When `eta` is 0.1, AUC grows more slowly but peaks at a higher value. For a smaller learning rate, it takes more trees to reach the peak, but we could achieve better performance.

For comparison, we can also try other values of `eta` (figure 6.34):

- For 0.05, the best AUC is 82.2% (after 120 iterations).
- For 0.01, the best AUC is 82.1% (after 500 iterations).

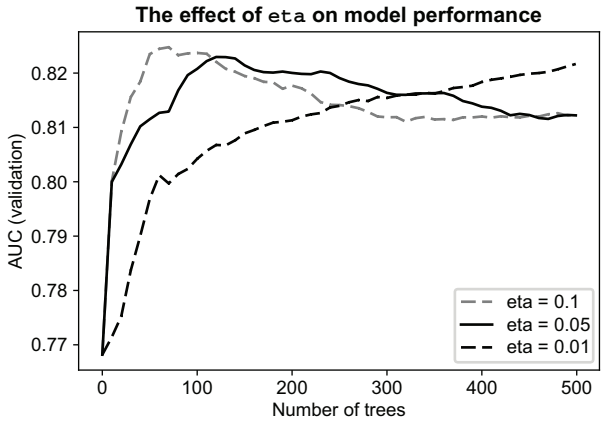


Figure 6.34 The model requires more trees when `eta` is small.

When `eta` is 0.05, the performance is similar to 0.1, but it takes 60 more iterations to reach the peak.

For `eta` of 0.01, it grows too slowly, and even after 500 iterations, it hasn't reached the peak. If we tried it for more iterations, it could potentially get to the same level of AUC as other values. Even if it was the case, it's not practical: it becomes computationally expensive to evaluate all these trees during prediction time.

Thus, we use the value of 0.1 for `eta`. Next, let's tune other parameters.

Exercise 6.3

We have a gradient boosting model with `eta=0.1`. It needs 60 trees to get the peak performance. If we increase `eta` to 0.5, what will happen?

a. The number of trees will not change.
b. The model will need more trees to reach its peak performance.
c. The model will need fewer trees to reach its peak performance.

TUNING OTHER PARAMETERS

The next parameter we tune is `max_depth`. The default value is 6, so we can try

- A lower value; for example, 3
- A higher value; for example, 10

The outcome should give us an idea if the best value for `max_depth` is between 3 and 6 or between 6 and 10.

First, check 3:

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Changes max_depth from 6 to 3

The best AUC we get with it is 83.6%.

Next, try 10. In this case, the best value is 81.1%.

This means that the optimal parameter of `max_depth` should be between 3 and 6. When we try 4, however, we see that the best AUC is 83%, which is slightly worse than the AUC we got with the depth of 3 (figure 6.35).

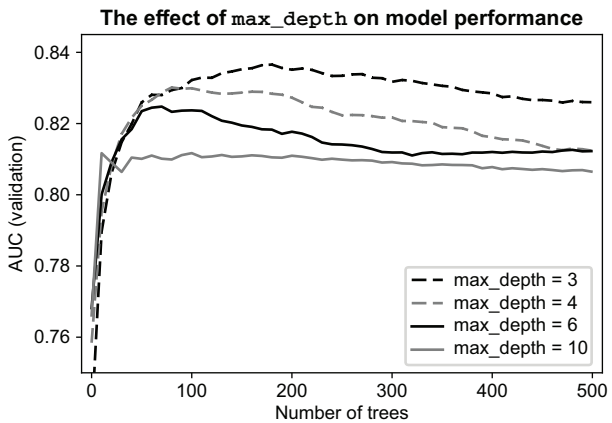


Figure 6.35 The optimal value for `max_depth` is 4: with it, we can achieve an AUC of 83.6%.

The next parameter we tune is `min_child_weight`. It's the same as `min_leaf_size` in decision trees from Scikit-learn: it controls the minimal number of observations a tree can have in a leaf.

Let's try a range of values and see which one works best. In addition to the default value (1), we can try 10 and 30 (figure 6.36).

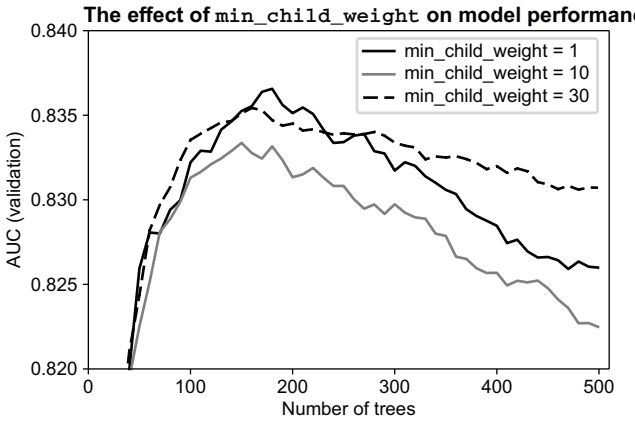


Figure 6.36 The optimal value for `min_child_weight` is 1, but it's not drastically different from other values for this parameter.

