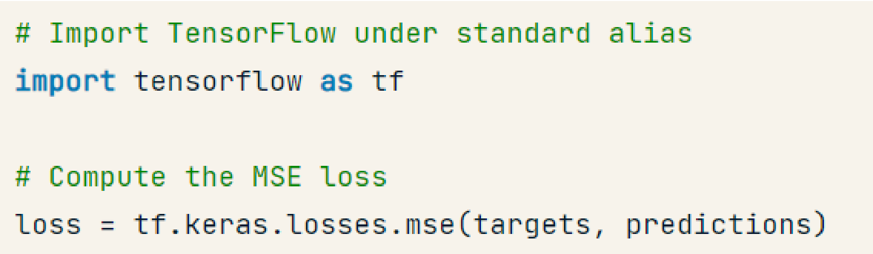
- Strongly penalises outliers
- High (gradient) sensitivity near minimum

MAE

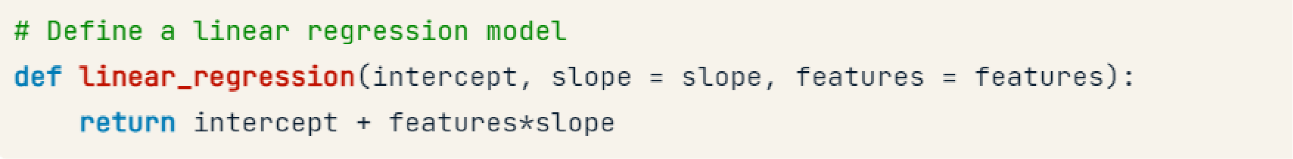
- Scales linearly with size of error
- Low sensitivity near minimum

Huber

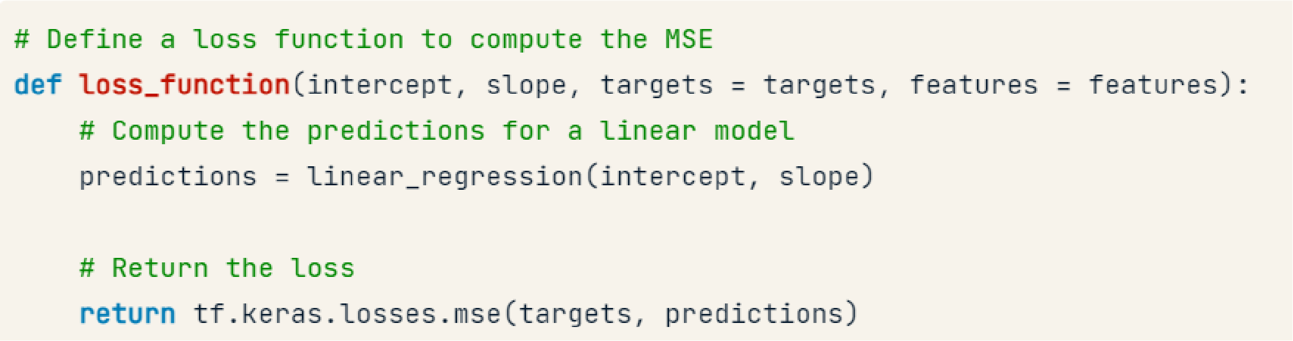
- Similar to MSE near minimum
- Similar to MAE away from minimum



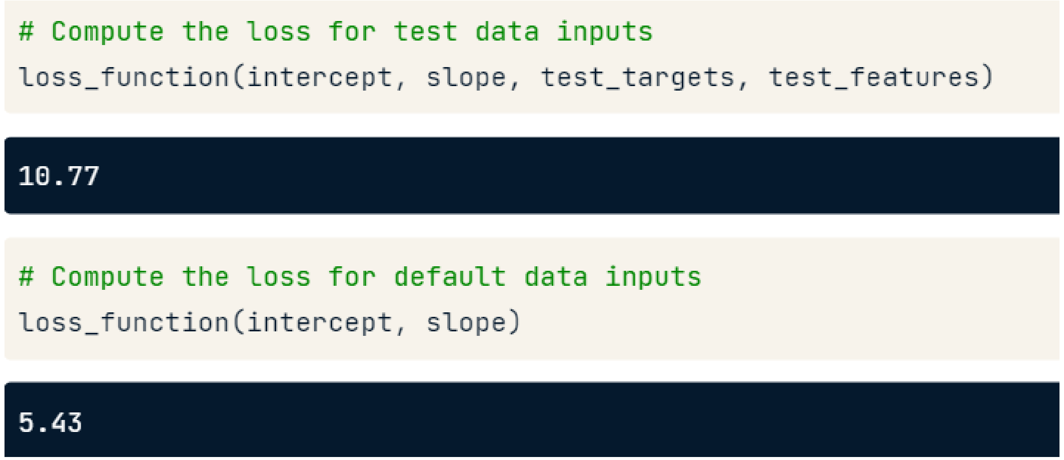
```python
# Import TensorFlow under standard alias
import tensorflow as tf

# Compute the MSE loss
loss = tf.keras.losses.mse(targets, predictions)
```



```python
# Define a linear regression model
def linear_regression(intercept, slope = slope, features = features):
    return intercept + features*slope
```



```python
# Define a loss function to compute the MSE
def loss_function(intercept, slope, targets = targets, features = features):
    # Compute the predictions for a linear model
    predictions = linear_regression(intercept, slope)

    # Return the loss
    return tf.keras.losses.mse(targets, predictions)
```



```
# Compute the loss for test data inputs
loss_function(intercept, slope, test_targets, test_features)
```

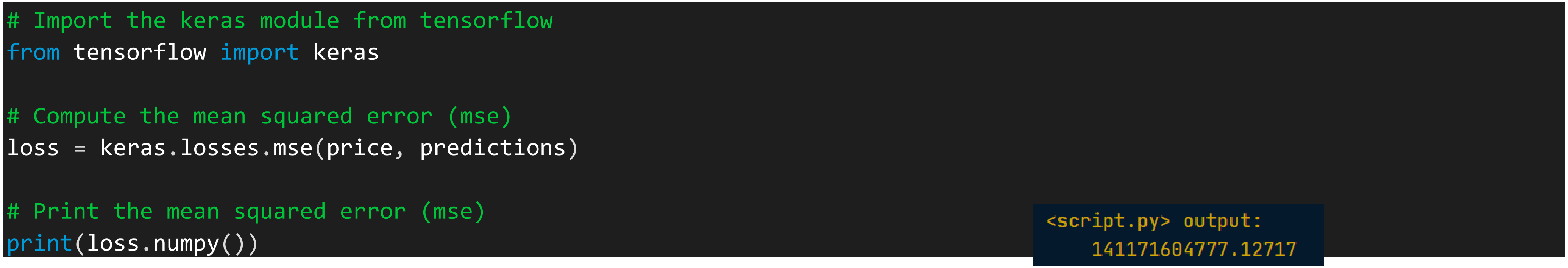
```
10.77
```

```
# Compute the loss for default data inputs
loss_function(intercept, slope)
```

```
5.43
```

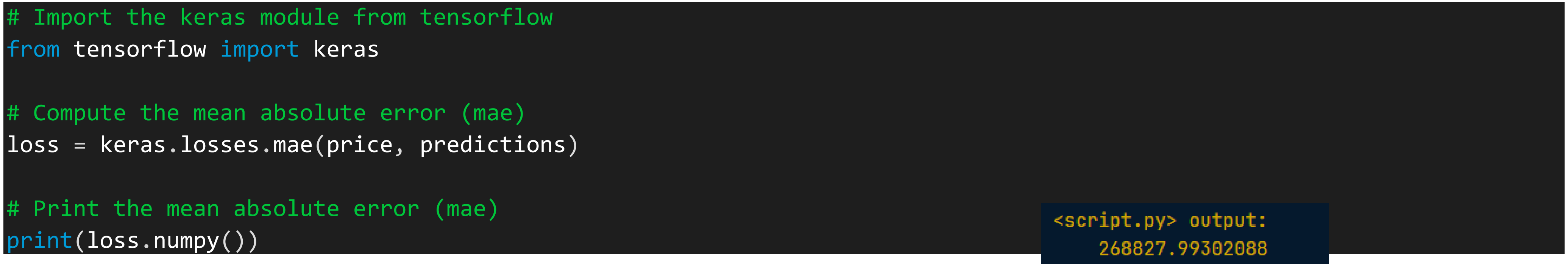
Loss functions in TensorFlow

In this exercise, you will compute the loss using data from the King County housing dataset. You are given a target, `price`, which is a tensor of house prices, and `predictions`, which is a tensor of predicted house prices. You will evaluate the loss function and print out the value of the loss.



```
# Import the keras module from tensorflow
from tensorflow import keras

# Compute the mean squared error (mse)
loss = keras.losses.mse(price, predictions)

# Print the mean squared error (mse)
print(loss.numpy())
```

```
<script.py> output:
    141171604777.12717
```



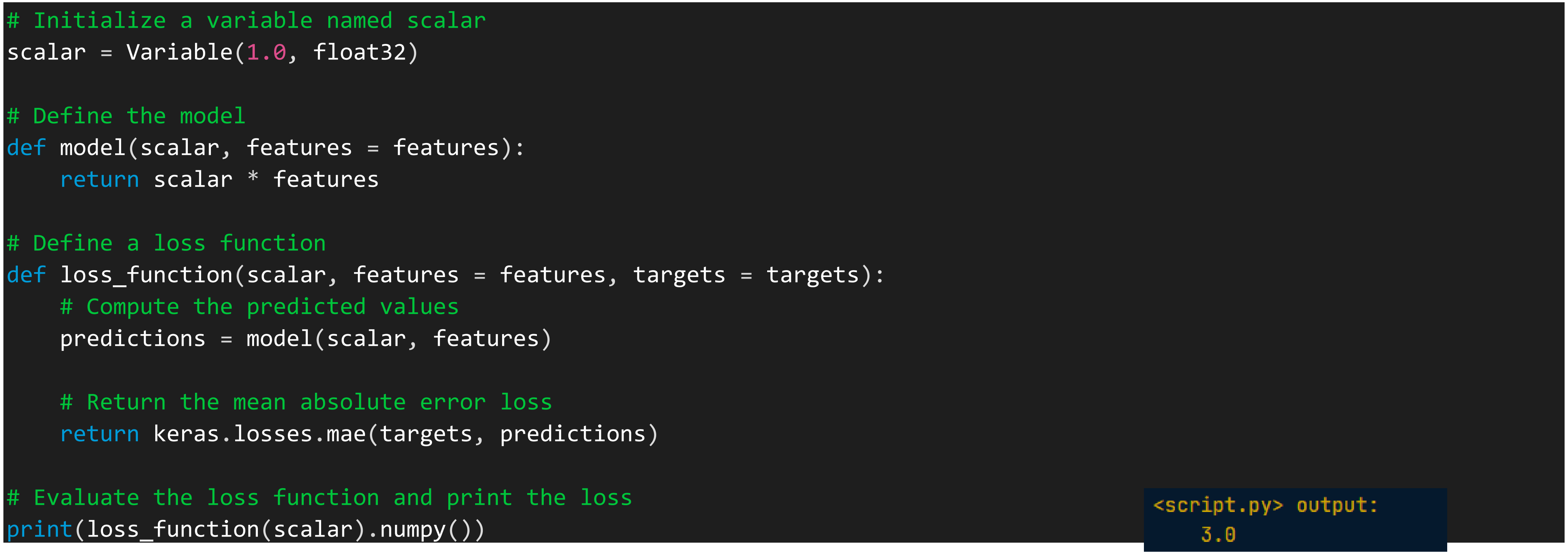
```
# Import the keras module from tensorflow
from tensorflow import keras

# Compute the mean absolute error (mae)
loss = keras.losses.mae(price, predictions)

# Print the mean absolute error (mae)
print(loss.numpy())
```

```
<script.py> output:
    268827.99302088
```

You may have noticed that the MAE was much smaller than the MSE, even though `price` and `predictions` were the same. This is because the different loss functions penalize deviations of `predictions` from `price` differently. MSE does not like large deviations and punishes them harshly.

Modifying the loss function

In the previous exercise, you defined a `tensorflow` loss function and then evaluated it once for a set of actual and predicted values. In this exercise, you will compute the loss within another function called `loss_function()`, which first generates predicted values from the data and variables. The purpose of this is to construct a function of the trainable model variables that returns the loss. You can then repeatedly evaluate this function for different variable values until you find the minimum. In practice, you will pass this function to an optimizer in `tensorflow`. Note that `features` and `targets` have been defined and are available. Additionally, `Variable`, `float32`, and `keras` are available.



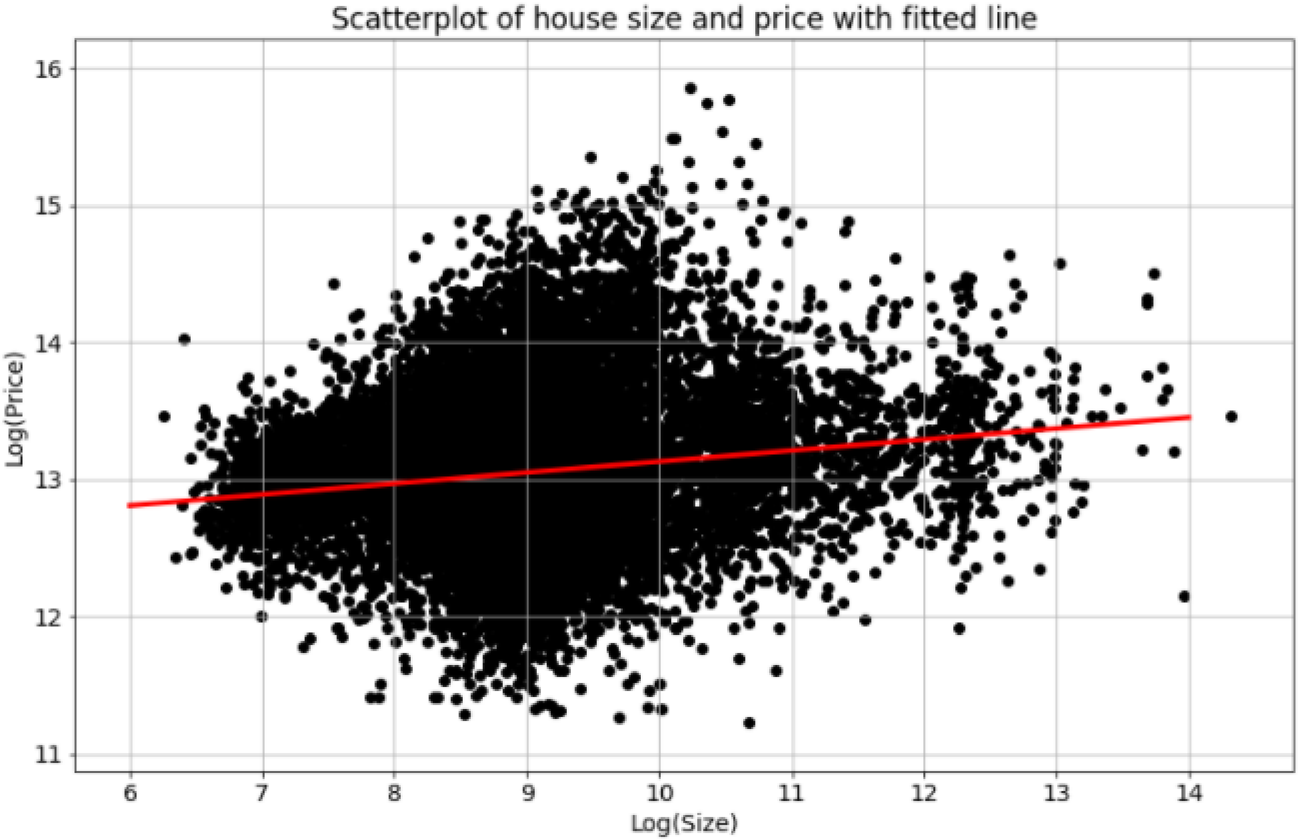
```
# Initialize a variable named scalar
scalar = Variable(1.0, float32)

# Define the model
def model(scalar, features = features):
    return scalar * features

# Define a loss function
def loss_function(scalar, features = features, targets = targets):
    # Compute the predicted values
    predictions = model(scalar, features)

    # Return the mean absolute error loss
    return keras.losses.mae(targets, predictions)

# Evaluate the loss function and print the loss
print(loss_function(scalar).numpy())
```

```
<script.py> output:
    3.0
```

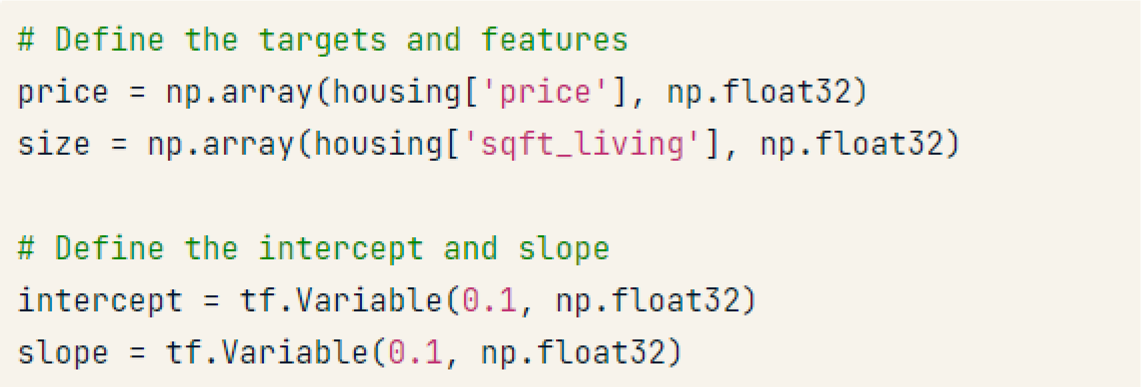
As you will see in the following lessons, this exercise was the equivalent of evaluating the loss function for a linear regression where the intercept is 0.

Linear regression

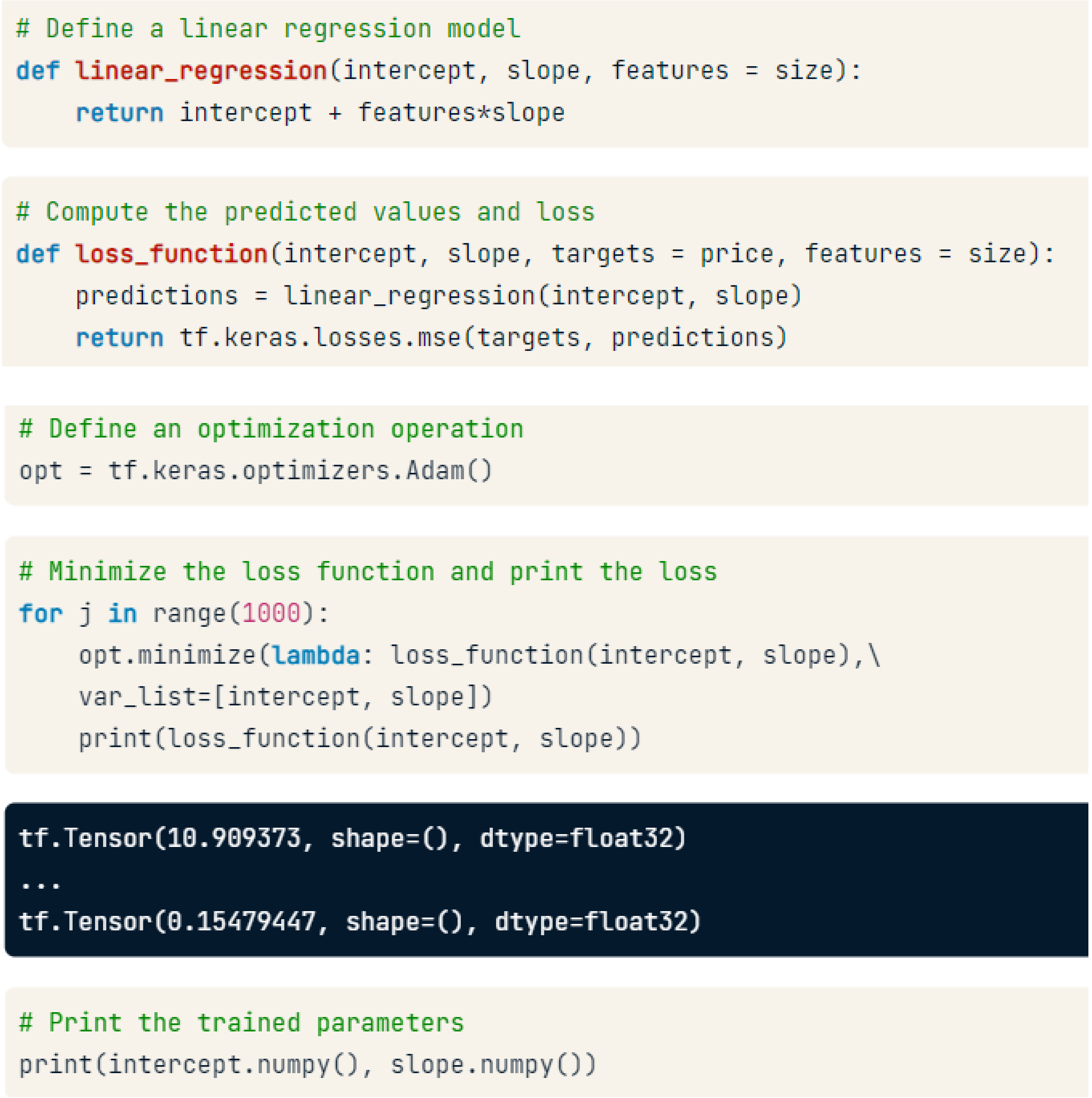
To examine the relationship between house size and price: plot LOG(Price) vs LOG(Size)



A linear regression model assumes a linear relationship
Price = intercept + Size * slope + error



```
# Define the targets and features
price = np.array(housing['price'], np.float32)
size = np.array(housing['sqft_living'], np.float32)

# Define the intercept and slope
intercept = tf.Variable(0.1, np.float32)
slope = tf.Variable(0.1, np.float32)
```



```
# Define a linear regression model
def linear_regression(intercept, slope, features = size):
    return intercept + features*slope
```

```
# Compute the predicted values and loss
def loss_function(intercept, slope, targets = price, features = size):
    predictions = linear_regression(intercept, slope)
    return tf.keras.losses.mse(targets, predictions)
```

```
# Define an optimization operation
opt = tf.keras.optimizers.Adam()
```

```
# Minimize the loss function and print the loss
for j in range(1000):
    opt.minimize(lambda: loss_function(intercept, slope),\
    var_list=[intercept, slope])
    print(loss_function(intercept, slope))
```

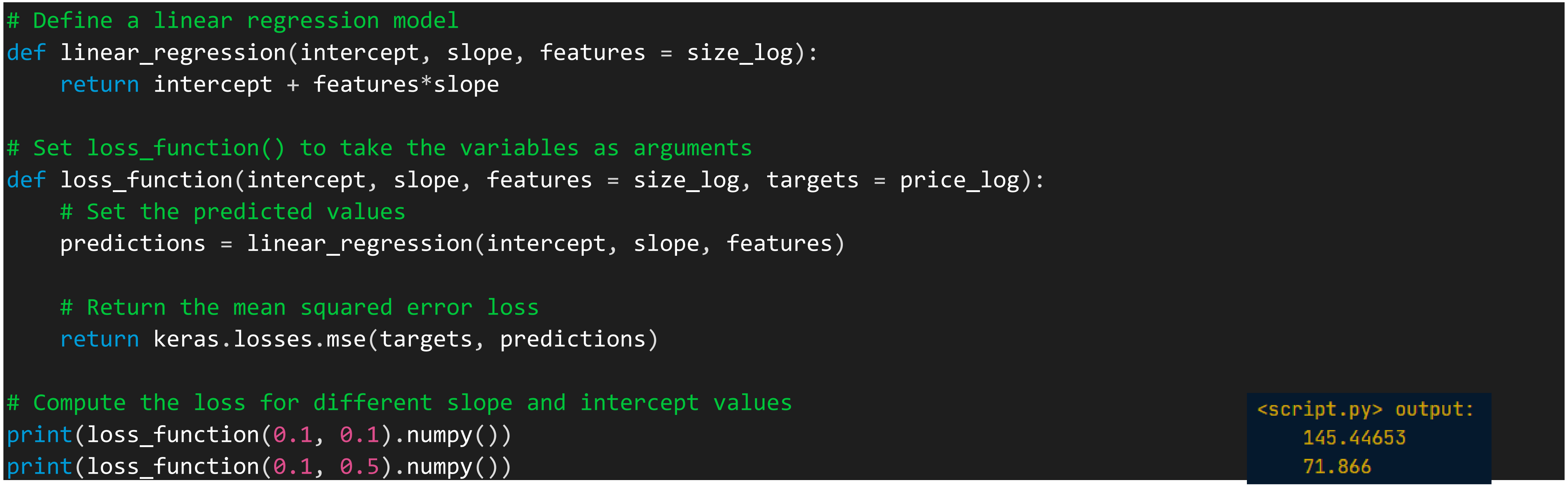
```
tf.Tensor(10.909373, shape=(), dtype=float32)
...
tf.Tensor(0.15479447, shape=(), dtype=float32)
```

```
# Print the trained parameters
print(intercept.numpy(), slope.numpy())
```

Set up a linear regression

A univariate linear regression identifies the relationship between a single feature and the target tensor. In this exercise, we will use a property's lot size and price. Just as we discussed in the video, we will take the natural logarithms of both tensors, which are available as `price_log` and `size_log`.