From figure 6.36 we see that

- For `min_child_weight=1`, AUC is 83.6%.
- For `min_child_weight=10`, AUC is 83.3%.
- For `min_child_weight=30`, AUC is 83.5%.

The difference between these options is not significant, so we'll leave the default value.

The parameters for our final model are

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

We need to do one last step before we can finish the model: we need to select the optimal number of trees. It's quite simple: look at the iteration when the validation score peaked and use this number.

In our case, we need to train 180 trees for the final model (figure 6.37):

```
[160] train-auc:0.935513    val-auc:0.835536
[170] train-auc:0.937885    val-auc:0.836384
```

```
[180] train-auc:0.93971    val-auc:0.836565 <- best
[190] train-auc:0.942029   val-auc:0.835621
[200] train-auc:0.943343   val-auc:0.835124
```

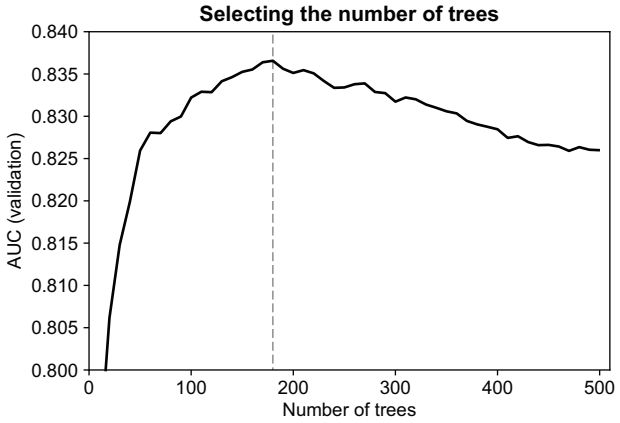


Figure 6.37 The optimal number of trees for the final model is 180.

The best the random forest model was able to get 82.5% AUC, whereas the best the gradient boosting model could get was 1% more (83.6%).

This is the best model, so let's use it as our final model — and we should use it for scoring loan applications.

6.4.4 Testing the final model

We're almost ready to use it for risk scoring. We still need to do two things before we can use it:

- Retrain the final model on both train and validation datasets combined. We no longer need the validation dataset, so we can use more data for training, which will make the model slightly better.
- Test the model on the test set. This is the part of data we kept aside from the beginning. Now we use it to make sure the model didn't overfit and performs well on completely unseen data.

The next steps are:

- Apply the same preprocessing to `df_full_train` and `df_test` as we did to `df_train` and `df_val`. As a result, we get the feature matrices `X_train` and `X_test` as well as our target variables `y_train` and `y_test`.
- Train a model on the combined dataset with the parameters we selected previously.
- Apply the model to the test data to get the test predictions.
- Verify that the model performs well and doesn't overfit.

Let's do it. First, create the target variable:

```
y_train = (df_train_full.status == 'default').values
y_test = (df_test.status == 'default').values
```

Because we use the entire DataFrame for creating the feature matrix, we need to remote the target variable:

```
del df_train_full['status']
del df_test['status']
```

Next, we convert DataFrames into lists of dictionaries and then use one-hot encoding to get the feature matrices:

```
dict_train = df_train_full.fillna(0).to_dict(orient='records')
dict_test = df_test.fillna(0).to_dict(orient='records')

dv = DictVectorizer(sparse=False)
X_train = dv.fit_transform(dict_train)
X_test = dv.transform(dict_test)
```

Finally, we train the XGBoost model using this data and the optimal parameters we determined previously:

```
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=dv.feature_names_)
dtest = xgb.DMatrix(X_test, label=y_test, feature_names=dv.feature_names_)

xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}

num_trees = 160

model = xgb.train(xgb_params, dtrain, num_boost_round=num_trees)
```

Then evaluate its performance on the test set:

```
y_pred_xgb = model.predict(dtest)
roc_auc_score(y_test, y_pred_xgb)
```

The output is 83.2%, which is comparable to 83.6% — the performance on the validation set. It means that our model doesn't overfit and can work well with customers it hasn't seen.

Exercise 6.4

The main difference between random forest and gradient boosting is

a. Trees in gradient boosting are trained sequentially, and each next tree improves the previous one. In a random forest, all trees are trained independently.
b. Gradient boosting is a lot faster than using a random forest.
c. Trees in a random forest are trained sequentially, and each next tree improves the previous one. In gradient boosting, all trees are trained independently.

6.5 Next steps

We've learned the basics about decision trees, random forest, and gradient boosting. We've learned a lot, but there's much more than we could fit in this chapter. You can explore this topic further by doing the exercises.

6.5.1 Exercises

- Feature engineering is the process of creating new features out of existing ones. For this project, we haven't created any features; we simply used the ones provided in the dataset. Adding more features should help improve the performance of our model. For example, we can add the ratio of requested money to the total price of the item. Experiment with engineering more features.
- When training a random forest, we get different models by selecting a random subset of features for each tree. To control the size of the subset, we use the `max_features` parameter. Try adjusting this parameter, and see if it changes the AUC on validation.
- Extreme randomized trees (or extra trees, for short) is a variation of a random forest where the idea of randomization is taken to the extreme. Instead of finding the best possible split, it picks a splitting condition randomly. This approach has a few advantages: extra trees are faster to train, and they are less prone to overfitting. On the other hand, they require more trees to have adequate performance. In Scikit-learn, `ExtraTreesClassifier` from the `ensemble` package implements it. Experiment with it for this project.
- In XGBoost, the `colsample_bytree` parameter controls the number of features we select for each tree — it's similar to `max_features` from the random forest. Experiment with this parameter, and see if it improves the performance: try values from 0.1 to 1.0 with a step of 0.1. Usually the optimal values are between 0.6 and 0.8, but sometimes 1.0 gives the best result.
- In addition to randomly selecting columns (features), we can also select a subset of rows (customers). This is called *subsampling*, and it helps to prevent overfitting. In XGBoost, the `subsample` parameter controls the fraction of examples we select for training each tree in the ensemble. Try values from 0.4 to 1.0 with a step of 0.1. Usually the optimal values are between 0.6 and 0.8.

6.5.2 Other projects

- All tree-based models can solve the regression problem — predict a number. In Scikit-learn, DecisionTreeRegressor, and RandomForestRegressor, implement the regression variation of the models. In XGBoost, we need to change the objective to `reg:squarederror`. Use these models for predicting the price of the car, and try to solve other regression problems as well.

Summary

- Decision tree is a model that represents a sequence of if-then-else decisions. It's easy to understand, and it also performs quite well in practice.
- We train decision trees by selecting the best split using impurity measures. The main parameters that we control are the depth of the tree and the maximum number of samples in each leaf.
- A random forest is a way to combine many decision trees into one model. Like a team of experts, individual trees can make mistakes, but together, they are less likely to reach an incorrect decision.
- A random forest should have a diverse set of models to make good predictions. That's why each tree in the model uses a different set of features for training.
- The main parameters we need to change for random forest are the same as for decision trees: the depth and the maximum number of samples in each leaf. Additionally, we need to select the number of trees we want to have in the ensemble.
- While in a random forest the trees are independent, in gradient boosting, the trees are sequential, and each next model corrects the mistakes of the previous one. In some cases, this leads to better predictive performance.
- The parameters we need to tune for gradient boosting are similar for a random forest: the depth, the maximum number of observations in the leaf, and the number of trees. In addition to that, we have `eta` — the learning rate. It specifies the contribution of each individual tree to the ensemble.

Tree-based models are easy to interpret and understand, and often they perform quite well. Gradient boosting is great and often achieves the best possible performance on structured data (data in tabular format).

In the next chapter, we look at neural nets: a different type of model, which, in contrast, achieves best performance on unstructured data, such as images.

Answers to exercises

- Exercise 6.1 A) With one more feature, training takes longer.
- Exercise 6.3 C) The model will need fewer trees to reach its peak performance.
- Exercise 6.2 B) Randomly selecting a different subset of features for each tree.
- Exercise 6.4 A) Trees in gradient boosting are trained sequentially. In a random forest, trees are trained independently.

7 Neural networks and deep learning

This chapter covers

- Convolutional neural networks for image classification
- TensorFlow and Keras — frameworks for building neural networks
- Using pretrained neural networks
- Internals of a convolutional neural network
- Training a model with transfer learning
- Data augmentations — the process of generating more training data

Previously, we only dealt with tabular data — data in CSV files. In this chapter, we'll work with a completely different type of data — images.

The project we prepared for this chapter is classification of clothes. We will predict if an image of clothing is a T-shirt, a shirt, a skirt, a dress, or something else.

This is an image classification problem. To solve it, we will learn how to train a deep neural network using TensorFlow and Keras to recognize the types of clothes. The materials of this chapter will help you start using neural networks and perform any similar image classification project.