In this exercise, you will define the model and the loss function. You will then evaluate the loss function for two different values of `intercept` and `slope`. Remember that the predicted values are given by `intercept + features*slope`. Additionally, note that `keras.losses.mse()` is available for you. Furthermore, `slope` and `intercept` have been defined as variables.



```python
# Define a linear regression model
def linear_regression(intercept, slope, features = size_log):
    return intercept + features*slope

# Set loss_function() to take the variables as arguments
def loss_function(intercept, slope, features = size_log, targets = price_log):
    # Set the predicted values
    predictions = linear_regression(intercept, slope, features)

    # Return the mean squared error loss
    return keras.losses.mse(targets, predictions)

# Compute the loss for different slope and intercept values
print(loss_function(0.1, 0.1).numpy())
print(loss_function(0.1, 0.5).numpy())
```

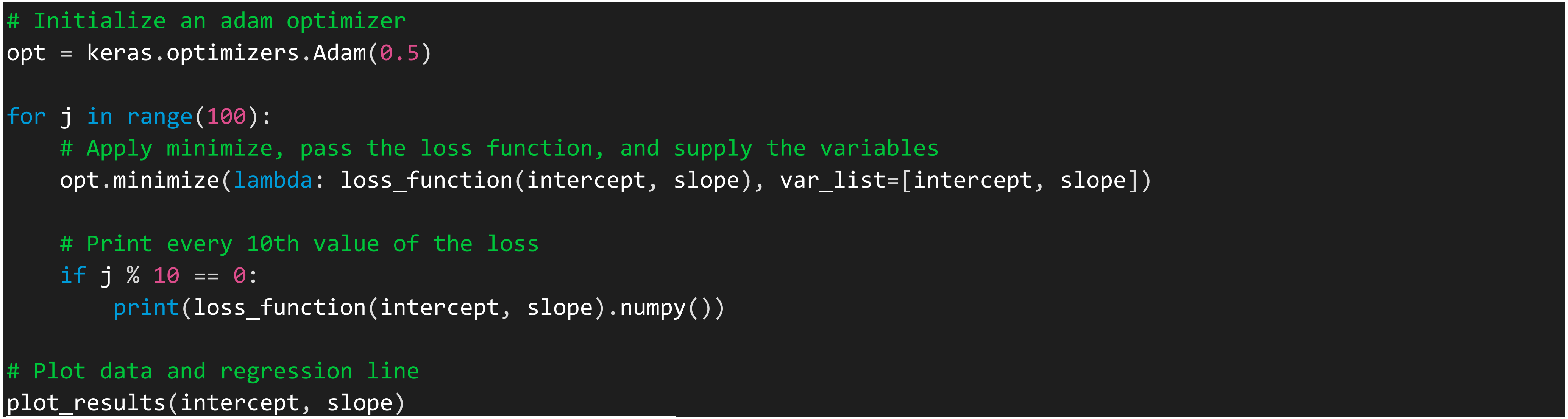
```
<script.py> output:
    145.44653
    71.866
```

In the next exercise, you will actually run the regression and train `intercept` and `slope`.

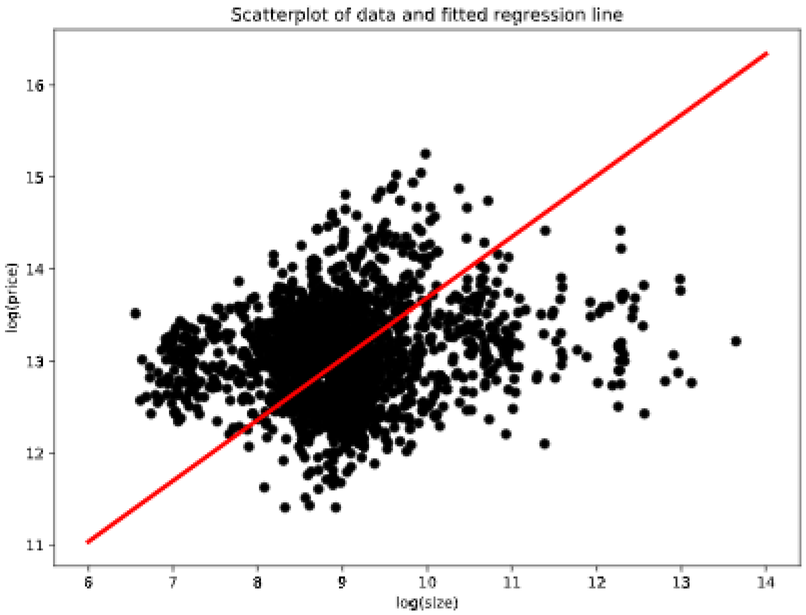
Train a linear model

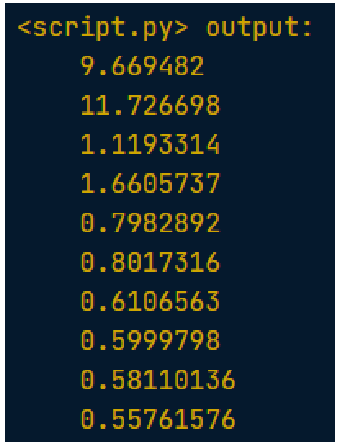
In this exercise, we will pick up where the previous exercise ended. The intercept and slope, `intercept` and `slope`, have been defined and initialized. Additionally, a function has been defined, `loss_function(intercept, slope)`, which computes the loss using the data and model variables.

You will now define an optimization operation as `opt`. You will then train a univariate linear model by minimizing the loss to find the optimal values of `intercept` and `slope`. Note that the `opt` operation will try to move closer to the optimum with each step, but will require many steps to find it. Thus, you must repeatedly execute the operation.



```python
# Initialize an adam optimizer
opt = keras.optimizers.Adam(0.5)

for j in range(100):
    # Apply minimize, pass the loss function, and supply the variables
    opt.minimize(lambda: loss_function(intercept, slope), var_list=[intercept, slope])

    # Print every 10th value of the loss
    if j % 10 == 0:
        print(loss_function(intercept, slope).numpy())

# Plot data and regression line
plot_results(intercept, slope)
```





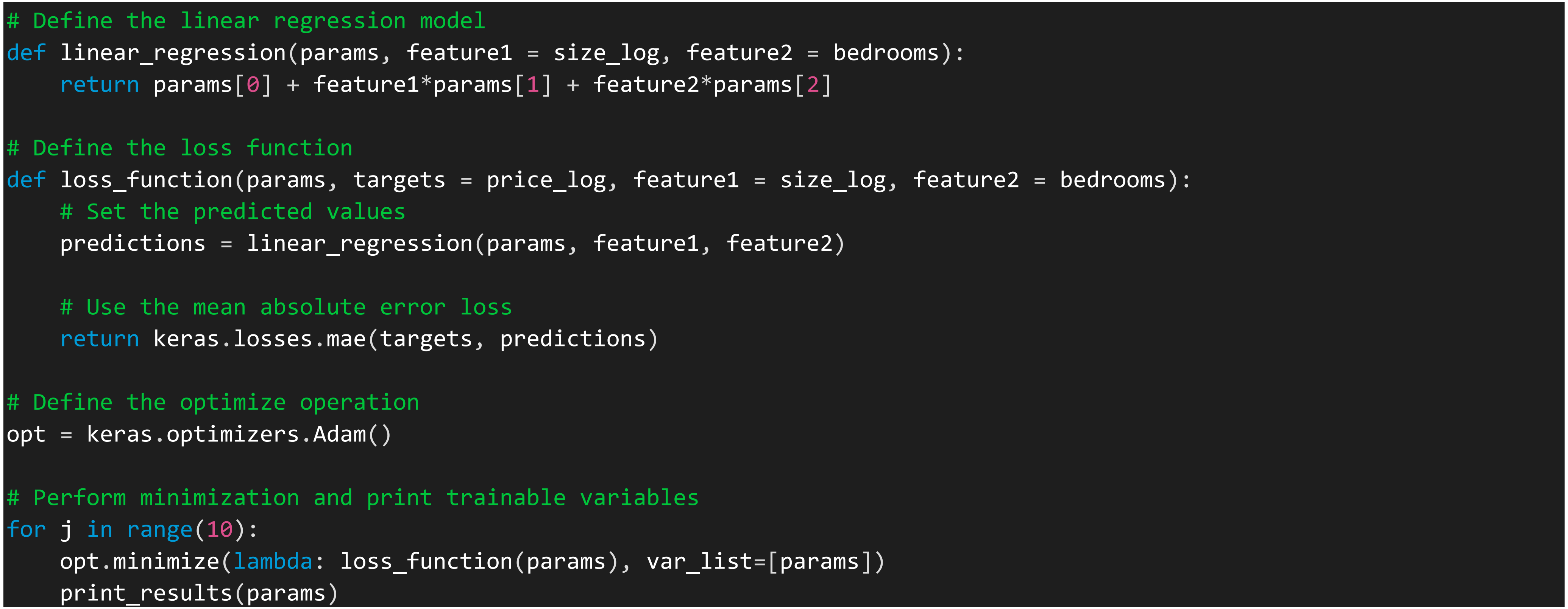
```
<script.py> output:
    9.669482
    11.726698
    1.1193314
    1.6605737
    0.7982892
    0.8017316
    0.6106563
    0.5999798
    0.58110136
    0.55761576
```

Notice that we printed `loss_function(intercept, slope)` every 10th execution for 100 executions. Each time, the loss got closer to the minimum as the optimizer moved the `slope` and `intercept` parameters closer to their optimal values.

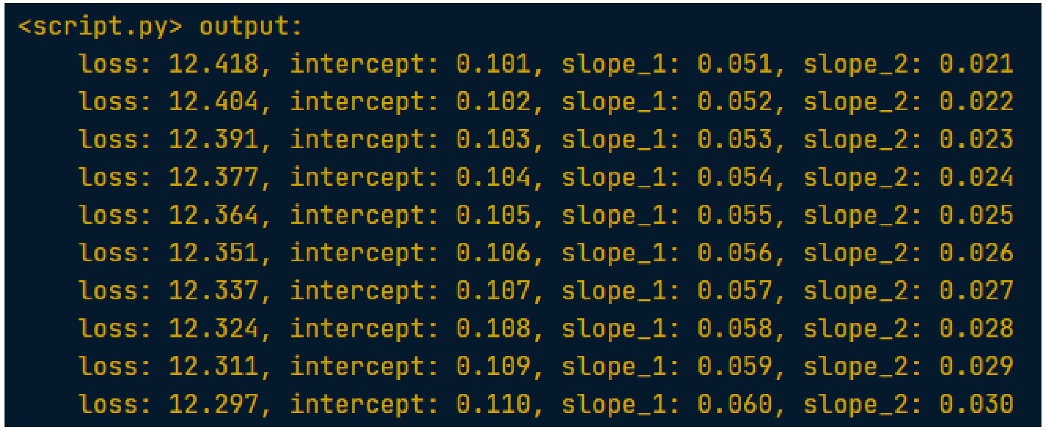
Multiple linear regression

In most cases, performing a univariate linear regression will not yield a model that is useful for making accurate predictions. In this exercise, you will perform a multiple regression, which uses more than one feature.

You will use `price_log` as your target and `size_log` and `bedrooms` as your features. Each of these tensors has been defined and is available. You will also switch from using the the mean squared error loss to the mean absolute error loss: `keras.losses.mae()`. Finally, the predicted values are computed as follows: `params[0] + feature1*params[1] + feature2*params[2]`. Note that we've defined a vector of parameters, `params`, as a variable, rather than using three variables. Here, `params[0]` is the intercept and `params[1]` and `params[2]` are the slopes.



```
# Define the linear regression model
def linear_regression(params, feature1 = size_log, feature2 = bedrooms):
    return params[0] + feature1*params[1] + feature2*params[2]

# Define the loss function
def loss_function(params, targets = price_log, feature1 = size_log, feature2 = bedrooms):
    # Set the predicted values
    predictions = linear_regression(params, feature1, feature2)

    # Use the mean absolute error loss
    return keras.losses.mae(targets, predictions)

# Define the optimize operation
opt = keras.optimizers.Adam()

# Perform minimization and print trainable variables
for j in range(10):
    opt.minimize(lambda: loss_function(params), var_list=[params])
    print_results(params)
```



```
<script.py> output:
    loss: 12.418, intercept: 0.101, slope_1: 0.051, slope_2: 0.021
    loss: 12.404, intercept: 0.102, slope_1: 0.052, slope_2: 0.022
    loss: 12.391, intercept: 0.103, slope_1: 0.053, slope_2: 0.023
    loss: 12.377, intercept: 0.104, slope_1: 0.054, slope_2: 0.024
    loss: 12.364, intercept: 0.105, slope_1: 0.055, slope_2: 0.025
    loss: 12.351, intercept: 0.106, slope_1: 0.056, slope_2: 0.026
    loss: 12.337, intercept: 0.107, slope_1: 0.057, slope_2: 0.027
    loss: 12.324, intercept: 0.108, slope_1: 0.058, slope_2: 0.028
    loss: 12.311, intercept: 0.109, slope_1: 0.059, slope_2: 0.029
    loss: 12.297, intercept: 0.110, slope_1: 0.060, slope_2: 0.030
```

Note that `params[2]` tells us how much the price will increase in percentage terms if we add one more bedroom. You could train `params[2]` and the other model parameters by increasing the number of times we iterate over `opt`.

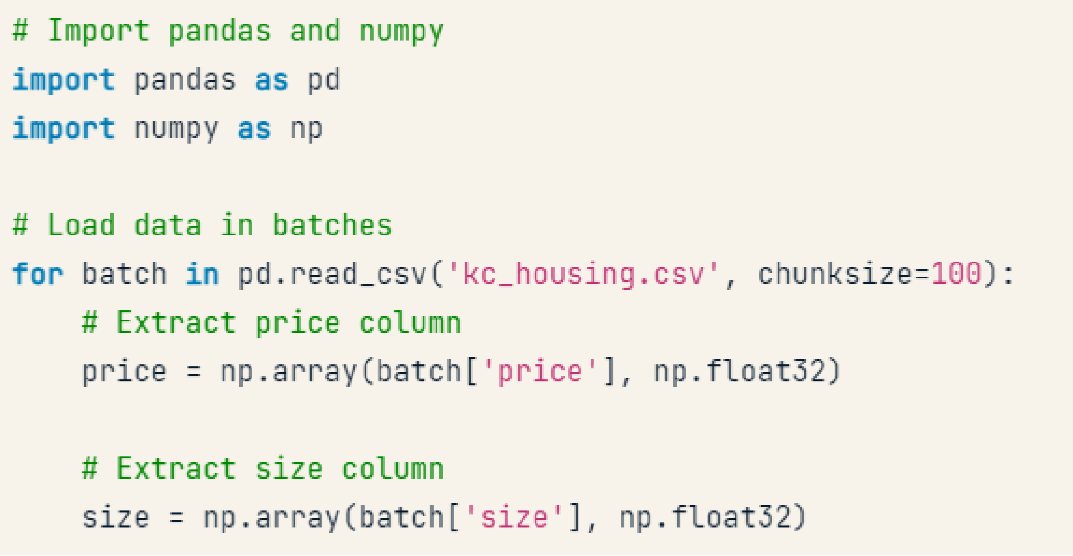
Batch training

Batch training to handle large data sets, unable to fit all in memory.

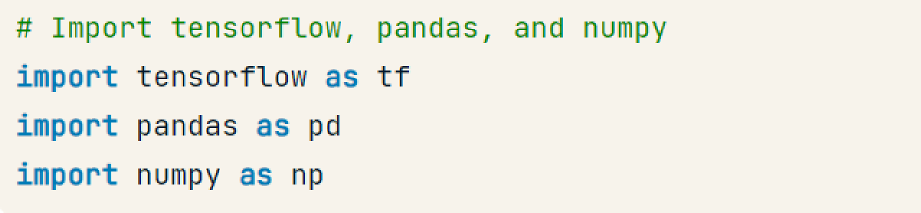
A single pass over all of the batches is called an epoch, and the process itself is called batch training.

Divide it into batches and train on those batches sequentially.

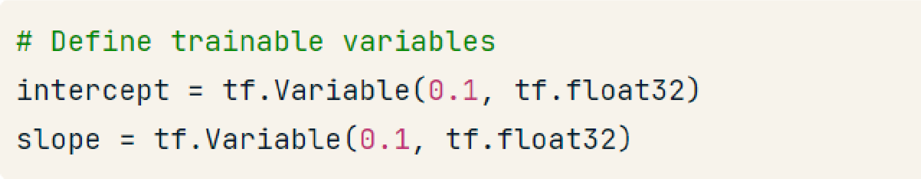
Update model weights and optimizer parameters after each batch, rather than at the end of the epoch.



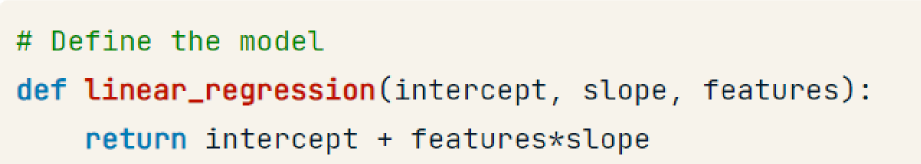
```
# Import pandas and numpy
import pandas as pd
import numpy as np

# Load data in batches
for batch in pd.read_csv('kc_housing.csv', chunksize=100):
    # Extract price column
    price = np.array(batch['price'], np.float32)

    # Extract size column
    size = np.array(batch['size'], np.float32)
```



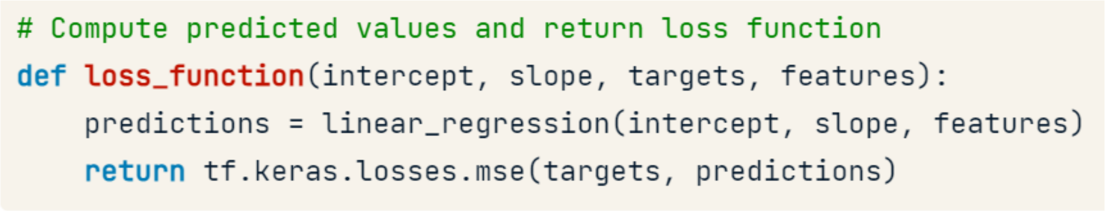
```
# Import tensorflow, pandas, and numpy
import tensorflow as tf
import pandas as pd
import numpy as np
```



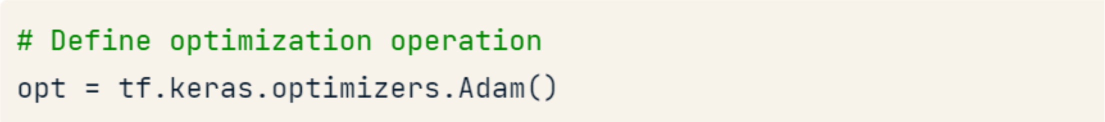
```
# Define trainable variables
intercept = tf.Variable(0.1, tf.float32)
slope = tf.Variable(0.1, tf.float32)
```



```
# Define the model
def linear_regression(intercept, slope, features):
    return intercept + features*slope
```

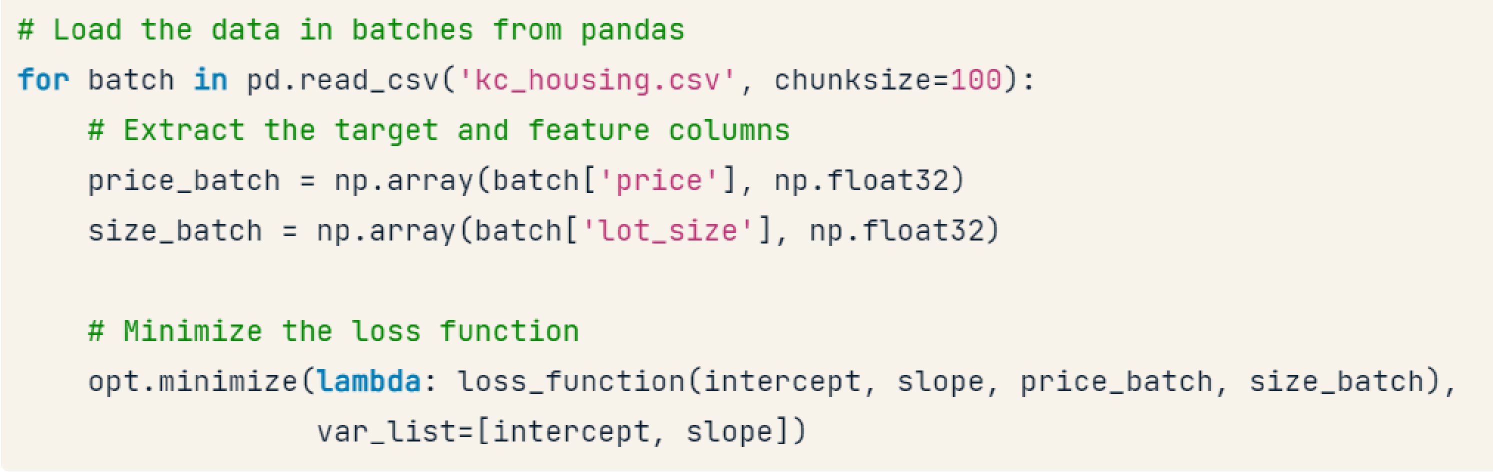


```
# Compute predicted values and return loss function
def loss_function(intercept, slope, targets, features):
    predictions = linear_regression(intercept, slope, features)
    return tf.keras.losses.mse(targets, predictions)
```

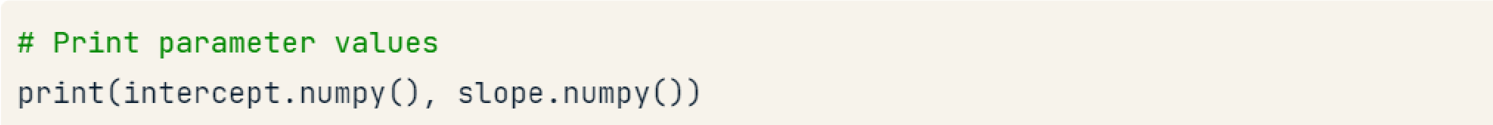


```
# Define optimization operation
opt = tf.keras.optimizers.Adam()
```

Training a linear model in batches



```
# Load the data in batches from pandas
for batch in pd.read_csv('kc_housing.csv', chunksize=100):
    # Extract the target and feature columns
    price_batch = np.array(batch['price'], np.float32)
    size_batch = np.array(batch['lot_size'], np.float32)

    # Minimize the loss function
    opt.minimize(lambda: loss_function(intercept, slope, price_batch, size_batch),
                 var_list=[intercept, slope])
```



```
# Print parameter values
print(intercept.numpy(), slope.numpy())
```

Full Sample Training	Batch Training
One update per epoch	Multiple updates per epoch
Accepts dataset without modification	Requires division of dataset
Limited by memory	No limit on dataset size

Able to automate batch training by using high level APIs

Preparing to batch train

Before we can train a linear model in batches, we must first define **variables**, a **loss function**, and an **optimization operation**. In this exercise, we will prepare to train a model that will predict `price_batch`, a batch of house prices, using `size_batch`, a batch of lot sizes in square feet. In contrast to the previous lesson, we will do this by loading batches of data using `pandas`, converting it to `numpy` arrays, and then using it to minimize the loss function in steps.

`Variable()`, `keras()`, and `float32` have been imported for you. Note that you should not set default argument values for either the model or loss function, since we will generate the data in batches during the training process.

```python
# Define the intercept and slope
intercept = Variable(10.0, float32)
slope = Variable(0.5, float32)

# Define the model
def linear_regression(intercept, slope, features):
    # Define the predicted values
    return intercept + features*slope

# Define the loss function
def loss_function(intercept, slope, targets, features):
    # Define the predicted values
    predictions = linear_regression(intercept, slope, features)

    # Define the MSE loss
    return keras.losses.mse(targets, predictions)
```

Notice that we did not use default argument values for the input data, `features` and `targets`. This is because the input data has not been defined in advance. Instead, with batch training, we will load it during the training process.

Training a linear model in batches

In this exercise, we will train a linear regression model in batches, starting where we left off in the previous exercise. We will do this by stepping through the dataset in batches and updating the model's variables, `intercept` and `slope`, after each step. This approach will allow us to train with datasets that are otherwise too large to hold in memory.

Note that the loss function,`loss_function(intercept, slope, targets, features)`, has been defined for you. Additionally, `keras` has been imported for you and `numpy` is available as `np`. The trainable variables should be entered into `var_list` in the order in which they appear as loss function arguments.

```
# Initialize adam optimizer
opt = keras.optimizers.Adam()

# Load data in batches
for batch in pd.read_csv('kc_house_data.csv', chunksize=100):
    size_batch = np.array(batch['sqft_lot'], np.float32)

    # Extract the price values for the current batch
    price_batch = np.array(batch['price'], np.float32)

    # Complete the loss, fill in the variable list, and minimize
    opt.minimize(lambda: loss_function(intercept, slope, price_batch, size_batch), var_list=[intercept, slope])

# Print trained parameters
print(intercept.numpy(), slope.numpy())
```

```
<script.py> output:
    10.217888 0.7016
```

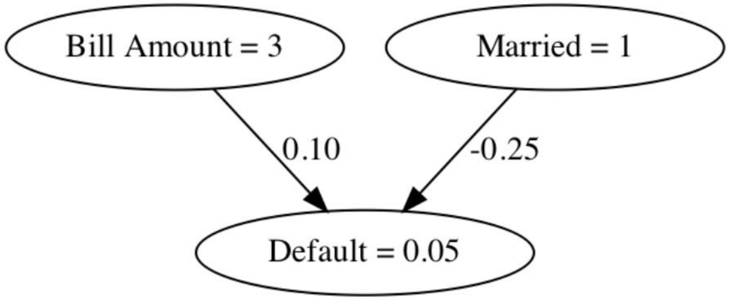
Batch training will be very useful when you train neural networks, which we will do next.

Chapter 3. Neural Networks

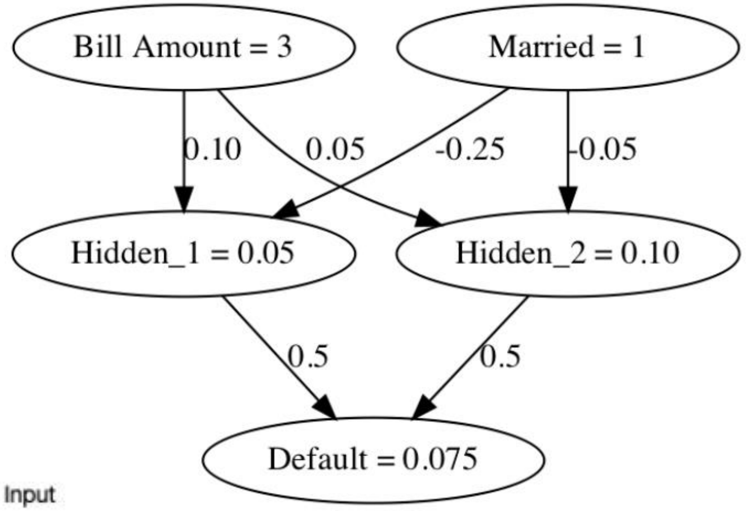
The previous chapters taught you how to build models in TensorFlow 2. In this chapter, you will apply those same tools to build, train, and make predictions with neural networks. You will learn how to define dense layers, apply activation functions, select an optimizer, and apply regularization to reduce overfitting. You will take advantage of TensorFlow's flexibility by using both low-level linear algebra and high-level Keras API operations to define and train models.

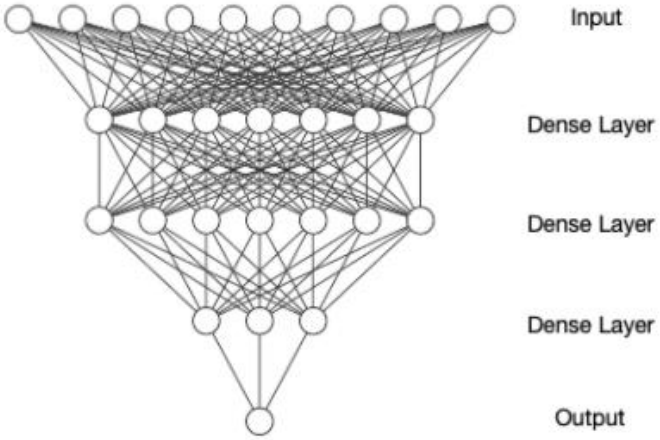
Dense layers

Linear regression model to predict credit card defaults



Neural network model to predict credit card defaults

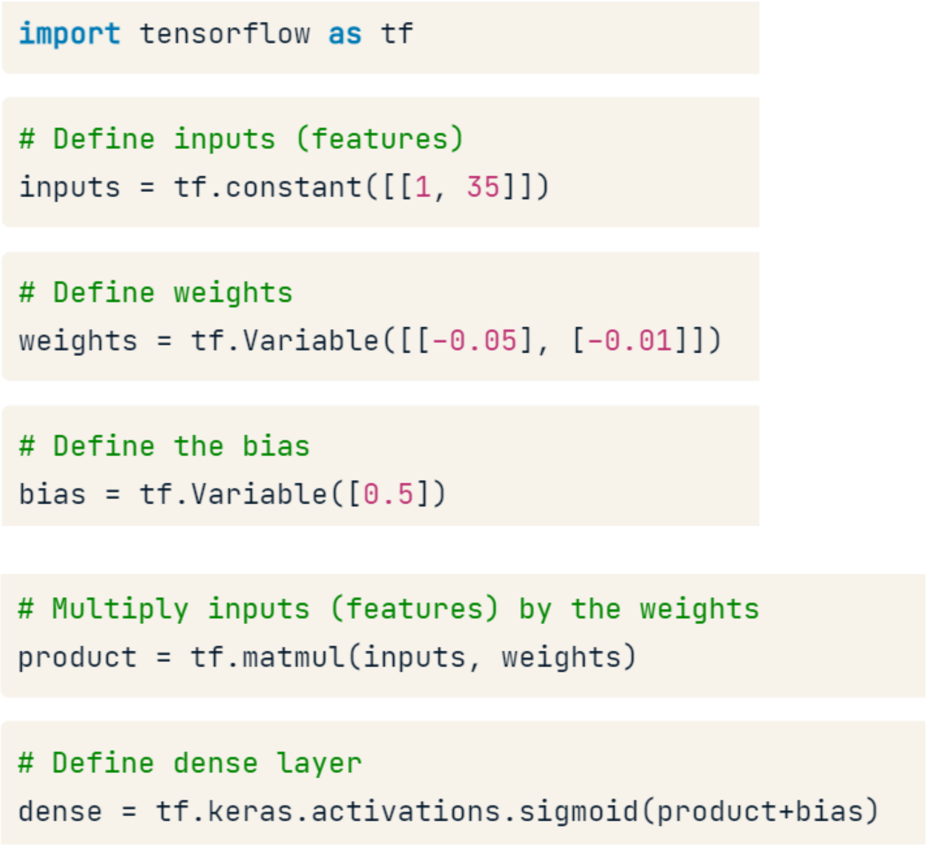




Input = features

Dense = applies weights and activation functions to all nodes

Output = prediction



```python
import tensorflow as tf
```

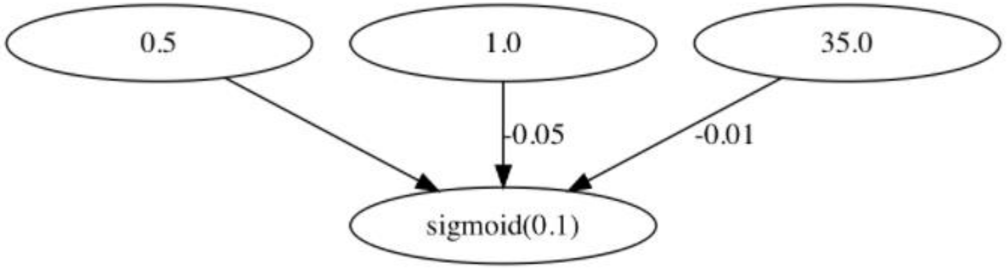
```python
# Define inputs (features)
inputs = tf.constant([[1, 35]])
```

```python
# Define weights
weights = tf.Variable([[-0.05], [-0.01]])
```

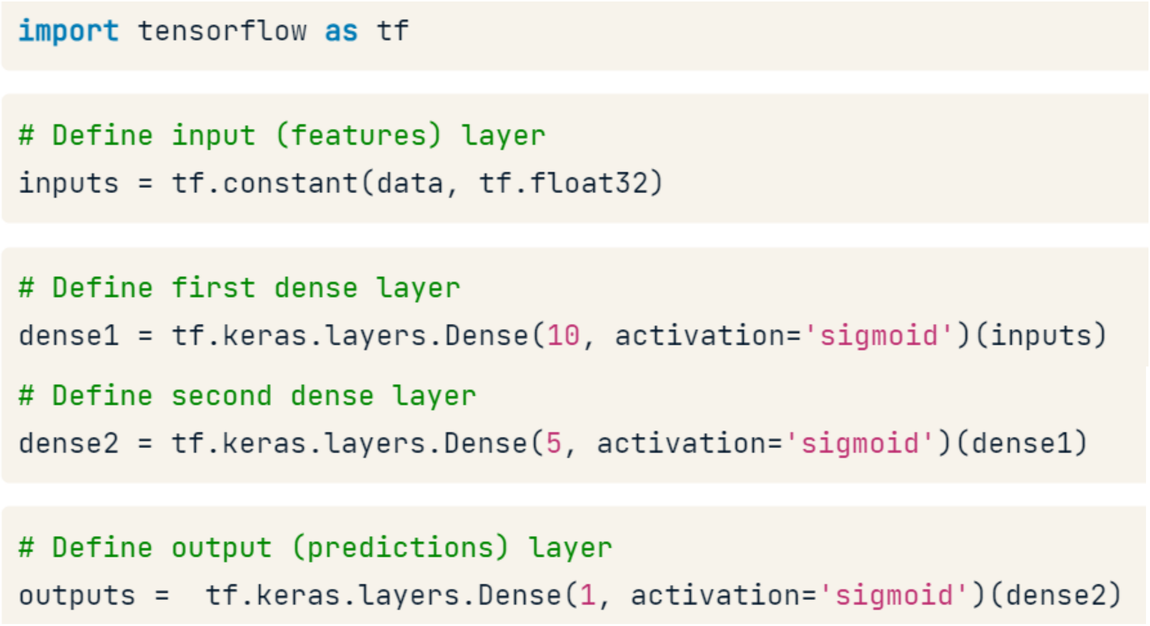
```python
# Define the bias
bias = tf.Variable([0.5])
```

```python
# Multiply inputs (features) by the weights
product = tf.matmul(inputs, weights)
```

```python
# Define dense layer
dense = tf.keras.activations.sigmoid(product+bias)
```



The bias is not associated with a feature, and is analogous to the intercept in a linear regression.

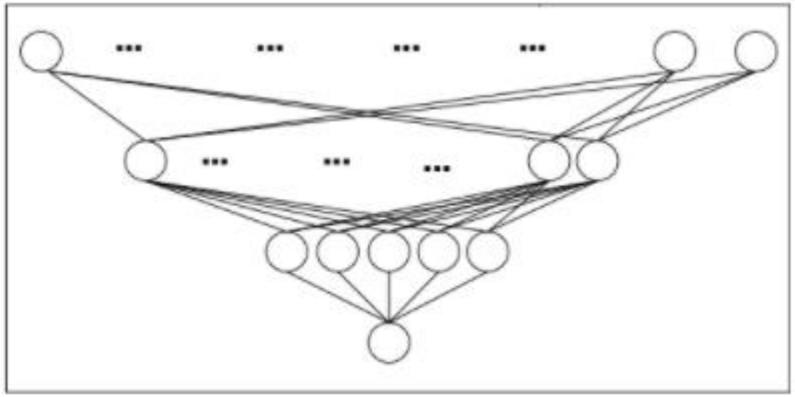


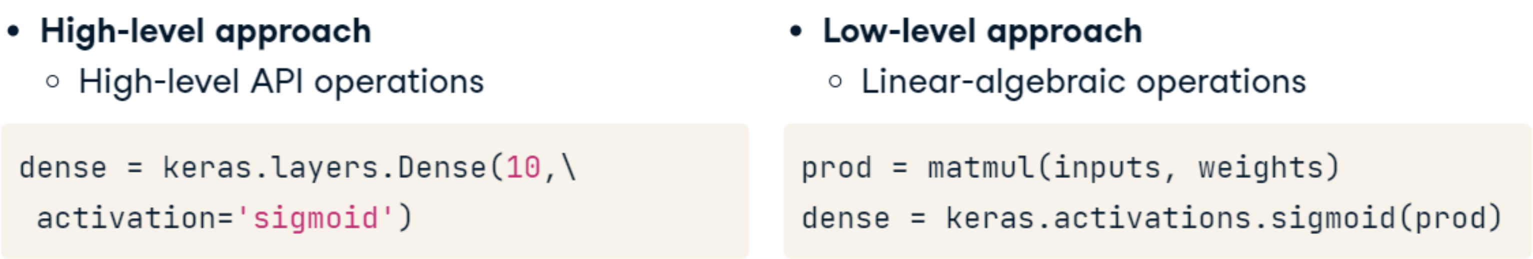
```python
import tensorflow as tf
```

```python
# Define input (features) layer
inputs = tf.constant(data, tf.float32)
```

```python
# Define first dense layer
dense1 = tf.keras.layers.Dense(10, activation='sigmoid')(inputs)
# Define second dense layer
dense2 = tf.keras.layers.Dense(5, activation='sigmoid')(dense1)
```

```python
# Define output (predictions) layer
outputs =  tf.keras.layers.Dense(1, activation='sigmoid')(dense2)
```





- **High-level approach**
  - High-level API operations

```
dense = keras.layers.Dense(10,\
 activation='sigmoid')
```

- **Low-level approach**
  - Linear-algebraic operations

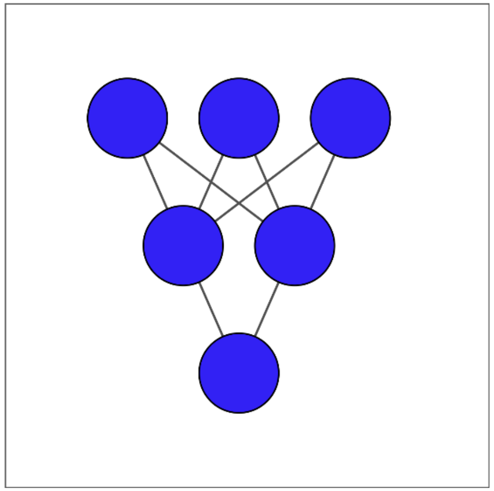
```
prod = matmul(inputs, weights)
dense = keras.activations.sigmoid(prod)
```

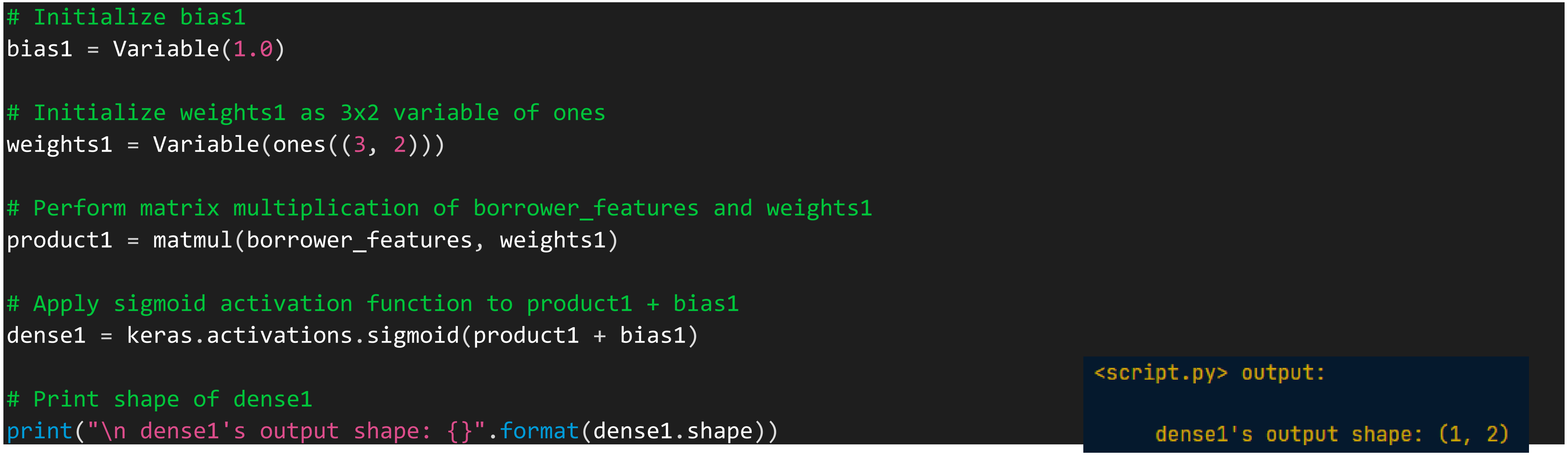
The linear algebra of dense layers (skip)

There are two ways to define a dense layer in `tensorflow`. The first involves the use of low-level, linear algebraic operations. The second makes use of high-level `keras` operations. In this exercise, we will use the first method to construct the network shown on the right.

The input layer contains 3 features -- education, marital status, and age -- which are available as `borrower_features`. The hidden layer contains 2 nodes and the output layer contains a single node.

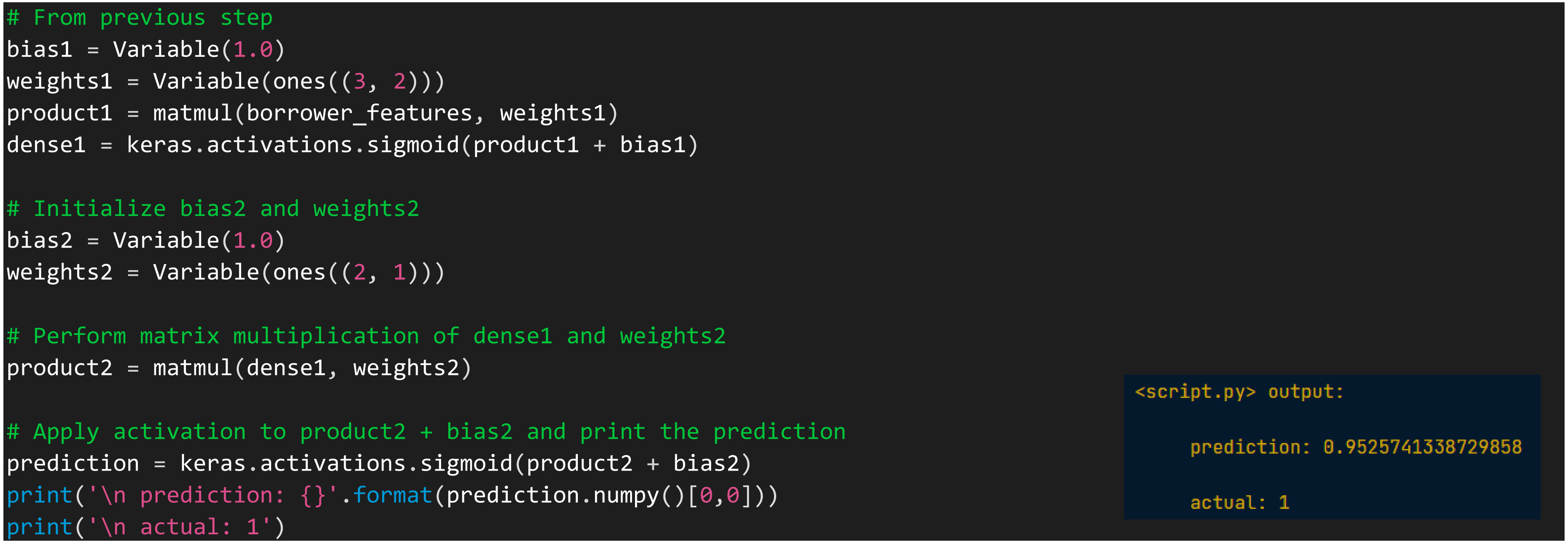
For each layer, you will take the previous layer as an input, initialize a set of weights, compute the product of the inputs and weights, and then apply an activation function. Note that `Variable()`, `ones()`, `matmul()`, and `keras()` have been imported from `tensorflow`.





```
# Initialize bias1
bias1 = Variable(1.0)

# Initialize weights1 as 3x2 variable of ones
weights1 = Variable(ones((3, 2)))

# Perform matrix multiplication of borrower_features and weights1
product1 = matmul(borrower_features, weights1)

# Apply sigmoid activation function to product1 + bias1
dense1 = keras.activations.sigmoid(product1 + bias1)

# Print shape of dense1
print("\n dense1's output shape: {}".format(dense1.shape))
```

```
<script.py> output:

     dense1's output shape: (1, 2)
```



```
# From previous step
bias1 = Variable(1.0)
weights1 = Variable(ones((3, 2)))
product1 = matmul(borrower_features, weights1)
dense1 = keras.activations.sigmoid(product1 + bias1)

# Initialize bias2 and weights2
bias2 = Variable(1.0)
weights2 = Variable(ones((2, 1)))

# Perform matrix multiplication of dense1 and weights2
product2 = matmul(dense1, weights2)

# Apply activation to product2 + bias2 and print the prediction
prediction = keras.activations.sigmoid(product2 + bias2)
print('\n prediction: {}'.format(prediction.numpy()[0,0]))
print('\n actual: 1')
```

```
<script.py> output:

     prediction: 0.9525741338729858

     actual: 1
```

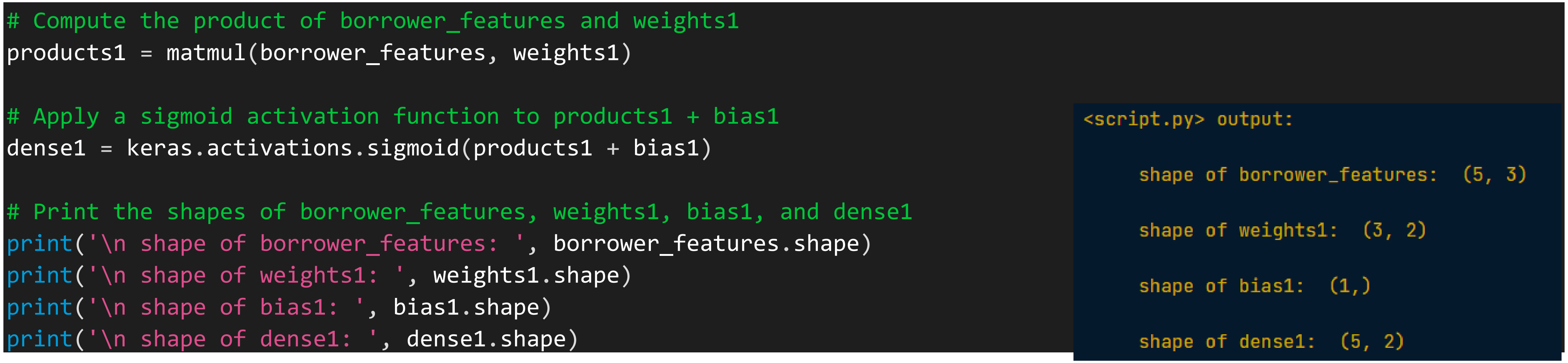
Our model produces predicted values in the interval between 0 and 1. For the example we considered, the actual value was 1 and the predicted value was a probability between 0 and 1. This, of course, is not meaningful, since we have not yet trained our model's parameters.

The low-level approach with multiple examples (skip)

In this exercise, we'll build further intuition for the low-level approach by constructing the first dense hidden layer for the case where we have multiple examples. We'll assume the model is trained and the first layer weights, `weights1`, and bias, `bias1`, are available. We'll then perform matrix multiplication of the `borrower_features` tensor by the `weights1` variable. Recall that the `borrower_features` tensor includes education, marital status, and age. Finally, we'll apply the sigmoid function to the elements of `products1 + bias1`, yielding `dense1`.

$$products1 = \begin{bmatrix} 3 & 3 & 23 \\ 2 & 1 & 24 \\ 1 & 1 & 49 \\ 1 & 1 & 49 \\ 2 & 1 & 29 \end{bmatrix} \begin{bmatrix} -0.6 & 0.6 \\ 0.8 & -0.3 \\ -0.09 & -0.08 \end{bmatrix}$$

Note that `matmul()` and `keras()` have been imported from `tensorflow`.

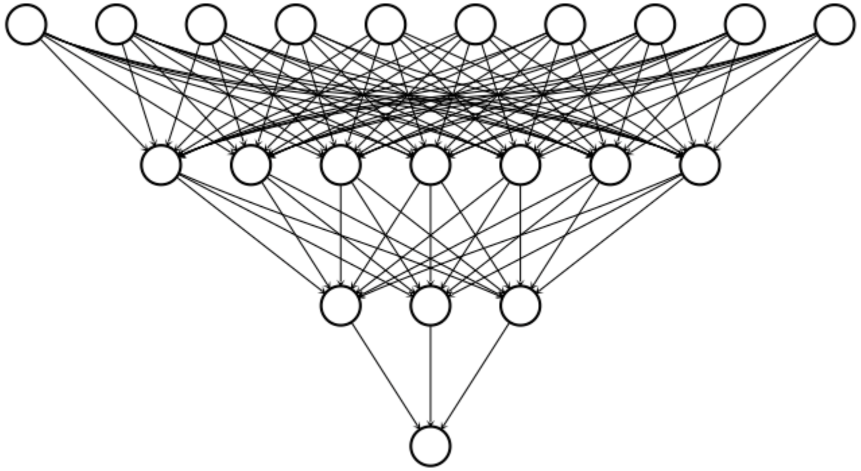


```
# Compute the product of borrower_features and weights1
products1 = matmul(borrower_features, weights1)

# Apply a sigmoid activation function to products1 + bias1
dense1 = keras.activations.sigmoid(products1 + bias1)

# Print the shapes of borrower_features, weights1, bias1, and dense1
print('\n shape of borrower_features: ', borrower_features.shape)
print('\n shape of weights1: ', weights1.shape)
print('\n shape of bias1: ', bias1.shape)
print('\n shape of dense1: ', dense1.shape)
```

```
<script.py> output:

     shape of borrower_features:  (5, 3)

     shape of weights1:  (3, 2)

     shape of bias1:  (1,)

     shape of dense1:  (5, 2)
```

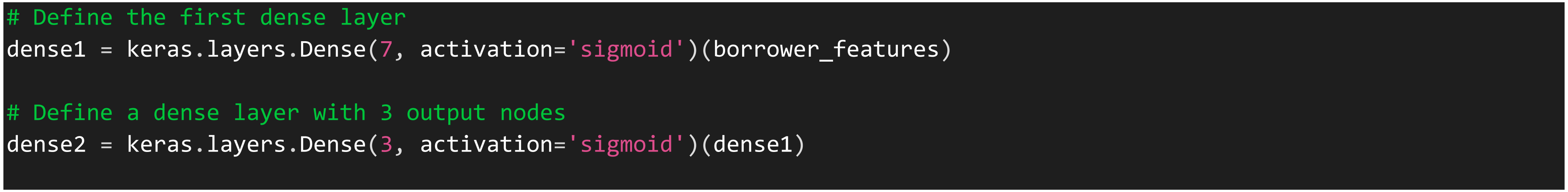
Note that our input data, `borrower_features`, is 5x3 because it consists of 5 examples for 3 features. The shape of `weights1` is 3x2, as it was in the previous exercise, since it does not depend on the number of examples. Additionally, `bias1` is a scalar. Finally, `dense1` is 5x2, which means that we can multiply it by the following set of weights, `weights2`, which we defined to be 2x1 in the previous exercise.

Using the dense layer operation (use this!)

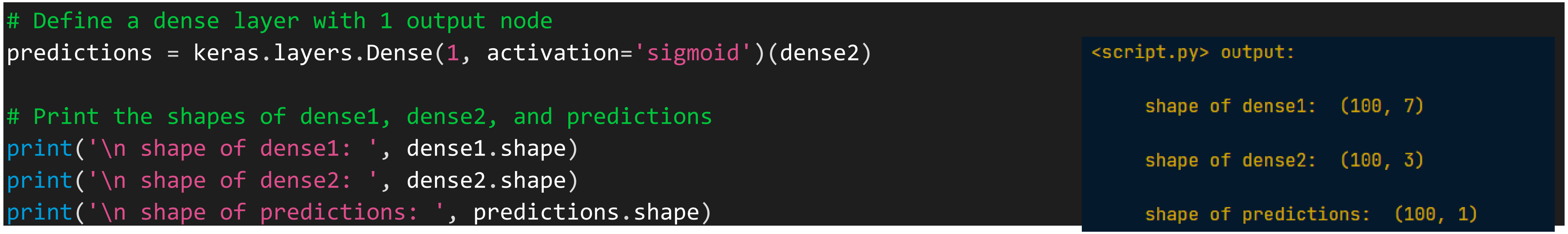


We've now seen how to define dense layers in `tensorflow` using linear algebra. In this exercise, we'll skip the linear algebra and let `keras` work out the details. This will allow us to construct the network below, which has 2 hidden layers and 10 features, using less code than we needed for the network with 1 hidden layer and 3 features.

To construct this network, we'll need to define three dense layers, each of which takes the previous layer as an input, multiplies it by weights, and applies an activation function. Note that input data has been defined and is available as a 100x10 tensor: `borrower_features`. Additionally, the `keras.layers` module is available.



```
# Define the first dense layer
dense1 = keras.layers.Dense(7, activation='sigmoid')(borrower_features)

# Define a dense layer with 3 output nodes
dense2 = keras.layers.Dense(3, activation='sigmoid')(dense1)
```



```
# Define a dense layer with 1 output node
predictions = keras.layers.Dense(1, activation='sigmoid')(dense2)

# Print the shapes of dense1, dense2, and predictions
print('\n shape of dense1: ', dense1.shape)
print('\n shape of dense2: ', dense2.shape)
print('\n shape of predictions: ', predictions.shape)
```

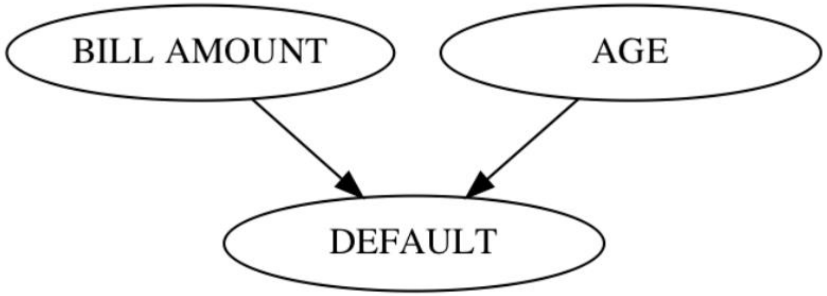
```
<script.py> output:

     shape of dense1:  (100, 7)

     shape of dense2:  (100, 3)

     shape of predictions:  (100, 1)
```

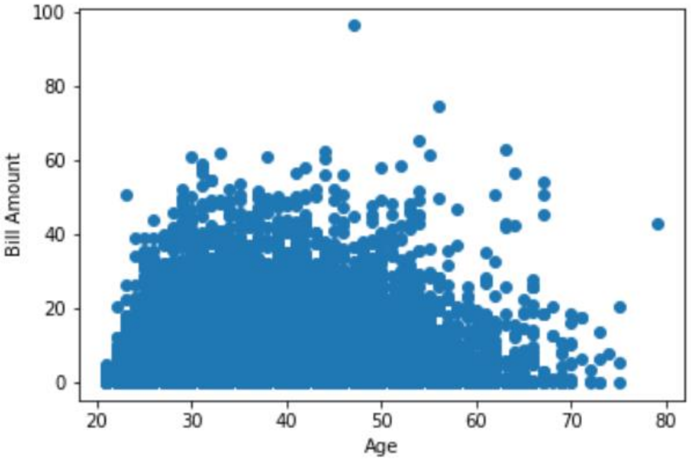
With just 8 lines of code, you were able to define 2 dense hidden layers and an output layer. This is the advantage of using high-level operations in `tensorflow`. Note that each layer has 100 rows because the input data contains 100 examples.

Activation functions

Components of a typical hidden layer

- Linear operation: matrix multiplication
- Non-Linear operation: activation function

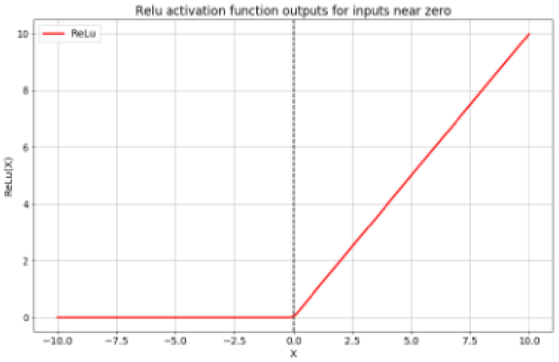




Bill Amount and Age have non-linear relationship

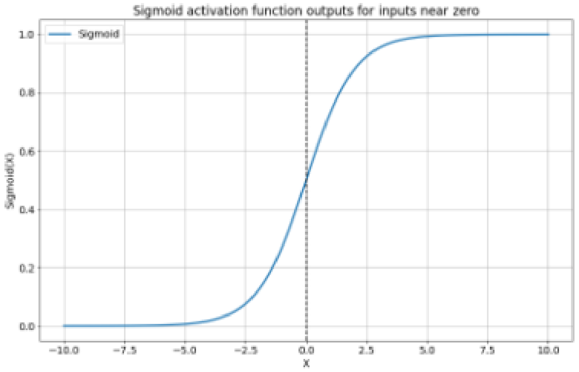
ReLu activation function

- Hidden layers
- Low-level: `tf.keras.activations.relu()`
- High-level: `relu`



Sigmoid activation function

- Output binary classification
- Low-level: `tf.keras.activations.sigmoid()`
- High-level: `sigmoid`



Softmax activation function

- Output classification (>2 classes)
- Low-level: `tf.keras.activations.softmax()`
- High-level: `softmax`

```python
import tensorflow as tf
```

```python
# Define input layer
inputs = tf.constant(borrower_features, tf.float32)
```

```python
# Define dense layer 1
dense1 = tf.keras.layers.Dense(16, activation='relu')(inputs)
```

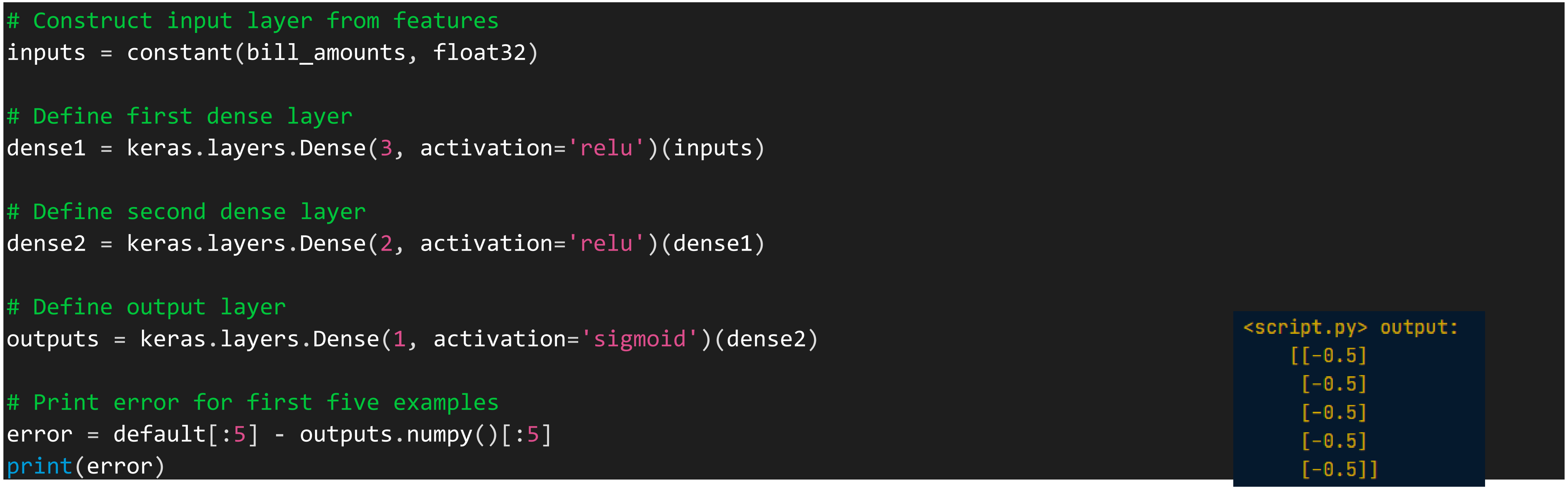
```python
# Define dense layer 2
dense2 = tf.keras.layers.Dense(8, activation='sigmoid')(dense1)
```

```python
# Define output layer
outputs = tf.keras.layers.Dense(4, activation='softmax')(dense2)
```

Binary classification problems

In this exercise, you will again make use of credit card data. The target variable, `default`, indicates whether a credit card holder defaults on his or her payment in the following period. Since there are only two options--default or not--this is a binary classification problem. While the dataset has many features, you will focus on just three: the size of the three latest credit card bills. Finally, you will compute predictions from your untrained network, `outputs`, and compare those the target variable, `default`.

The tensor of features has been loaded and is available as `bill_amounts`. Additionally, the `constant()`, `float32`, and `keras.layers.Dense()` operations are available.



```
# Construct input layer from features
inputs = constant(bill_amounts, float32)

# Define first dense layer
dense1 = keras.layers.Dense(3, activation='relu')(inputs)

# Define second dense layer
dense2 = keras.layers.Dense(2, activation='relu')(dense1)

# Define output layer
outputs = keras.layers.Dense(1, activation='sigmoid')(dense2)

# Print error for first five examples
error = default[:5] - outputs.numpy()[:5]
print(error)
```

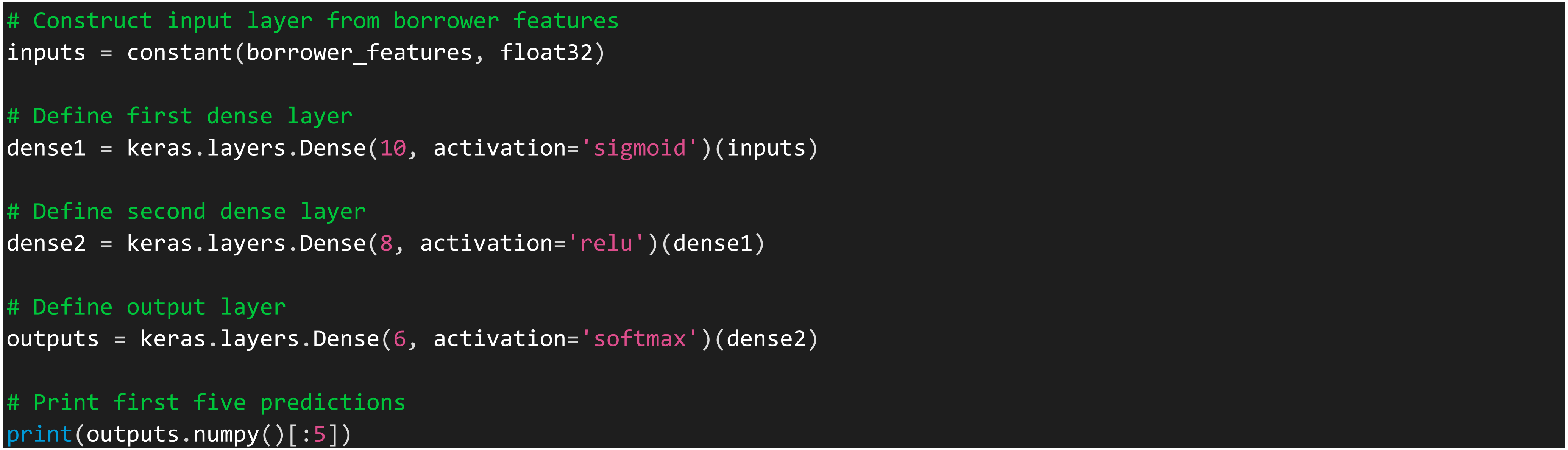
```
<script.py> output:
    [[-0.5]
     [-0.5]
     [-0.5]
     [-0.5]
     [-0.5]]
```

If you run the code several times, you'll notice that the errors change each time. This is because you're using an untrained model with randomly initialized parameters. Furthermore, the errors fall on the interval between -1 and 1 because `default` is a binary variable that takes on values of 0 and 1 and `outputs` is a probability between 0 and 1.

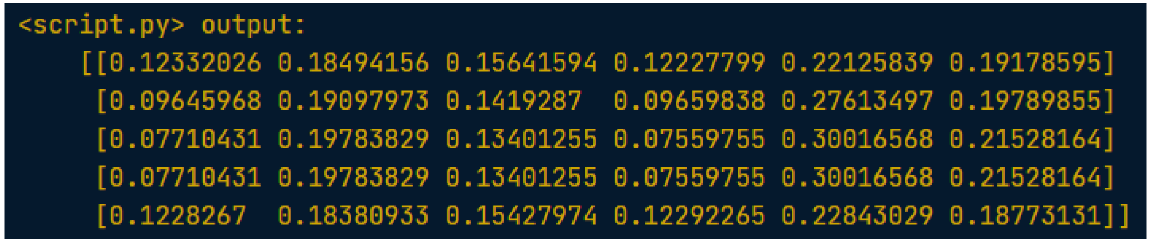
Multiclass classification problems

In this exercise, we expand beyond binary classification to cover multiclass problems. A multiclass problem has targets that can take on three or more values. In the credit card dataset, the education variable can take on 6 different values, each corresponding to a different level of education. We will use that as our target in this exercise and will also expand the feature set from 3 to 10 columns.

As in the previous problem, you will define an input layer, dense layers, and an output layer. You will also print the untrained model's predictions, which are probabilities assigned to the classes. The tensor of features has been loaded and is available as `borrower_features`. Additionally, the `constant()`, `float32`, and `keras.layers.Dense()` operations are available.



```python
# Construct input layer from borrower features
inputs = constant(borrower_features, float32)

# Define first dense layer
dense1 = keras.layers.Dense(10, activation='sigmoid')(inputs)

# Define second dense layer
dense2 = keras.layers.Dense(8, activation='relu')(dense1)

# Define output layer
outputs = keras.layers.Dense(6, activation='softmax')(dense2)

# Print first five predictions
print(outputs.numpy()[:5])
```



```
<script.py> output:
    [[0.12332026 0.18494156 0.15641594 0.12227799 0.22125839 0.19178595]
     [0.09645968 0.19097973 0.1419287  0.09659838 0.27613497 0.19789855]
     [0.07710431 0.19783829 0.13401255 0.07559755 0.30016568 0.21528164]
     [0.07710431 0.19783829 0.13401255 0.07559755 0.30016568 0.21528164]
     [0.1228267  0.18380933 0.15427974 0.12292265 0.22843029 0.18773131]]
```

Notice that each row of `outputs` sums to one. This is because a row contains the predicted class probabilities for one example. As with the previous exercise, our predictions are not yet informative, since we are using an untrained model with randomly initialized parameters. This is why the model tends to assign similar probabilities to each class.

Optimizers

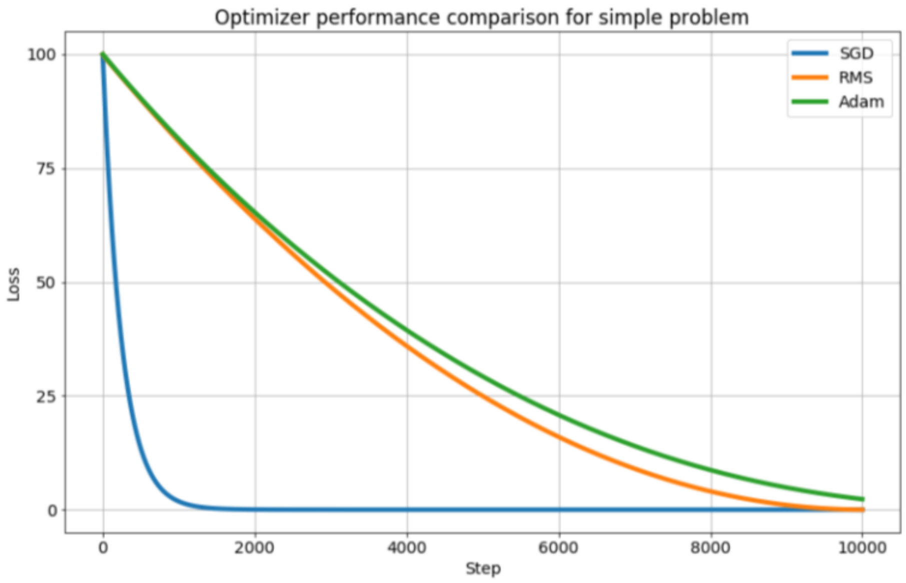
Optimizers find the minimum value in a loss function, using a gradient descent algorithm.
Stochastic gradient descent (SGD) is an improved version of gradient descent that is less likely to get stuck in a local minima.

Stochastic gradient descent (SGD) optimizer

- `tf.keras.optimizers.SGD()`
- `learning_rate` from 0.001 to 0.5
- simple and easy to interpret

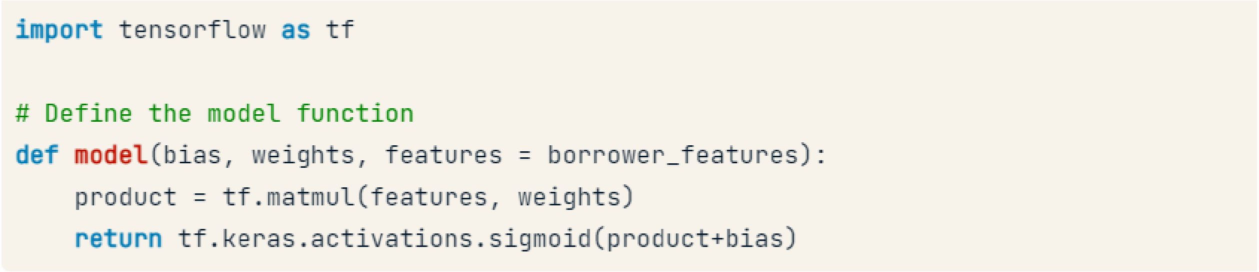
RMS propagation optimizer

- applies different leaarning rates to each feature
- `tf.keras.optimizers.RMSprop()`
- `learning_rate`
- `momentum`
- `decay`
- allows for momentum to both build and decay

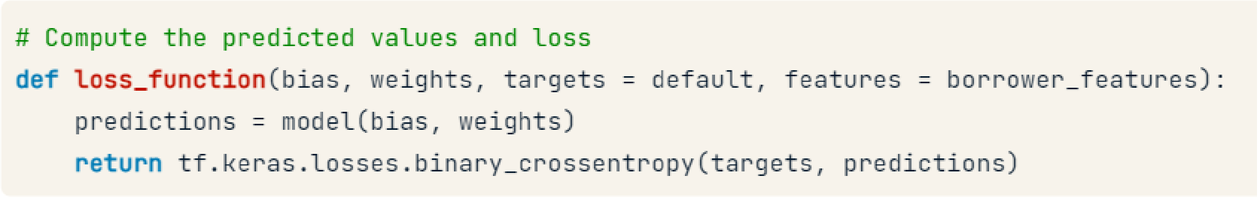


Adaptive moment (Adam) optimizer

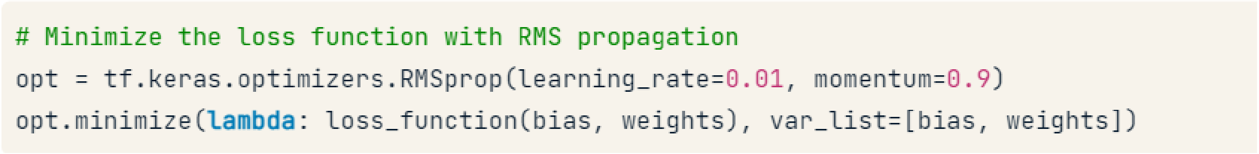
- `tf.keras.optimizers.Adam()`
- `learning_rate`
- `beta1` → set momentum to decay faster by lowering beta1 parameter
- performs well with default parameter values



```python
import tensorflow as tf

# Define the model function
def model(bias, weights, features = borrower_features):
    product = tf.matmul(features, weights)
    return tf.keras.activations.sigmoid(product+bias)
```



```python
# Compute the predicted values and loss
def loss_function(bias, weights, targets = default, features = borrower_features):
    predictions = model(bias, weights)
    return tf.keras.losses.binary_crossentropy(targets, predictions)
```

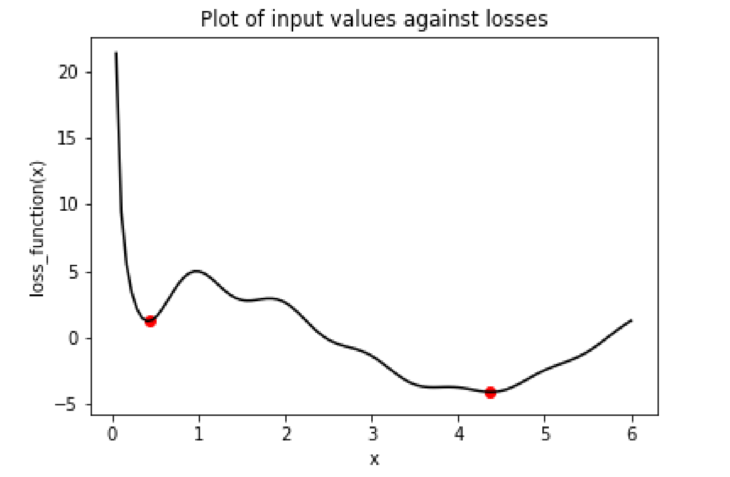


```python
# Minimize the loss function with RMS propagation
opt = tf.keras.optimizers.RMSprop(learning_rate=0.01, momentum=0.9)
opt.minimize(lambda: loss_function(bias, weights), var_list=[bias, weights])
```

The dangers of local minima

Consider the plot of the following loss function, `loss_function()`, which contains a global minimum, marked by the dot on the right, and several local minima, including the one marked by the dot on the left.

In this exercise, you will try to find the global minimum of `loss_function()` using `keras.optimizers.SGD()`. You will do this twice, each time with a different initial value of the input to `loss_function()`. First, you will use `x_1`, which is a variable with an initial value of 6.0. Second, you will use `x_2`, which is a variable with an initial value of 0.3. Note that `loss_function()` has been defined and is available.



```
# Initialize x_1 and x_2
x_1 = Variable(6.0,float32)
x_2 = Variable(0.3,float32)

# Define the optimization operation
opt = keras.optimizers.SGD(learning_rate=0.01)

for j in range(100):
    # Perform minimization using the loss function and x_1
    opt.minimize(lambda: loss_function(x_1), var_list=[x_1])
    # Perform minimization using the loss function and x_2
    opt.minimize(lambda: loss_function(x_2), var_list=[x_2])

# Print x_1 and x_2 as numpy arrays
print(x_1.numpy(), x_2.numpy())
```

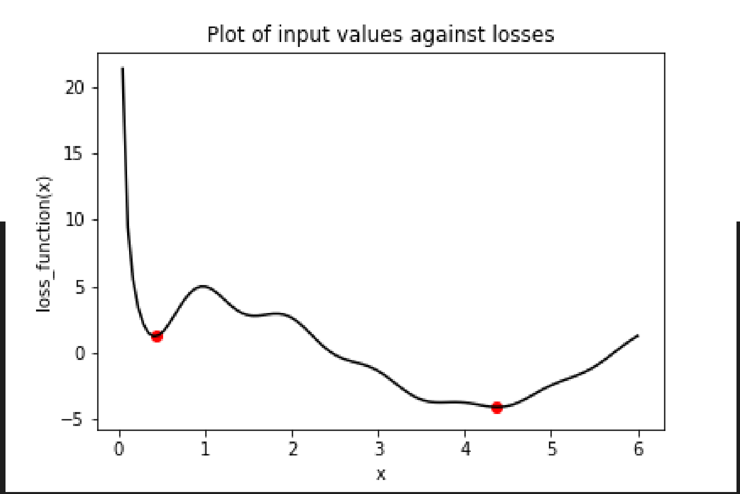
```
<script.py> output:
    4.3801394 0.42052683
```

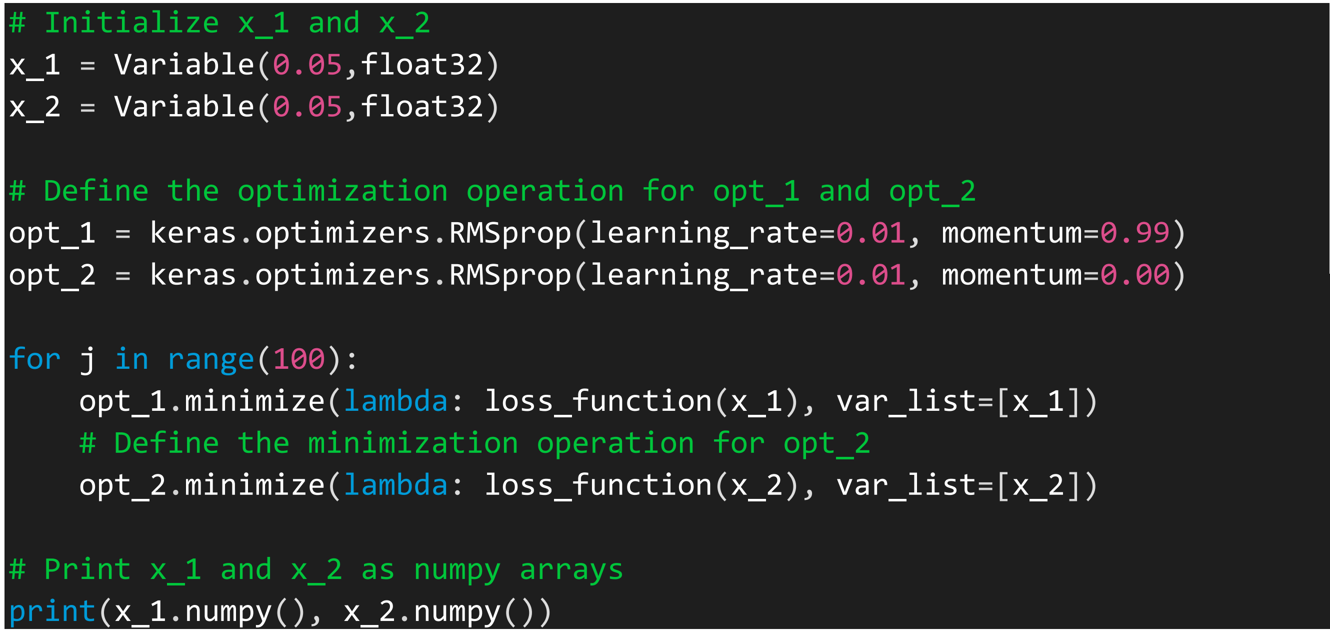
Notice that we used the same optimizer and loss function, but two different initial values. When we started at 6.0 with `x_1`, we found the global minimum at 4.38, marked by the dot on the right. When we started at 0.3, we stopped around 0.42 with `x_2`, the local minimum marked by a dot on the far left.

Avoiding local minima

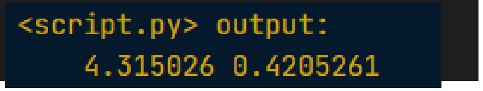
The previous problem showed how easy it is to get stuck in local minima. We had a simple optimization problem in one variable and gradient descent still failed to deliver the global minimum when we had to travel through local minima first. One way to avoid this problem is to use momentum, which allows the optimizer to break through local minima. We will again use the loss function from the previous problem, which has been defined and is available for you as `loss_function()`.

Several optimizers in `tensorflow` have a momentum parameter, including `SGD` and `RMSprop`. You will make use of `RMSprop` in this exercise. Note that `x_1` and `x_2` have been initialized to the same value this time. Furthermore, `keras.optimizers.RMSprop()` has also been imported for you from `tensorflow`.





```
# Initialize x_1 and x_2
x_1 = Variable(0.05,float32)
x_2 = Variable(0.05,float32)

# Define the optimization operation for opt_1 and opt_2
opt_1 = keras.optimizers.RMSprop(learning_rate=0.01, momentum=0.99)
opt_2 = keras.optimizers.RMSprop(learning_rate=0.01, momentum=0.00)

for j in range(100):
    opt_1.minimize(lambda: loss_function(x_1), var_list=[x_1])
    # Define the minimization operation for opt_2
    opt_2.minimize(lambda: loss_function(x_2), var_list=[x_2])

# Print x_1 and x_2 as numpy arrays
print(x_1.numpy(), x_2.numpy())
```



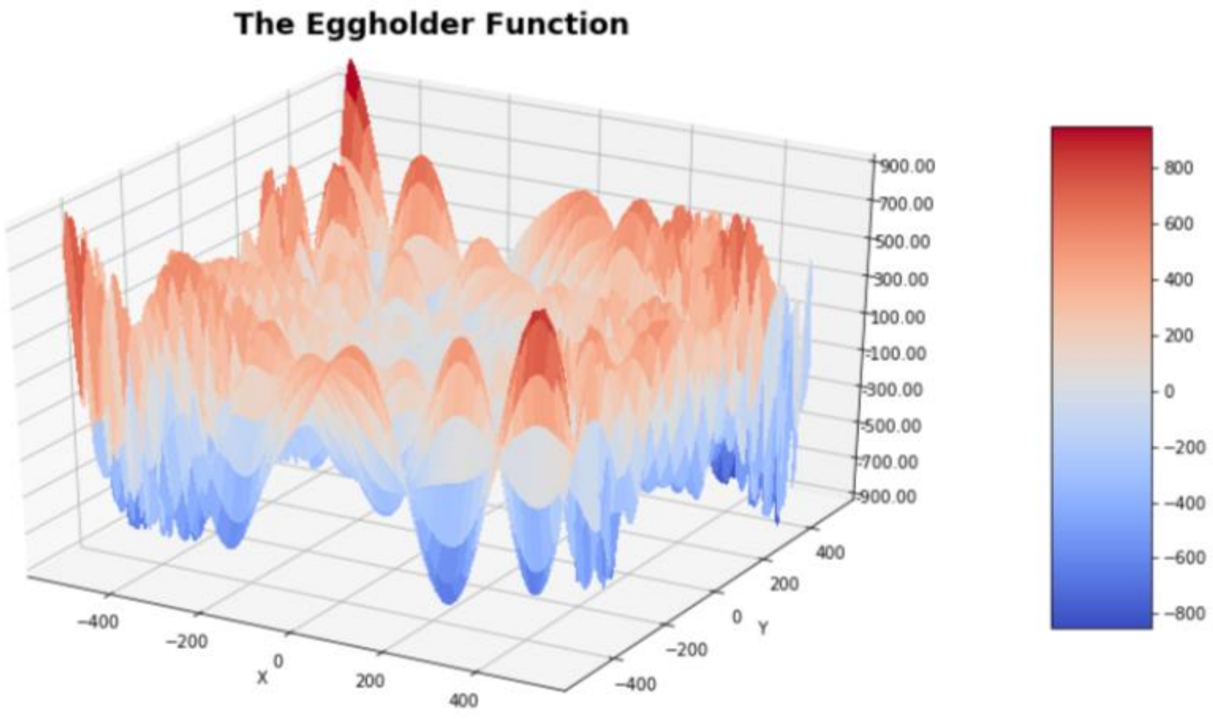
```
<script.py> output:
    4.315026 0.4205261
```

Recall that the global minimum is approximately 4.38. Notice that `opt_1` built momentum, bringing `x_1` closer to the global minimum. To the contrary, `opt_2`, which had a `momentum` parameter of 0.0, got stuck in the local minimum on the left.

Training a network in TensorFlow

To set initial values for x and y

- draw initial values from Normal distribution or Uniform distribution
- Glorot initializer, designed for ML algorithms

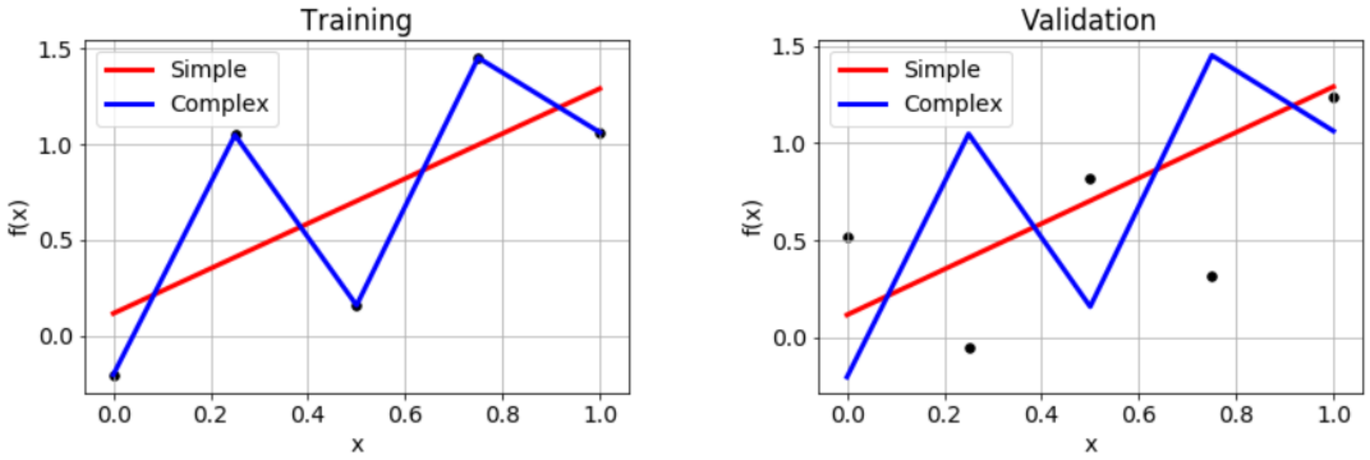


```
import tensorflow as tf

# Define 500x500 random normal variable
weights = tf.Variable(tf.random.normal([500, 500]))

# Define 500x500 truncated random normal variable
weights = tf.Variable(tf.random.truncated_normal([500, 500]))
```

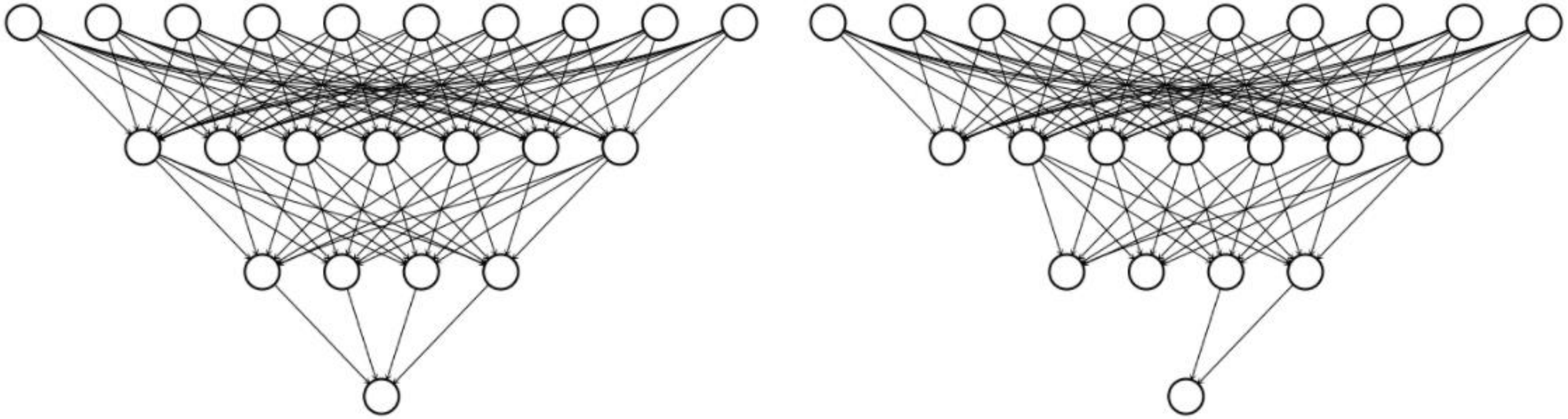
```
# Define a dense layer with the default initializer
dense = tf.keras.layers.Dense(32, activation='relu')

# Define a dense layer with the zeros initializer
dense = tf.keras.layers.Dense(32, activation='relu',\
    kernel_initializer='zeros')
```

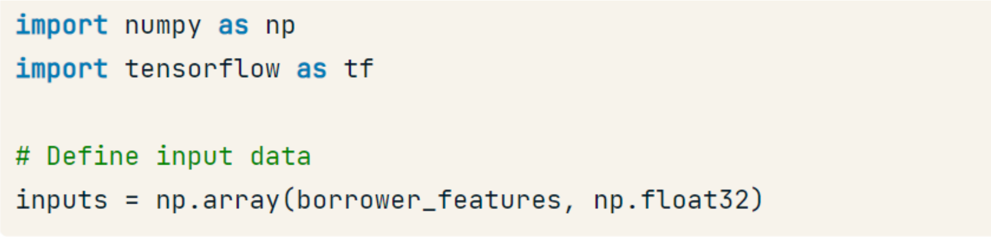
Overfitting

- Complex model performs well in training set, but badly on validation/test set
- Memorized examples rather than learning general patterns, especially problematic for neural networks which contains many parameters and are good at memorization.

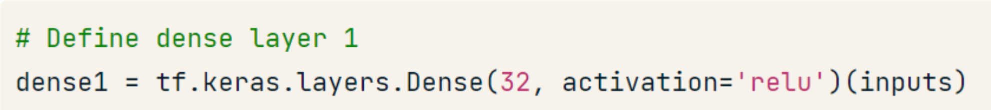


A simple solution is to use **dropout** to prevent overfitting, an operation that will randomly drop the weights connected to certain nodes in a layer during the training process. This will force the network to develop more robust rules for classification, since it cannot rely on any particular nodes being passed to an activation function. This will tend to improve out-of-sample performance.

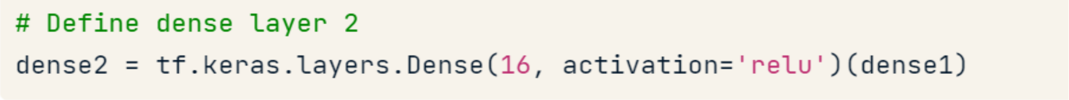




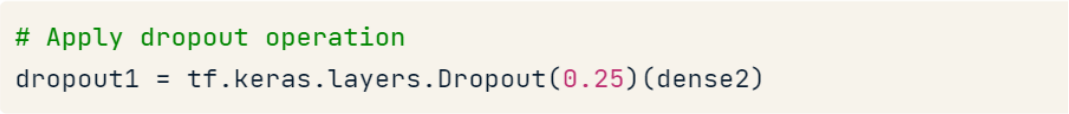
```python
import numpy as np
import tensorflow as tf

# Define input data
inputs = np.array(borrower_features, np.float32)
```



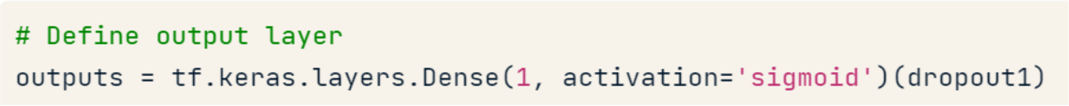
```python
# Define dense layer 1
dense1 = tf.keras.layers.Dense(32, activation='relu')(inputs)
```



```python
# Define dense layer 2
dense2 = tf.keras.layers.Dense(16, activation='relu')(dense1)
```



```python
# Apply dropout operation
dropout1 = tf.keras.layers.Dropout(0.25)(dense2)
```



```python
# Define output layer
outputs = tf.keras.layers.Dense(1, activation='sigmoid')(dropout1)
```

Initialization in TensorFlow

A good initialization can reduce the amount of time needed to find the global minimum. In this exercise, we will initialize weights and biases for a neural network that will be used to predict credit card default decisions. To build intuition, we will use the low-level, linear algebraic approach, rather than making use of convenience functions and high-level `keras` operations. We will also expand the set of input features from 3 to 23. Several operations have been imported from `tensorflow`: `Variable()`, `random()`, and `ones()`.

```python
# Define the layer 1 weights
w1 = Variable(random.normal([23, 7]))

# Initialize the layer 1 bias
b1 = Variable(ones([7]))

# Define the layer 2 weights
w2 = Variable(random.normal([7, 1]))
```

```
# Define the layer 2 bias
b2 = Variable(0.0)
```

In the next exercise, you will start where we've ended and will finish constructing the neural network.

Defining the model and loss function

In this exercise, you will train a neural network to predict whether a credit card holder will default. The features and targets you will use to train your network are available in the Python shell as `borrower_features` and `default`. You defined the weights and biases in the previous exercise.

Note that the `predictions` layer is defined as σ(layer1*w2+b2), where σ is the sigmoid activation, `layer1` is a tensor of nodes for the first hidden dense layer, `w2` is a tensor of weights, and `b2` is the bias tensor.

The trainable variables are `w1`, `b1`, `w2`, and `b2`. Additionally, the following operations have been imported for you: `keras.activations.relu()` and `keras.layers.Dropout()`.

```
# Define the model
def model(w1, b1, w2, b2, features = borrower_features):
    # Apply relu activation functions to layer 1
    layer1 = keras.activations.relu(matmul(features, w1) + b1)
    # Apply dropout
    dropout = keras.layers.Dropout(0.25)(layer1)
    return keras.activations.sigmoid(matmul(dropout, w2) + b2)

# Define the loss function
def loss_function(w1, b1, w2, b2, features = borrower_features, targets = default):
    predictions = model(w1, b1, w2, b2)
    # Pass targets and predictions to the cross entropy loss
    return keras.losses.binary_crossentropy(targets, predictions)
```

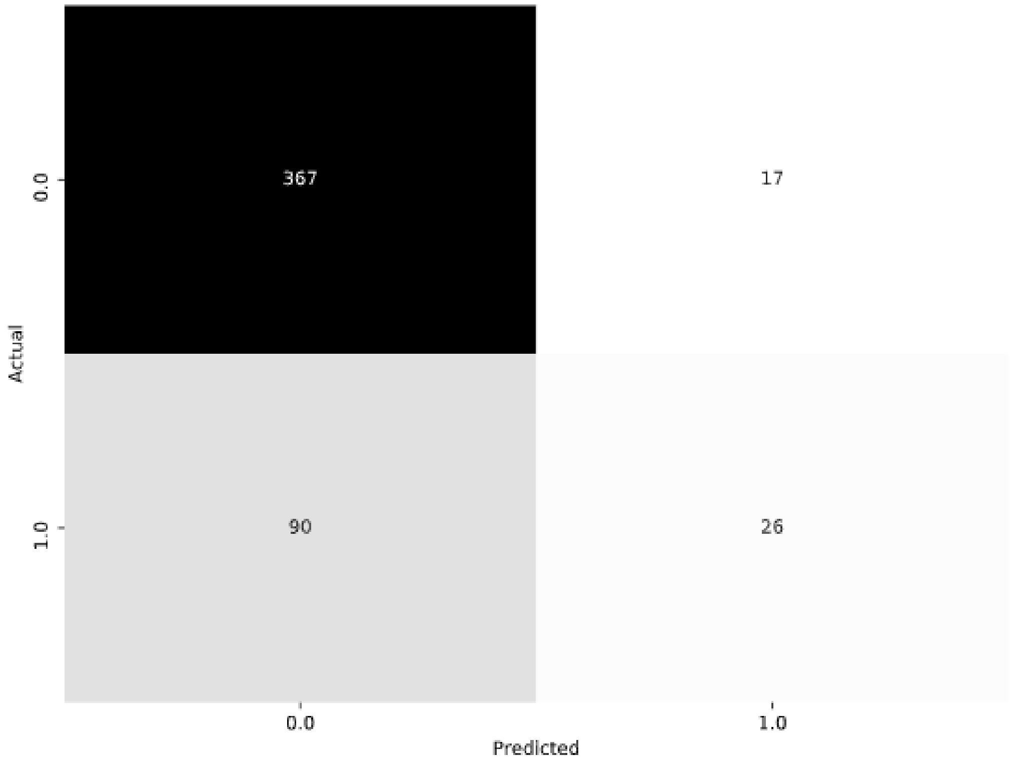
One of the benefits of using `tensorflow` is that you have the option to customize models down to the linear algebraic-level, as we've shown in the last two exercises. If you print `w1`, you can see that the objects we're working with are simply tensors.

Training neural networks with TensorFlow

In the previous exercise, you defined a model, `model(w1, b1, w2, b2, features)`, and a loss function, `loss_function(w1, b1, w2, b2, features, targets)`, both of which are available to you in this exercise. You will now train the model and then evaluate its performance by predicting default outcomes in a test set, which consists of `test_features` and `test_targets` and is available to you. The trainable variables are `w1`, `b1`, `w2`, and `b2`. Additionally, the following operations have been imported for you: `keras.activations.relu()` and `keras.layers.Dropout()`.

```python
# Train the model
for j in range(100):
    # Complete the optimizer
    opt.minimize(lambda: loss_function(w1, b1, w2, b2),
                 var_list=[w1, b1, w2, b2])

# Make predictions with model
model_predictions = model(w1, b1, w2, b2, test_features)

# Construct the confusion matrix
confusion_matrix(test_targets, model_predictions)
```

The diagram shown is called a ``confusion matrix.'' The diagonal elements show the number of correct predictions. The off-diagonal elements show the number of incorrect predictions. We can see that the model performs reasonably-well, but does so by overpredicting non-default. This suggests that we may need to train longer, tune the model's hyperparameters, or change the model's architecture.

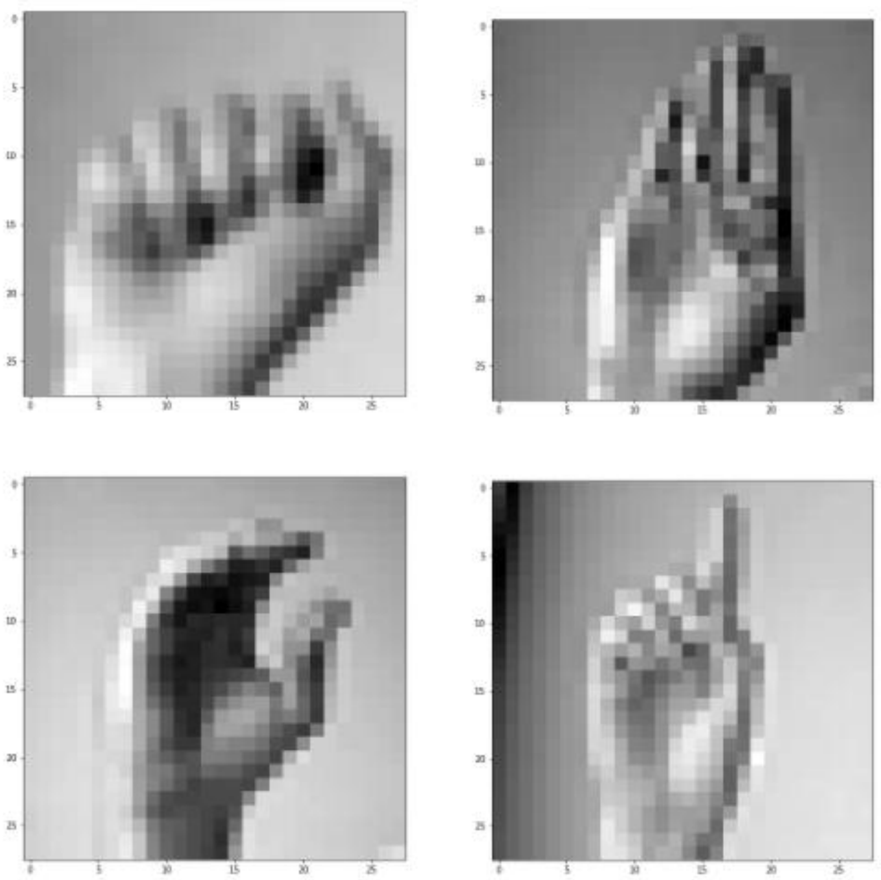


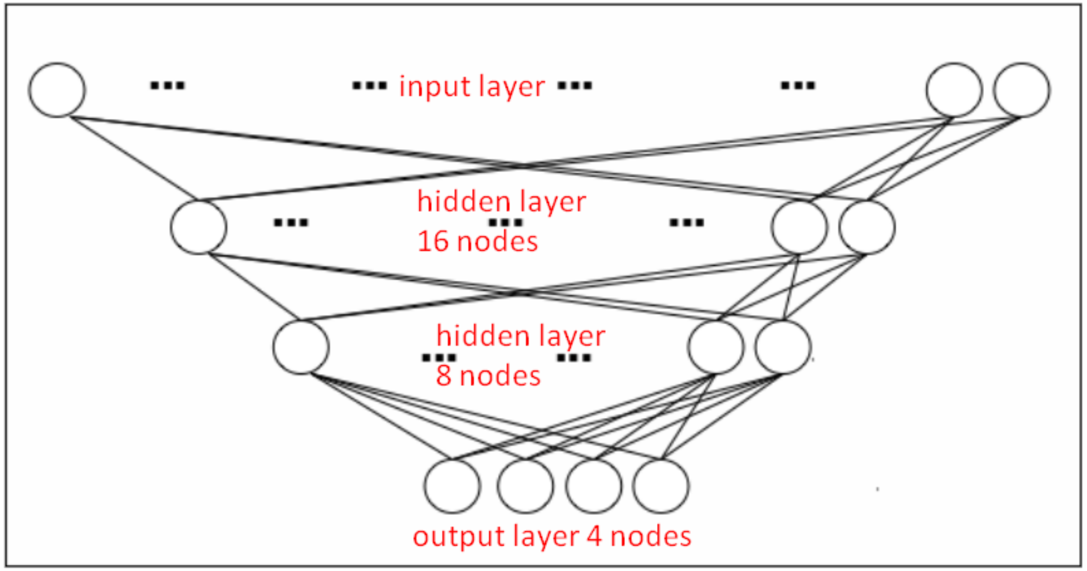
Chapter 4. High Level APIs

In the final chapter, you'll use high-level APIs in TensorFlow 2 to train a sign language letter classifier. You will use both the sequential and functional Keras APIs to train, validate, make predictions with, and evaluate models. You will also learn how to use the Estimators API to streamline the model definition and training process, and to avoid errors.

Defining neural networks with Keras

Using Keras to classify 4 letters from the Sign Language MNIST dataset: a, b, c, d.

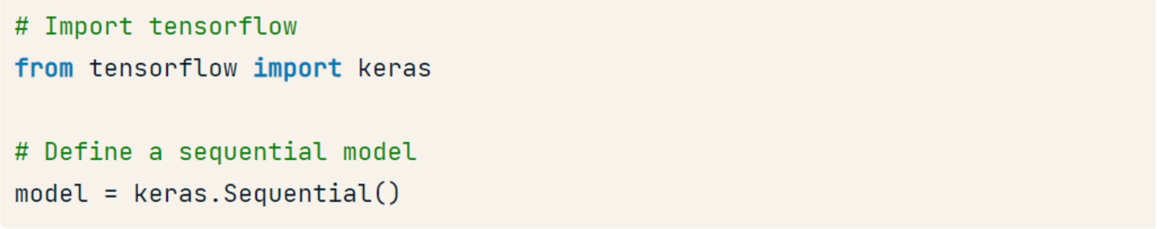




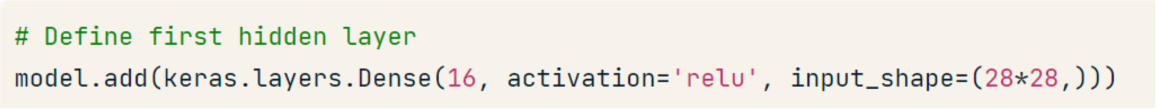
28x28 matrix

Sequential API

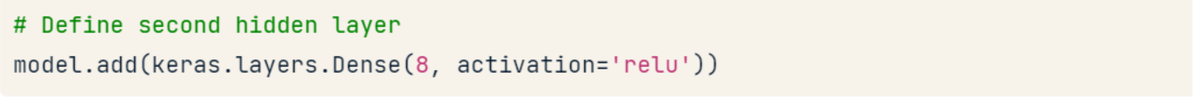
- Ordered in sequence : input layer, hidden layer, output layer



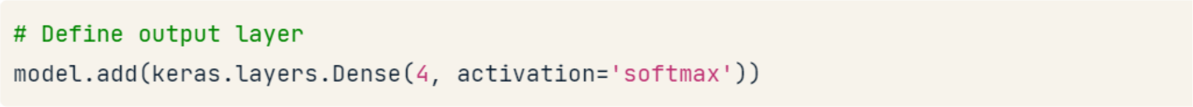
```python
# Import tensorflow
from tensorflow import keras

# Define a sequential model
model = keras.Sequential()
```



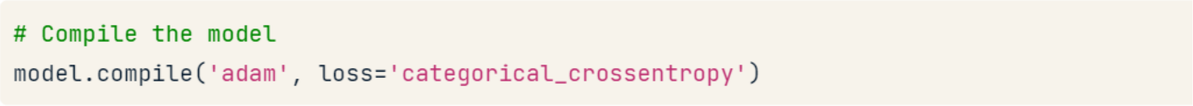
```python
# Define first hidden layer
model.add(keras.layers.Dense(16, activation='relu', input_shape=(28*28,)))
```



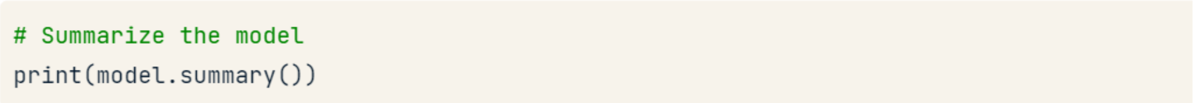
```python
# Define second hidden layer
model.add(keras.layers.Dense(8, activation='relu'))
```



```python
# Define output layer
model.add(keras.layers.Dense(4, activation='softmax'))
```



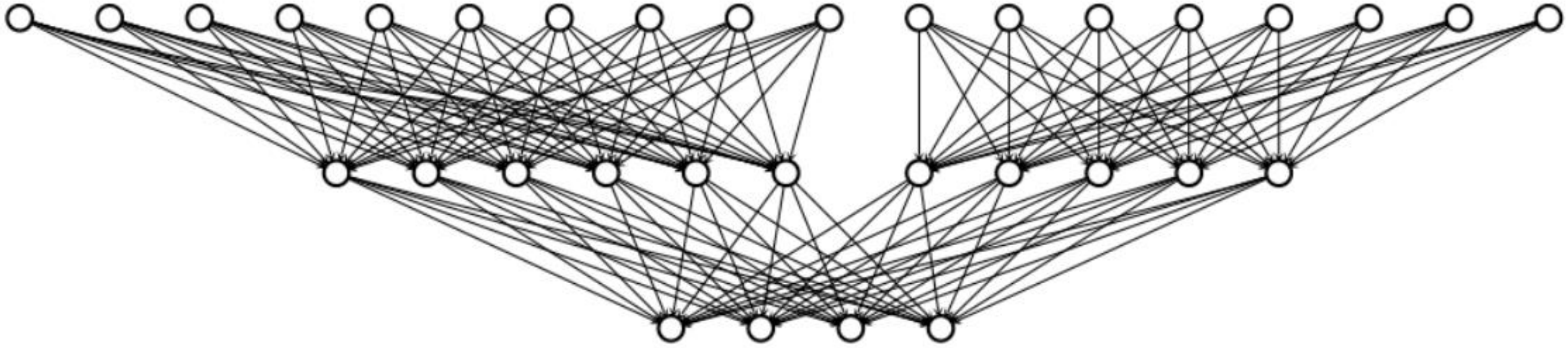
```python
# Compile the model
model.compile('adam', loss='categorical_crossentropy')
```

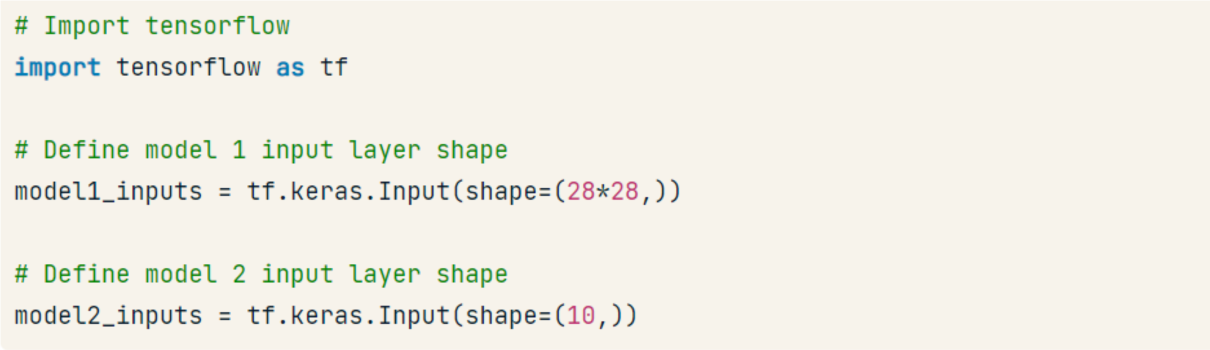


```python
# Summarize the model
print(model.summary())
```

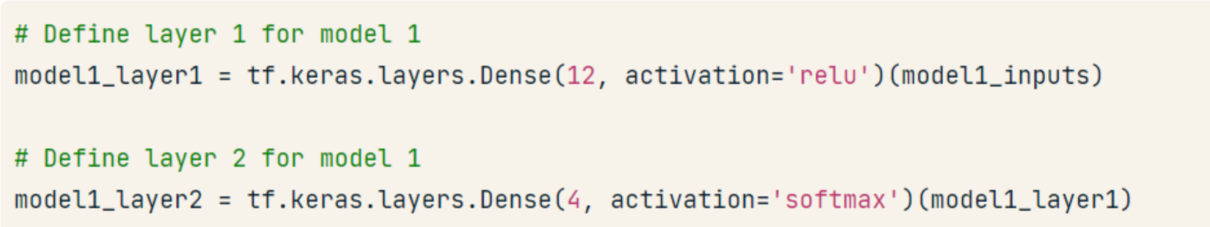
Functional API

- Train 2 model s jointly to predict the same target



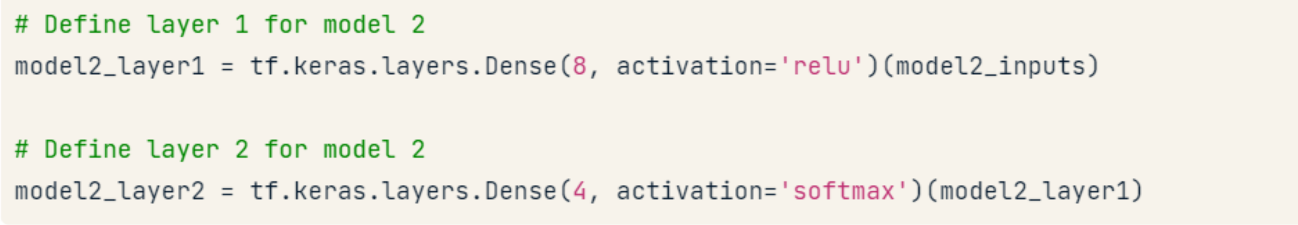


```
# Import tensorflow
import tensorflow as tf

# Define model 1 input layer shape
model1_inputs = tf.keras.Input(shape=(28*28,))

# Define model 2 input layer shape
model2_inputs = tf.keras.Input(shape=(10,))
```

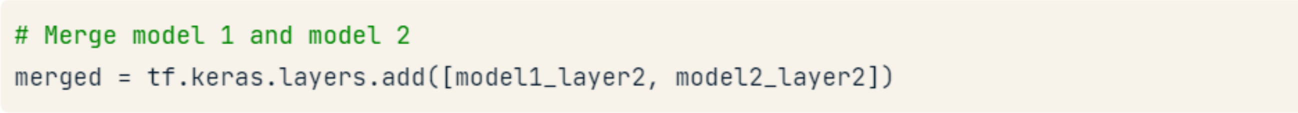


```
# Define layer 1 for model 1
model1_layer1 = tf.keras.layers.Dense(12, activation='relu')(model1_inputs)

# Define layer 2 for model 1
model1_layer2 = tf.keras.layers.Dense(4, activation='softmax')(model1_layer1)
```

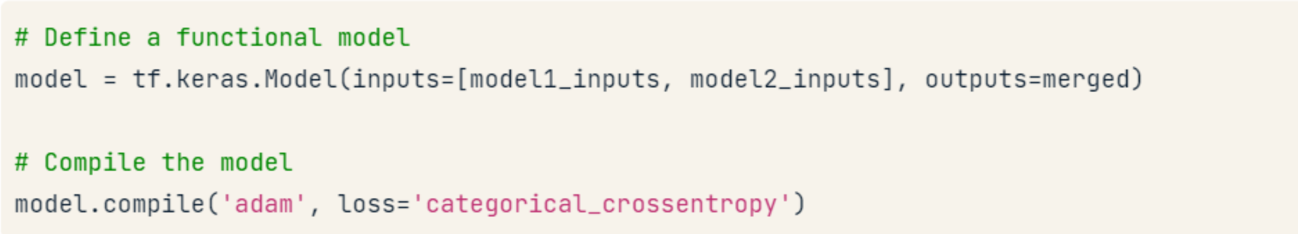
Note: to pass previous layer as an argument



```
# Define layer 1 for model 2
model2_layer1 = tf.keras.layers.Dense(8, activation='relu')(model2_inputs)

# Define layer 2 for model 2
model2_layer2 = tf.keras.layers.Dense(4, activation='softmax')(model2_layer1)
```



```
# Merge model 1 and model 2
merged = tf.keras.layers.add([model1_layer2, model2_layer2])
```



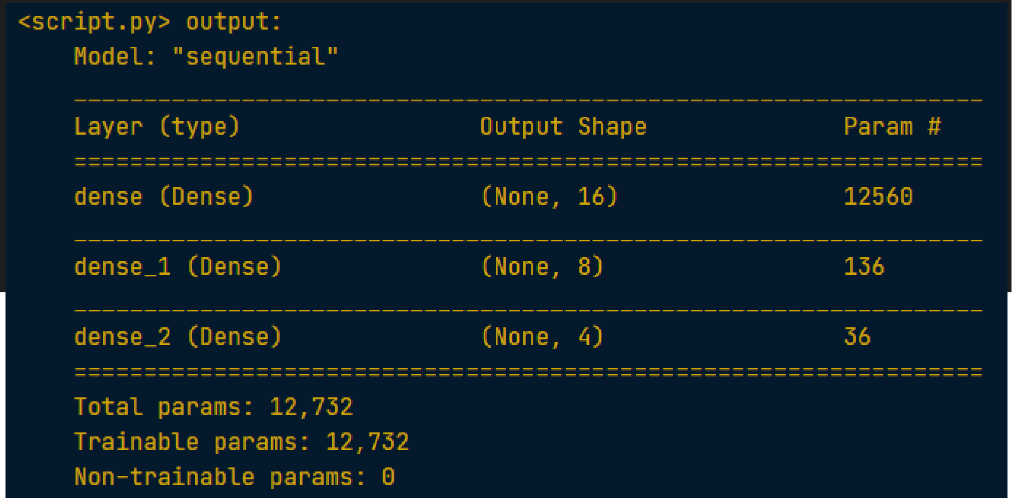
```
# Define a functional model
model = tf.keras.Model(inputs=[model1_inputs, model2_inputs], outputs=merged)

# Compile the model
model.compile('adam', loss='categorical_crossentropy')
```

The sequential model in Keras

In chapter 3, we used components of the `keras` API in `tensorflow` to define a neural network, but we stopped short of using its full capabilities to streamline model definition and training. In this exercise, you will use the `keras` sequential model API to define a neural network that can be used to classify images of sign language letters. You will also use the `.summary()` method to print the model's architecture, including the shape and number of parameters associated with each layer.

Note that the images were reshaped from (28, 28) to (784,), so that they could be used as inputs to a dense layer. Additionally, note that `keras` has been imported from `tensorflow` for you.

```
# Define a Keras sequential model
model = keras.Sequential()

# Define the first dense layer
model.add(keras.layers.Dense(16, activation='relu', input_shape=(784,)))

# Define the second dense layer
model.add(keras.layers.Dense(8, activation='relu'))

# Define the output layer
model.add(keras.layers.Dense(4, activation='softmax'))

# Print the model architectureprint(model.summary())
```



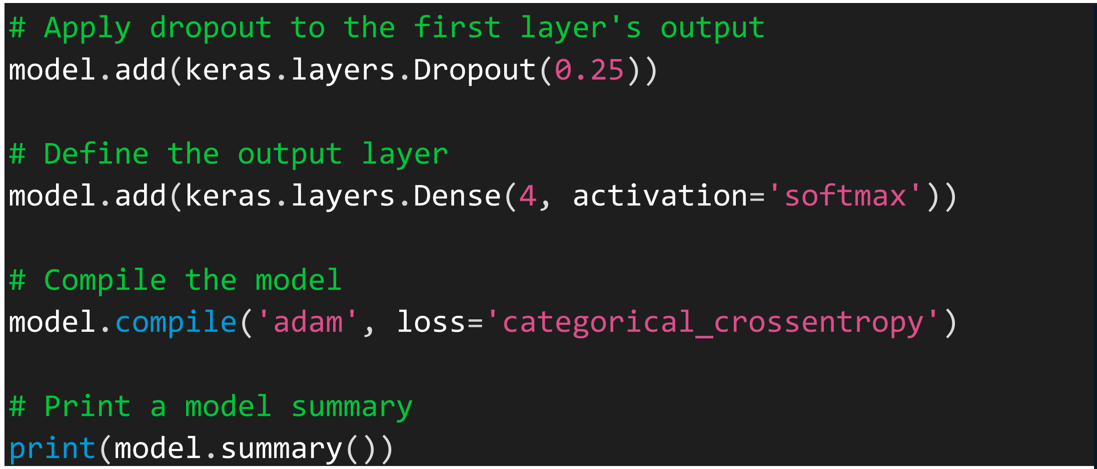
```
<script.py> output:
    Model: "sequential"
    
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #   
    =================================================================
    dense (Dense)                (None, 16)                12560     
    
    _________________________________________________________________
    dense_1 (Dense)              (None, 8)                 136       
    
    _________________________________________________________________
    dense_2 (Dense)              (None, 4)                 36        
    =================================================================
    Total params: 12,732
    Trainable params: 12,732
    Non-trainable params: 0
```

Notice that we've defined a model, but we haven't compiled it. The compilation step in `keras` allows us to set the optimizer, loss function, and other useful training parameters in a single line of code. Furthermore, the `.summary()` method allows us to view the model's architecture.

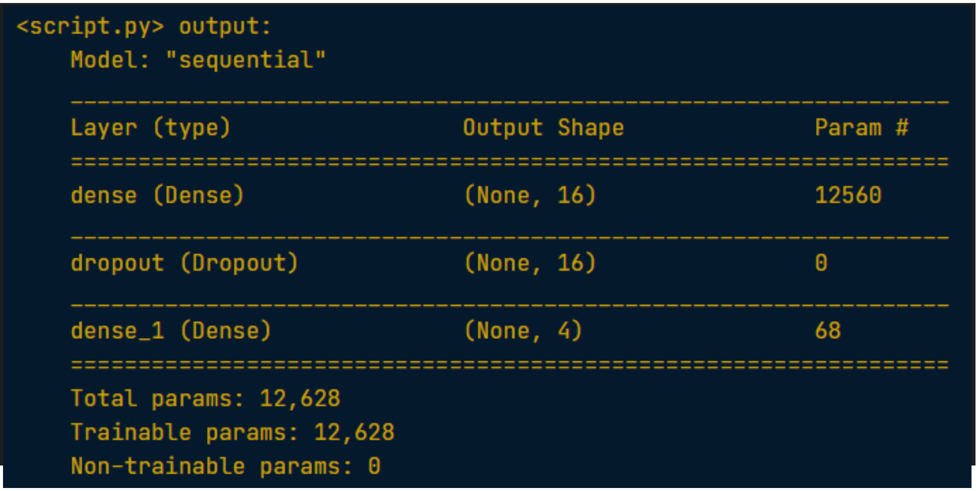
Compiling a sequential model

In this exercise, you will work towards classifying letters from the Sign Language MNIST dataset; however, you will adopt a more simple architecture. There will be fewer layers, but more nodes. You will also apply dropout to prevent overfitting. Finally, you will compile the model to use the `adam` optimizer and the `categorical_crossentropy` loss. You will also use a method in `keras` to summarize your model's architecture. Note that `keras` has been imported from `tensorflow` for you and a sequential `keras` model has been defined as `model`.

```
# Define the first dense layer
model.add(keras.layers.Dense(16, activation='sigmoid', input_shape=(784,)))
```



```
# Apply dropout to the first layer's output
model.add(keras.layers.Dropout(0.25))

# Define the output layer
model.add(keras.layers.Dense(4, activation='softmax'))

# Compile the model
model.compile('adam', loss='categorical_crossentropy')

# Print a model summary
print(model.summary())
```



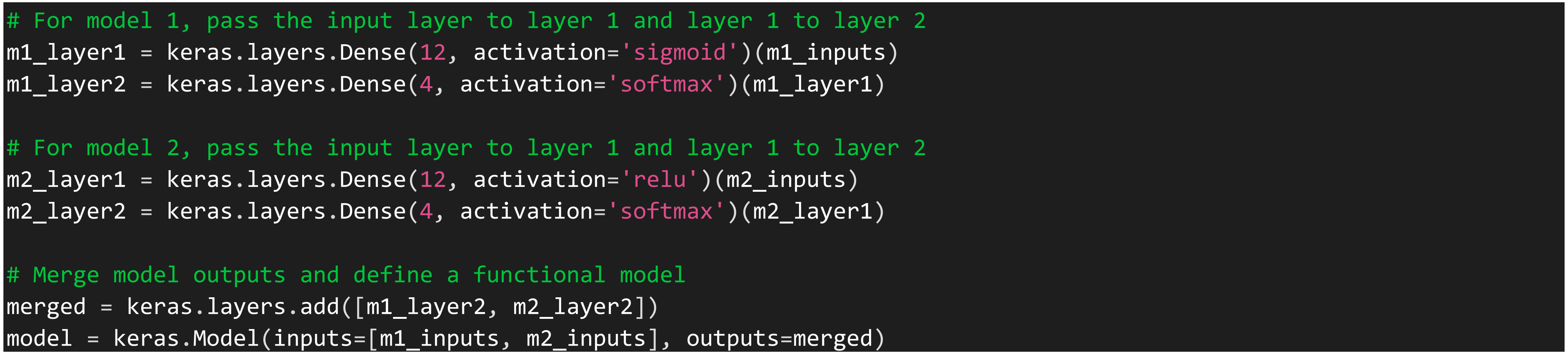
```
<script.py> output:
    Model: "sequential"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense (Dense)                (None, 16)                12560
    _________________________________________________________________
    dropout (Dropout)            (None, 16)                0
    _________________________________________________________________
    dense_1 (Dense)              (None, 4)                 68
    =================================================================
    Total params: 12,628
    Trainable params: 12,628
    Non-trainable params: 0
```

You've now defined and compiled a neural network using the `keras` sequential model. Notice that printing the `.summary()` method shows the layer type, output shape, and number of parameters of each layer.

Defining a multiple input model

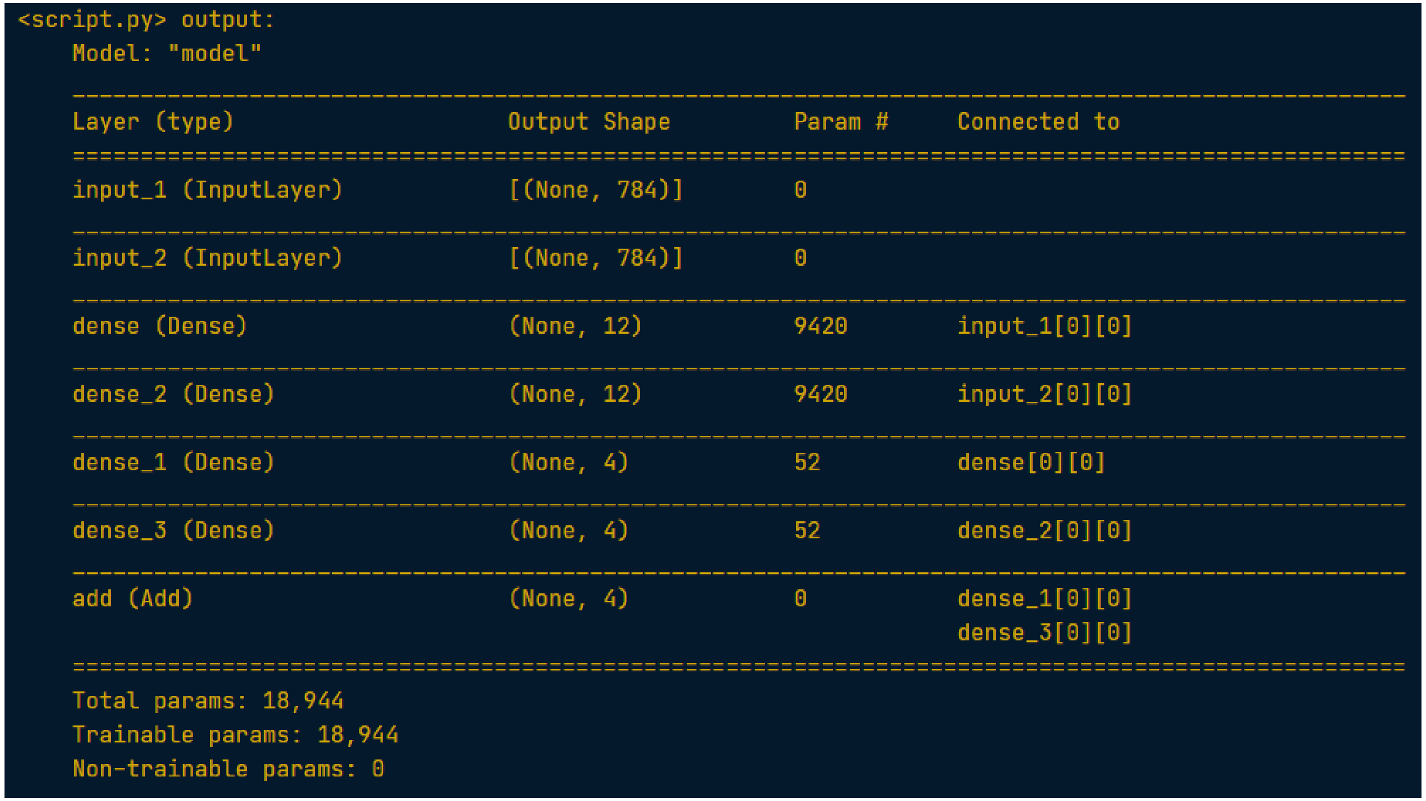
In some cases, the sequential API will not be sufficiently flexible to accommodate your desired model architecture and you will need to use the functional API instead. If, for instance, you want to train two models with different architectures jointly, you will need to use the functional API to do this. In this exercise, we will see how to do this. We will also use the `.summary()` method to examine the joint model's architecture.

Note that `keras` has been imported from `tensorflow` for you. Additionally, the input layers of the first and second models have been defined as `m1_inputs` and `m2_inputs`, respectively. Note that the two models have the same architecture, but one of them uses a `sigmoid` activation in the first layer and the other uses a `relu`.



```
# For model 1, pass the input layer to layer 1 and layer 1 to layer 2
m1_layer1 = keras.layers.Dense(12, activation='sigmoid')(m1_inputs)
m1_layer2 = keras.layers.Dense(4, activation='softmax')(m1_layer1)

# For model 2, pass the input layer to layer 1 and layer 1 to layer 2
m2_layer1 = keras.layers.Dense(12, activation='relu')(m2_inputs)
m2_layer2 = keras.layers.Dense(4, activation='softmax')(m2_layer1)

# Merge model outputs and define a functional model
merged = keras.layers.add([m1_layer2, m2_layer2])
model = keras.Model(inputs=[m1_inputs, m2_inputs], outputs=merged)
```

```
# Print a model summary
print(model.summary())
```



```
<script.py> output:
    Model: "model"
    __________________________________________________________________________________________________
    Layer (type)                    Output Shape         Param #     Connected to
    ==================================================================================================
    input_1 (InputLayer)            [(None, 784)]        0
    __________________________________________________________________________________________________
    input_2 (InputLayer)            [(None, 784)]        0
    __________________________________________________________________________________________________
    dense (Dense)                   (None, 12)           9420        input_1[0][0]
    __________________________________________________________________________________________________
    dense_2 (Dense)                 (None, 12)           9420        input_2[0][0]
    __________________________________________________________________________________________________
    dense_1 (Dense)                 (None, 4)            52          dense[0][0]
    __________________________________________________________________________________________________
    dense_3 (Dense)                 (None, 4)            52          dense_2[0][0]
    __________________________________________________________________________________________________
    add (Add)                       (None, 4)            0           dense_1[0][0]
                                                                     dense_3[0][0]
    ==================================================================================================
    Total params: 18,944
    Trainable params: 18,944
    Non-trainable params: 0
```

Notice that the `.summary()` method yields a new column: `connected to`. This column tells you how layers connect to each other within the network. We can see that `dense_2`, for instance, is connected to the `input_2` layer. We can also see that the `add` layer, which merged the two models, connected to both `dense_1` and `dense_3`.

Training and validation with Keras

Overview

- Load and clean data
- Define model
- Train and validate model
- Evaluate model

```
# Import tensorflow
import tensorflow as tf

# Define a sequential model
model = tf.keras.Sequential()
```

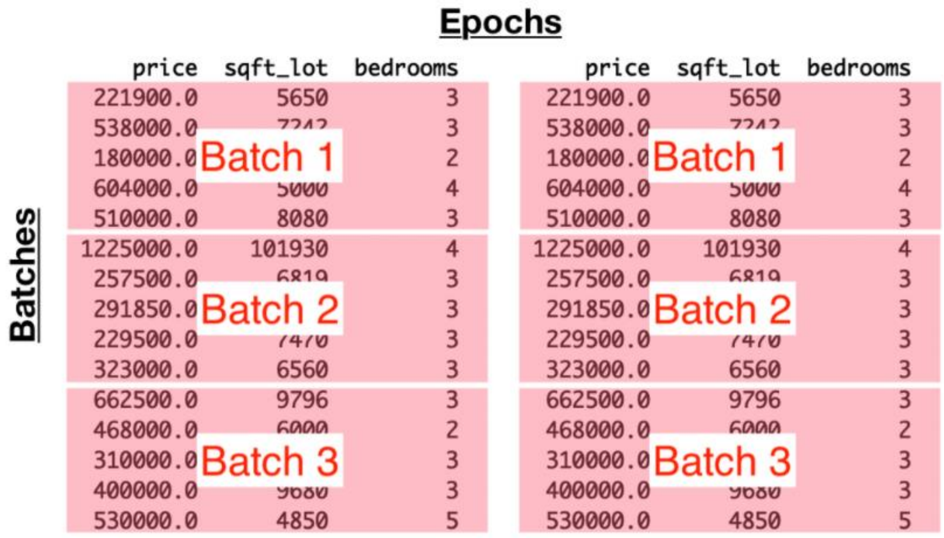
```
# Define the hidden layer
model.add(tf.keras.layers.Dense(16, activation='relu', input_shape=(784,)))
```

```
# Define the output layer
model.add(tf.keras.layers.Dense(4, activation='softmax'))
```

```
# Compile model
model.compile('adam', loss='categorical_crossentropy')
```

```
# Train model
model.fit(image_features, image_labels)
```

Batch size of 5, and 2 epochs

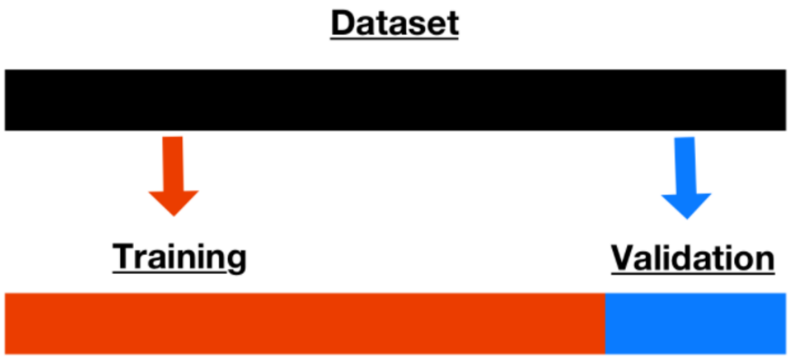

Epochs
Batches
Batch 1
Batch 2
Batch 3
Batch 1
Batch 2
Batch 3

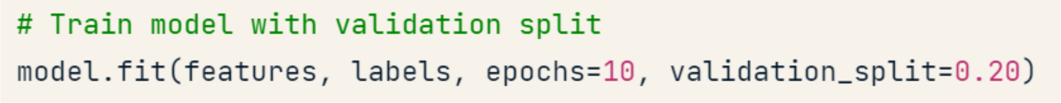
price	sqft_lot	bedrooms
221900.0	5650	3
538000.0	7242	3
180000.0	[illegible]	2
604000.0	5000	4
510000.0	8080	3
1225000.0	101930	4
257500.0	6819	3
291850.0	[illegible]	3
229500.0	7470	3
323000.0	6560	3
662500.0	9796	3
468000.0	6000	2
310000.0	[illegible]	3
400000.0	9680	3
530000.0	4850	5

price	sqft_lot	bedrooms
221900.0	5650	3
538000.0	7242	3
180000.0	[illegible]	2
604000.0	5000	4
510000.0	8080	3
1225000.0	101930	4
257500.0	6819	3
291850.0	[illegible]	3
229500.0	7470	3
323000.0	6560	3
662500.0	9796	3
468000.0	6000	2
310000.0	[illegible]	3
400000.0	9680	3
530000.0	4850	5

Using multiple epochs allows the model to revisit the same batches, but with different model weights and possibly optimizer parameters, since they are updated after each batch.

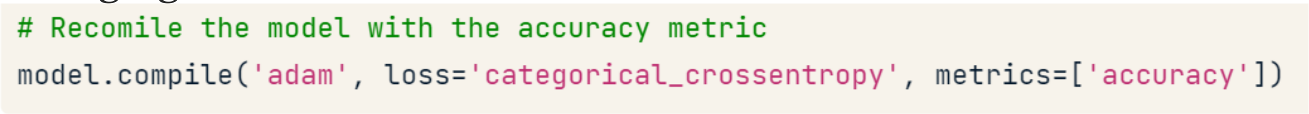
Performing validation



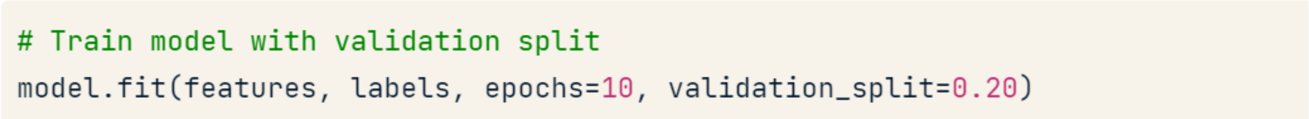


```
# Train model with validation split
model.fit(features, labels, epochs=10, validation_split=0.20)
```

Changing the metric

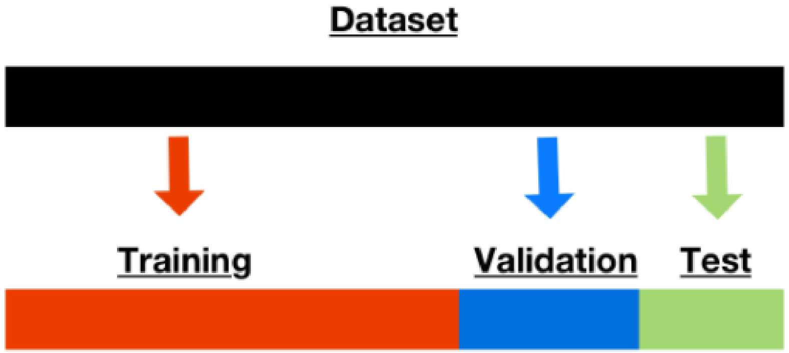


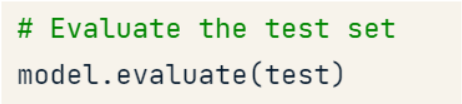
```
# Recomile the model with the accuracy metric
model.compile('adam', loss='categorical_crossentropy', metrics=['accuracy'])
```



```
# Train model with validation split
model.fit(features, labels, epochs=10, validation_split=0.20)
```

Use the evaluation() operation to check performance on the test set at the end of the training process





```
# Evaluate the test set
model.evaluate(test)
```

Training with Keras

In this exercise, we return to our sign language letter classification problem. We have 2000 images of four letters--A, B, C, and D--and we want to classify them with a high level of accuracy. We will complete all parts of the problem, including the model definition, compilation, and training.

Note that `keras` has been imported from `tensorflow` for you. Additionally, the features are available as `sign_language_features` and the targets are available as `sign_language_labels`.

```
# Define a sequential model
model = keras.Sequential()

# Define a hidden layer
model.add(keras.layers.Dense(16, activation='relu', input_shape=(784,)))

# Define the output layer
model.add(keras.layers.Dense(4, activation='softmax'))

# Compile the model
model.compile('SGD', loss='categorical_crossentropy')

# Complete the fitting operation
model.fit(sign_language_features, sign_language_labels, epochs=5)
```

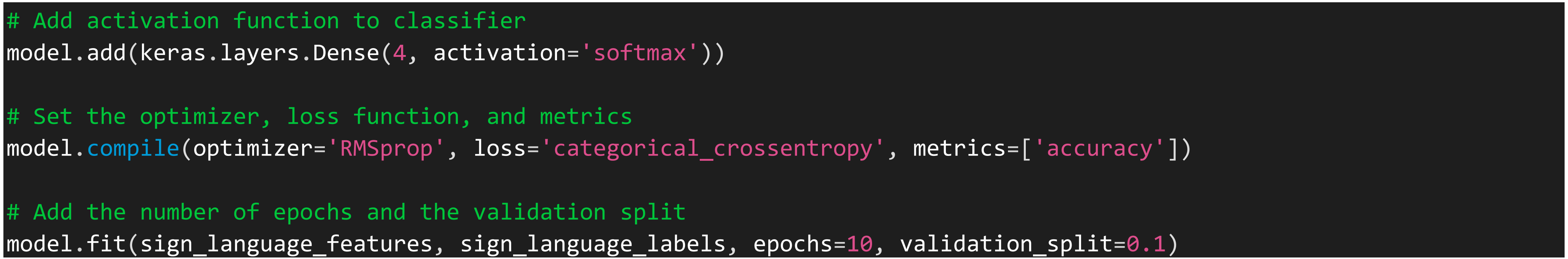
You probably noticed that your only measure of performance improvement was the value of the loss function in the training sam ple, which is not particularly informative. You will improve on this in the next exercise.

Metrics and validation with Keras

We trained a model to predict sign language letters in the previous exercise, but it is unclear how successful we were in doing so. In this exercise, we will try to improve upon the interpretability of our results. Since we did not use a validation split, we only observed performance improvements within the training set; however, it is unclear how much of that was due to overfitting. Furthermore, since we did not supply a metric, we only saw decreases in the loss function, which do not have any clear interpretation.

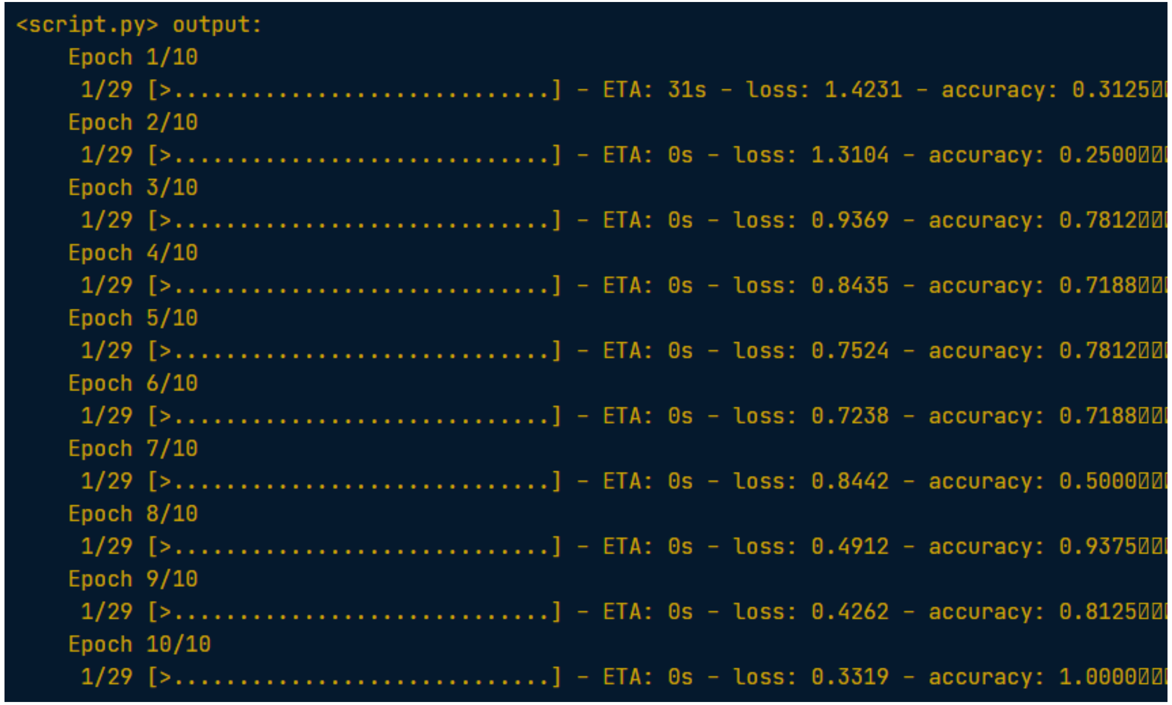
Note that `keras` has been imported for you from `tensorflow`.

```
# Define sequential model
model = keras.Sequential()

# Define the first layer
model.add(keras.layers.Dense(32, activation='sigmoid', input_shape=(784,)))
```



```
# Add activation function to classifier
model.add(keras.layers.Dense(4, activation='softmax'))

# Set the optimizer, loss function, and metrics
model.compile(optimizer='RMSprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Add the number of epochs and the validation split
model.fit(sign_language_features, sign_language_labels, epochs=10, validation_split=0.1)
```

With the `keras` API, you only needed 14 lines of code to define, compile, train, and validate a model. You may have noticed that your model performed quite well. In just 10 epochs, we achieved a classification accuracy of over 90% in the validation sample!



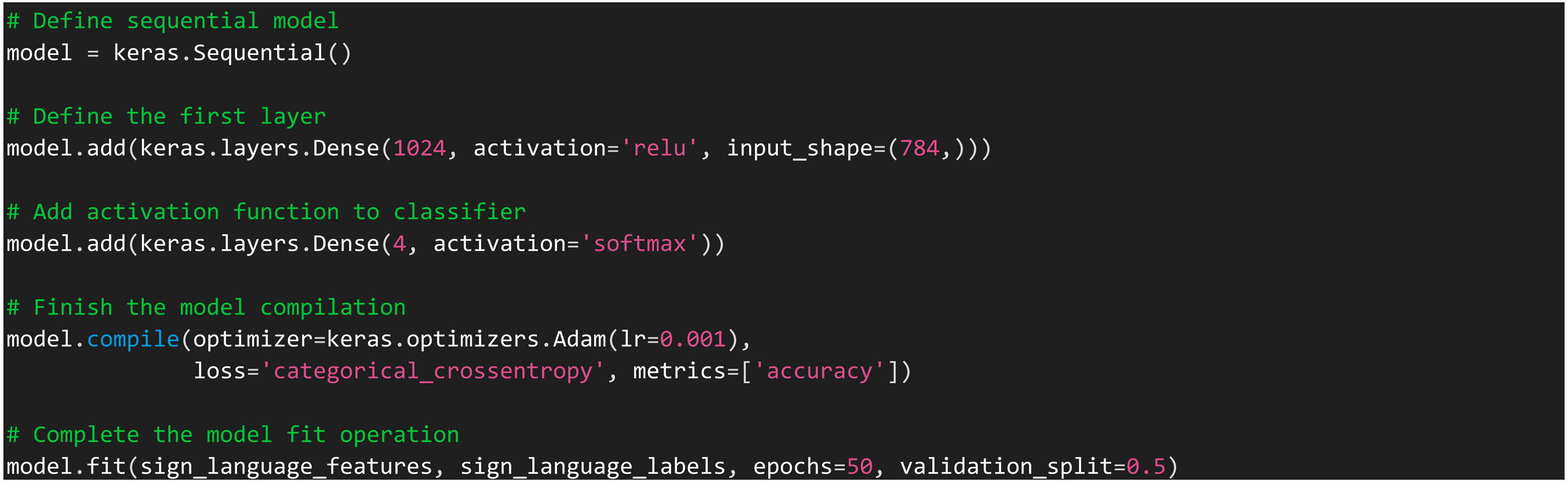
```
<script.py> output:
    Epoch 1/10
     1/29 [>.............................] - ETA: 31s - loss: 1.4231 - accuracy: 0.3125
    Epoch 2/10
     1/29 [>.............................] - ETA: 0s - loss: 1.3104 - accuracy: 0.2500
    Epoch 3/10
     1/29 [>.............................] - ETA: 0s - loss: 0.9369 - accuracy: 0.7812
    Epoch 4/10
     1/29 [>.............................] - ETA: 0s - loss: 0.8435 - accuracy: 0.7188
    Epoch 5/10
     1/29 [>.............................] - ETA: 0s - loss: 0.7524 - accuracy: 0.7812
    Epoch 6/10
     1/29 [>.............................] - ETA: 0s - loss: 0.7238 - accuracy: 0.7188
    Epoch 7/10
     1/29 [>.............................] - ETA: 0s - loss: 0.8442 - accuracy: 0.5000
    Epoch 8/10
     1/29 [>.............................] - ETA: 0s - loss: 0.4912 - accuracy: 0.9375
    Epoch 9/10
     1/29 [>.............................] - ETA: 0s - loss: 0.4262 - accuracy: 0.8125
    Epoch 10/10
     1/29 [>.............................] - ETA: 0s - loss: 0.3319 - accuracy: 1.0000
```

Overfitting detection

In this exercise, we'll work with a small subset of the examples from the original sign language letters dataset. A small sample, coupled with a heavily-parameterized model, will generally lead to overfitting. This means that your model will simply memorize the class of each example, rather than identifying features that generalize to many examples.

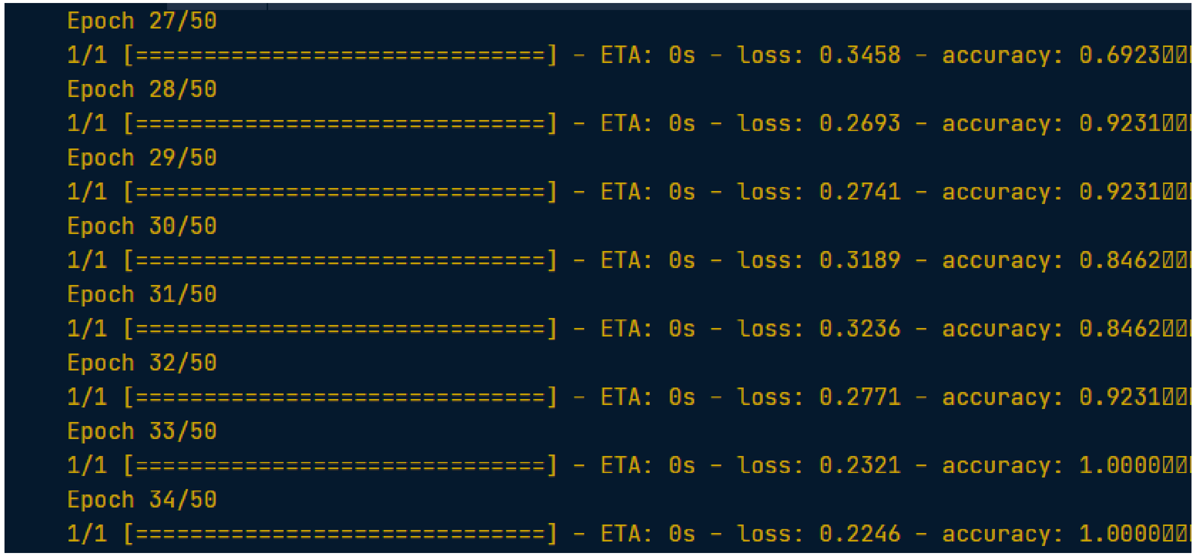
You will detect overfitting by checking whether the validation sample loss is substantially higher than the training sample loss and whether it increases with further training. With a small sample and a high learning rate, the model will struggle to converge on an optimum. You will set a low learning rate for the optimizer, which will make it easier to identify overfitting.

Note that `keras` has been imported from `tensorflow`.



```
# Define sequential model
model = keras.Sequential()

# Define the first layer
model.add(keras.layers.Dense(1024, activation='relu', input_shape=(784,)))

# Add activation function to classifier
model.add(keras.layers.Dense(4, activation='softmax'))

# Finish the model compilation
model.compile(optimizer=keras.optimizers.Adam(lr=0.001),
              loss='categorical_crossentropy', metrics=['accuracy'])

# Complete the model fit operation
model.fit(sign_language_features, sign_language_labels, epochs=50, validation_split=0.5)
```

You may have noticed that the validation loss, `val_loss`, was substantially higher than the training loss, `loss`. Furthermore, if `val_loss` started to increase before the training process was terminated, then we may have overfitted. When this happens, you will want to try decreasing the number of epochs.

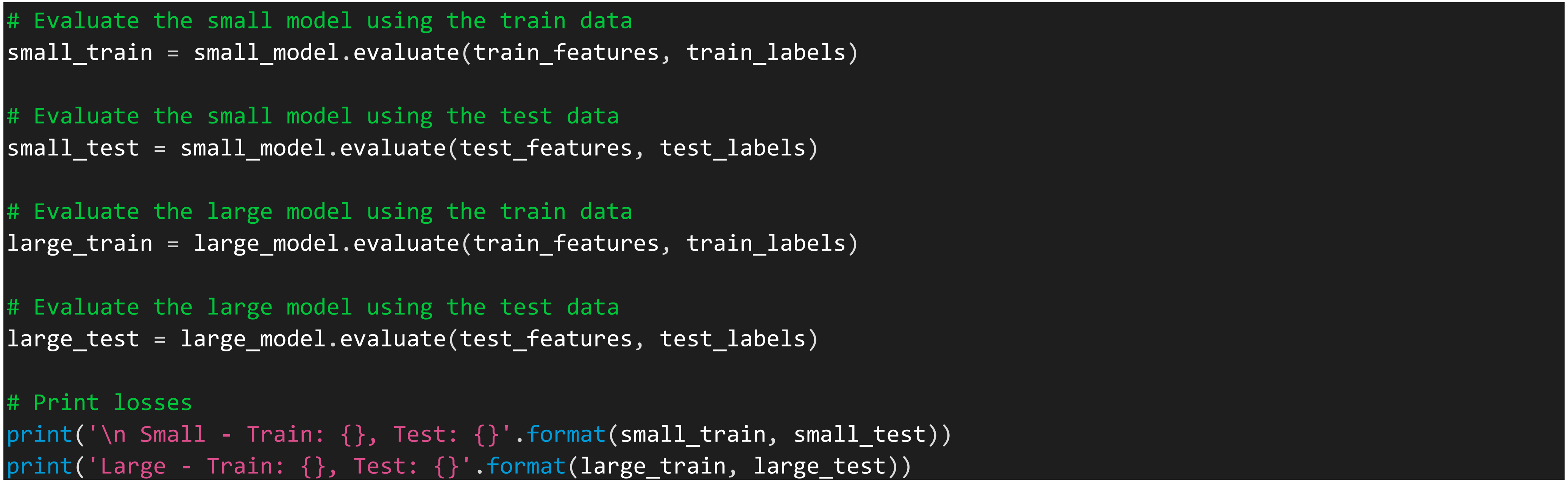


```
Epoch 27/50
1/1 [==============================] - ETA: 0s - loss: 0.3458 - accuracy: 0.6923
Epoch 28/50
1/1 [==============================] - ETA: 0s - loss: 0.2693 - accuracy: 0.9231
Epoch 29/50
1/1 [==============================] - ETA: 0s - loss: 0.2741 - accuracy: 0.9231
Epoch 30/50
1/1 [==============================] - ETA: 0s - loss: 0.3189 - accuracy: 0.8462
Epoch 31/50
1/1 [==============================] - ETA: 0s - loss: 0.3236 - accuracy: 0.8462
Epoch 32/50
1/1 [==============================] - ETA: 0s - loss: 0.2771 - accuracy: 0.9231
Epoch 33/50
1/1 [==============================] - ETA: 0s - loss: 0.2321 - accuracy: 1.0000
Epoch 34/50
1/1 [==============================] - ETA: 0s - loss: 0.2246 - accuracy: 1.0000
```

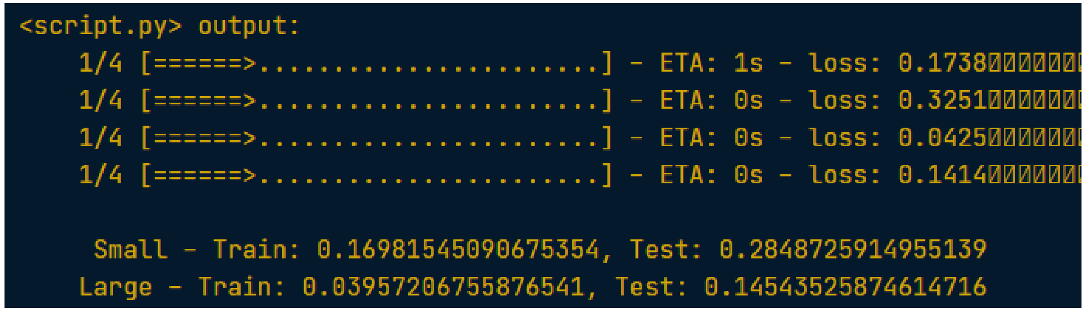
Evaluating models

Two models have been trained and are available: `large_model`, which has many parameters; and `small_model`, which has fewer parameters. Both models have been trained using `train_features` and `train_labels`, which are available to you. A separate test set, which consists of `test_features` and `test_labels`, is also available.

Your goal is to evaluate relative model performance and also determine whether either model exhibits signs of overfitting. You will do this by evaluating `large_model` and `small_model` on both the train and test sets. For each model, you can do this by applying the `.evaluate(x, y)` method to compute the loss for features `x` and labels `y`. You will then compare the four losses generated.



```python
# Evaluate the small model using the train data
small_train = small_model.evaluate(train_features, train_labels)

# Evaluate the small model using the test data
small_test = small_model.evaluate(test_features, test_labels)

# Evaluate the large model using the train data
large_train = large_model.evaluate(train_features, train_labels)

# Evaluate the large model using the test data
large_test = large_model.evaluate(test_features, test_labels)

# Print losses
print('\n Small - Train: {}, Test: {}'.format(small_train, small_test))
print('Large - Train: {}, Test: {}'.format(large_train, large_test))
```

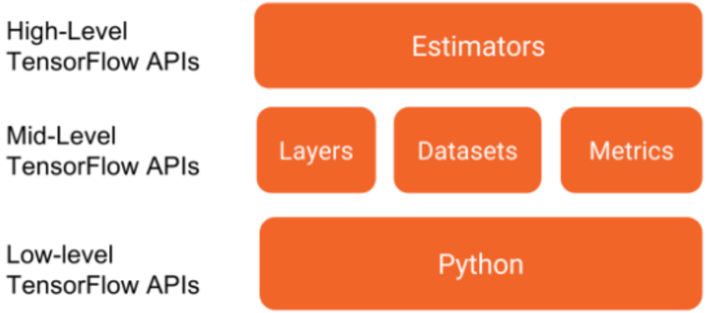


```
<script.py> output:
    1/4 [======>.......................] - ETA: 1s - loss: 0.1738▯▯▯▯▯▯▯
    1/4 [======>.......................] - ETA: 0s - loss: 0.3251▯▯▯▯▯▯▯
    1/4 [======>.......................] - ETA: 0s - loss: 0.0425▯▯▯▯▯▯▯
    1/4 [======>.......................] - ETA: 0s - loss: 0.1414▯▯▯▯▯▯▯

     Small - Train: 0.16981545090675354, Test: 0.2848725914955139
    Large - Train: 0.03957206755876541, Test: 0.14543525874614716
```

Notice that the gap between the test and train set losses is high for `large_model`, suggesting that overfitting may be an issue. Furthermore, both test and train set performance is better for `large_model`. This suggests that we may want to use `large_model`, but reduce the number of training epochs.

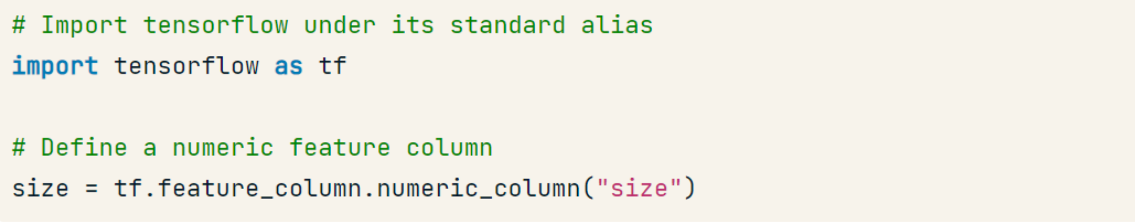
Training models with the Estimators API

- High level submodule
- Less flexible
- Enforces best practices
- Faster deployment
- Many pre-made models

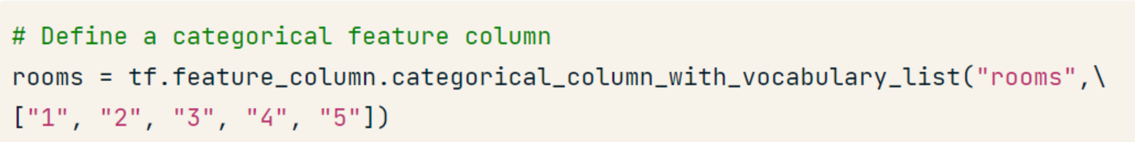

High-Level TensorFlow APIs
Estimators
Mid-Level TensorFlow APIs
Layers
Datasets
Metrics
Low-level TensorFlow APIs
Python

Model specifiation and training

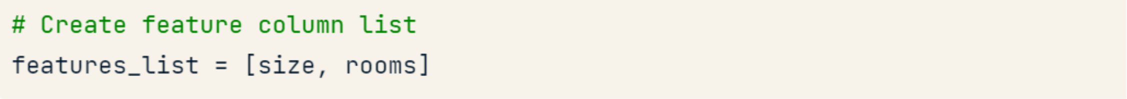
1. Define feature columns
2. Load and transform data
3. Define an estimator, eg pre-made or custom estimator
4. Apply train operation



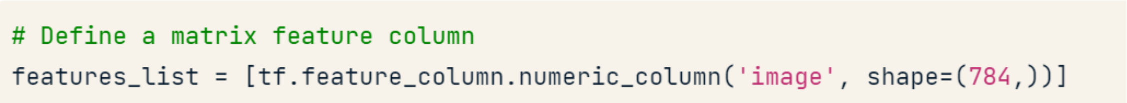
```python
# Import tensorflow under its standard alias
import tensorflow as tf


# Define a numeric feature column
size = tf.feature_column.numeric_column("size")
```



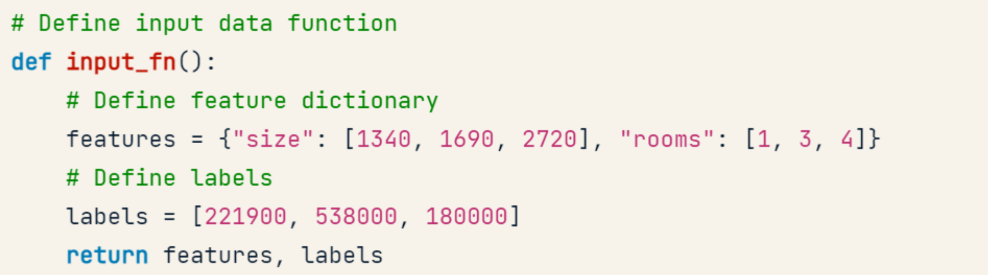
```python
# Define a categorical feature column
rooms = tf.feature_column.categorical_column_with_vocabulary_list("rooms",\
["1", "2", "3", "4", "5"])
```



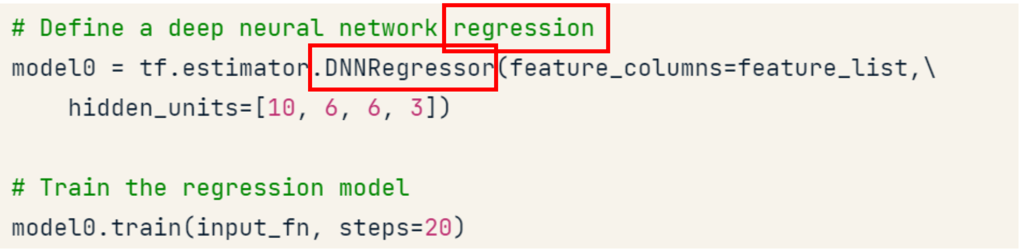
```python
# Create feature column list
features_list = [size, rooms]
```



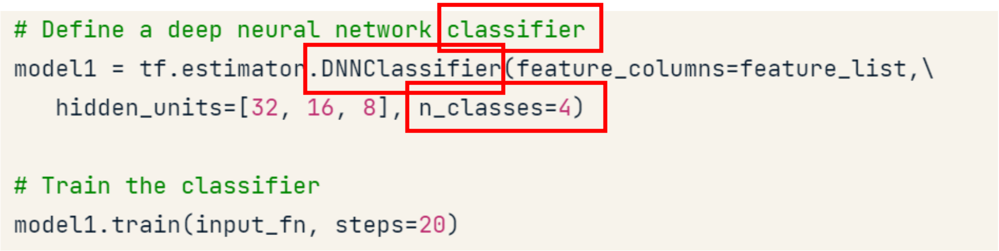
```python
# Define a matrix feature column
features_list = [tf.feature_column.numeric_column('image', shape=(784,))]
```



```python
# Define input data function
def input_fn():
    # Define feature dictionary
    features = {"size": [1340, 1690, 2720], "rooms": [1, 3, 4]}
    # Define labels
    labels = [221900, 538000, 180000]
    return features, labels
```



```python
# Define a deep neural network regression
model0 = tf.estimator.DNNRegressor(feature_columns=feature_list,\
    hidden_units=[10, 6, 6, 3])


# Train the regression model
model0.train(input_fn, steps=20)
```



```python
# Define a deep neural network classifier
model1 = tf.estimator.DNNClassifier(feature_columns=feature_list,\
   hidden_units=[32, 16, 8], n_classes=4)


# Train the classifier
model1.train(input_fn, steps=20)
```

Preparing to train with Estimators

For this exercise, we'll return to the King County housing transaction dataset from chapter 2. We will again develop and train a machine learning model to predict house prices; however, this time, we'll do it using the `estimator` API.

Rather than completing everything in one step, we'll break this procedure down into parts. We'll begin by defining the feature columns and loading the data. In the next exercise, we'll define and train a premade `estimator`. Note that `feature_column` has been imported for you from `tensorflow`. Additionally, `numpy` has been imported as `np`, and the Kings County housing dataset is available as a `pandas` `DataFrame`: `housing`.

```
# Define feature columns for bedrooms and bathrooms
bedrooms = feature_column.numeric_column("bedrooms")
bathrooms = feature_column.numeric_column("bathrooms")

# Define the list of feature columns
feature_list = [bedrooms, bathrooms]

def input_fn():
    # Define the labels
    labels = np.array(housing['price'])
    # Define the features
    features = {'bedrooms':np.array(housing['bedrooms']),
                'bathrooms':np.array(housing['bathrooms'])}
    return features, labels
```

In the next exercise, we'll use the feature columns and data input function to define and train an estimator.

Defining Estimators

In the previous exercise, you defined a list of feature columns, `feature_list`, and a data input function, `input_fn()`. In this exercise, you will build on that work by defining an `estimator` that makes use of input data.

```
# Define the model and set the number of steps
model = estimator.DNNRegressor(feature_columns=feature_list, hidden_units=[2,2])
model.train(input_fn, steps=1)
```

```
# Define the model and set the number of steps
model = estimator.LinearRegressor(feature_columns=feature_list)
model.train(input_fn, steps=2)
```

Note that you have other premade `estimator` options, such as `BoostedTreesRegressor()`, and can also create your own custom estimators.

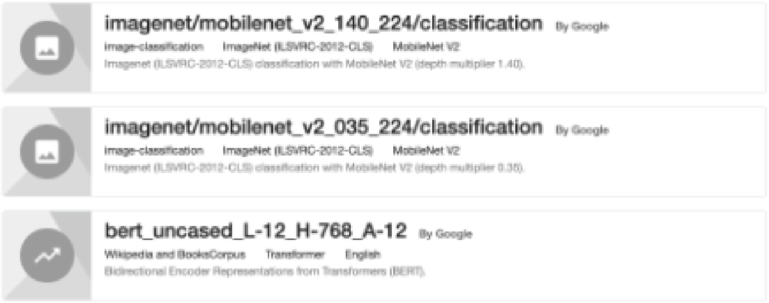
Course completed!

Recap topics covered:

- Low-level, basic, and advanced operations: define and manipulate variables and costants
- Graph-based computational model that underlies TensorFlow
- Gradient computation and solve arbitrary optimization problems
- Data loading and transformation
- Pre-defined and custom loss functions
- Linear models and batch training
- Define dense neural network layers using
    - low-level linear algebra operations
    - high-level Keras API operations
- Activation functions
- Optimization algorithms
- Training neural networks
- Neural networks in Keras
- Training and validation
- High-level Estimators API which can be used to streamline the production process

Next steps are to explore TensorFlow extensions:

- TensorFlow Hub
  - Pretrained models
  - Transfer learning



imagenet/mobilenet_v2_140_224/classification By Google

image-classification ImageNet (ILSVRC-2012-CLS) MobileNet V2

Imagenet (ILSVRC-2012-CLS) classification with MobileNet V2 (depth multiplier 1.40).

imagenet/mobilenet_v2_035_224/classification By Google

image-classification ImageNet (ILSVRC-2012-CLS) MobileNet V2

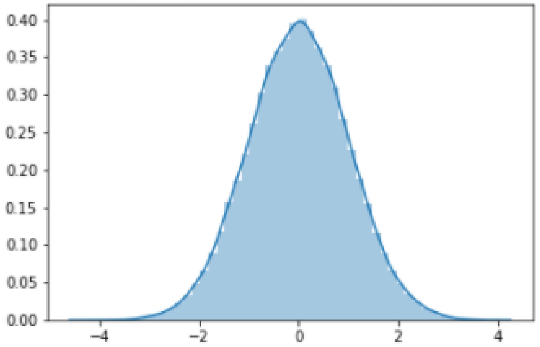
Imagenet (ILSVRC-2012-CLS) classification with MobileNet V2 (depth multiplier 0.35).

bert_uncased_L-12_H-768_A-12 By Google

Wikipedia and BooksCorpus Transformer English

Bidirectional Encoder Representations from Transformers (BERT).

- TensorFlow Probability
  - More statistical distributions
  - Trainable distributions
  - Extended set of optimizers

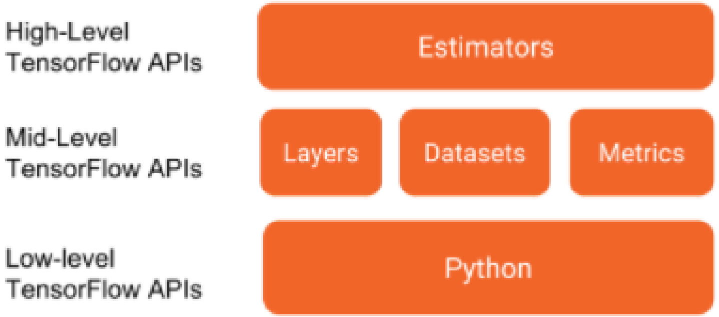


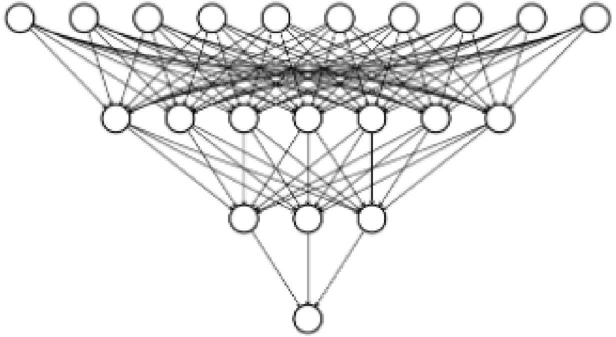
Difference between TensorFlow 2 and TensorFlow 1

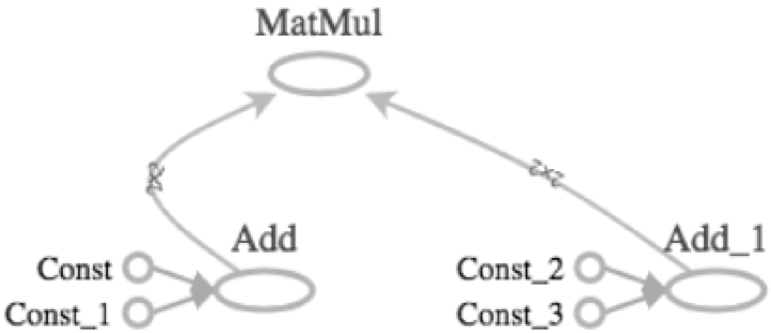
- If you primarily develop in 1, you may have noticed that you do not need to define static graphs or enable eager execution. This is done automatically in 2.
- Furthermore, TensorFlow 2 has substantially tighter integration with Keras. In fact, the core functionality of the TensorFlow 1 train module is handled by tf.Keras operations in 2.
- In addition to the centrality of Keras, the Estimators API also plays a more important role in TensorFlow 2.
- Finally, TensorFlow 2 also allows you to use static graphs, but they are available through the tf.function operation.

TensorFlow 2.0

- TensorFlow 2.0
  - `eager_execution()`
  - Tighter `keras` integration
  - `Estimators`
  - `function()`


High-Level TensorFlow APIs
Estimators
Mid-Level TensorFlow APIs
Layers
Datasets
Metrics
Low-level TensorFlow APIs
Python





Screenshot taken from https://www.tensorflow.org/guide/premade_estimators

---

Happy learning!