Let's start!

7.1 Fashion classification

Imagine that we work at an online fashion marketplace. Our users upload thousands of images every day to sell their clothes. We want to help our users create listings faster by automatically recommending the right category for their clothes.

To do it, we need a model for classifying images. Previously, we covered multiple models for classification: logistic regression, decision trees, random forests, and gradient boosting. These models work great with tabular data, but it's quite difficult to use them for images.

To solve our problem, we need a different type of model: a convolutional neural network, a special model used for images. These neural networks consist of many layers, and that's why they are often called "deep." Deep learning is a part of machine learning that deals with deep neural networks.

The frameworks for training these models are also different from what we saw previously, so in this chapter we use TensorFlow and Keras instead of Scikit-learn.

The plan for our project is

- First, we download the dataset and use a pretrained model to classify images.
- Then, we talk about neural networks, and see how they work internally.
- After that, we adjust the pretrained neural network for solving our tasks.
- Finally, we expand our dataset by generating many more images from the images we have.

For evaluating the quality of our models, let's use accuracy: the percentage of items we classified correctly.

It's not possible to cover all the theory behind deep learning in just one chapter. In this book, we focus on the most fundamental parts, which is enough for completing the project of this chapter and other similar projects about image classification. When we come across concepts that are nonessential for completing this project, for details, we refer to CS231n — a course about neural networks from Stanford University. The course notes are available online at cs231n.github.io.

The code for this project is available in the book's GitHub repository at https://github.com/alexeygrigorev/mlbookcamp-code, in the folder chapter-07-neural-nets. There are multiple notebooks in this folder. For most of the chapter, we need 07-neural-nets-train.ipynb. For section 7.5, we use 07-neural-nets-test.ipynb.

7.1.1 GPU vs. CPU

Training a neural network is a computationally demanding process, and it requires powerful hardware to make it faster. To speed up training, we usually use GPUs — graphical processing units, or, simply, graphic cards.

For this chapter, a GPU is not required. You can do everything on your laptop, but without a GPU, it will be approximately eight times slower than with a GPU.

If you have a GPU card, you need to install special drivers from TensorFlow to use it. (Check the official documentation of TensorFlow for more details: https://www.tensorflow.org/install/gpu.) Alternatively, you can rent a preconfigured GPU server. For example, we can use AWS SageMaker to rent a Jupyter Notebook instance with everything already set up. Refer to appendix E for details on how to use SageMaker. Other cloud providers also have servers with GPU, but we do not cover them in this book. Regardless of the environment you use, the code works anywhere, as long as you can install Python and TensorFlow there.

After deciding where to run the code, we can go to the next step: downloading the dataset.

7.1.2 Downloading the clothing dataset

First, let's create a folder for this project and call it 07-neural-nets.

For this project, we need a dataset of clothes. We will use a subset of the clothing dataset (for more information, check https://github.com/alexeygrigorev/clothing-dataset), which contains around 3,800 images of 10 different classes. The data is available in a GitHub repository. Let's clone it:

```
git clone https://github.com/alexeygrigorev/clothing-dataset-small.git
```

If you're doing this in AWS SageMaker, you can execute this command in a cell of the notebook. Just add the exclamation sign ("!") before the command (figure 7.1) .

```
!git clone https://github.com/alexeygrigorev/clothing-dataset-small.git
Cloning into 'clothing-dataset-small'...
remote: Enumerating objects: 3839, done.
remote: Counting objects: 100% (400/400), done.
remote: Compressing objects: 100% (400/400), done.
remote: Total 3839 (delta 9), reused 384 (delta 0), pack-reused 3439
Receiving objects: 100% (3839/3839), 100.58 MiB | 1.21 MiB/s, done.
Resolving deltas: 100% (10/10), done.
Checking out files: 100% (3783/3783), done.
```

Figure 7.1 Executing a shell script command in Jupyter: simply add the exclamation sign ("!") in front of the command.

The dataset is already split into folders (figure 7.2):

- train: Images for training a model (3,068 images)
- validation: Images for validating (341 image)
- test: Images for testing (372 images)

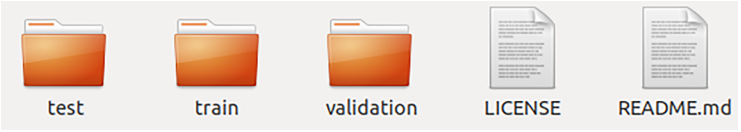


Figure 7.2 The dataset is already split into train, validation, and test.

Each of these folders has 10 subfolders: one subfolder for each type of clothing (figure 7.3).

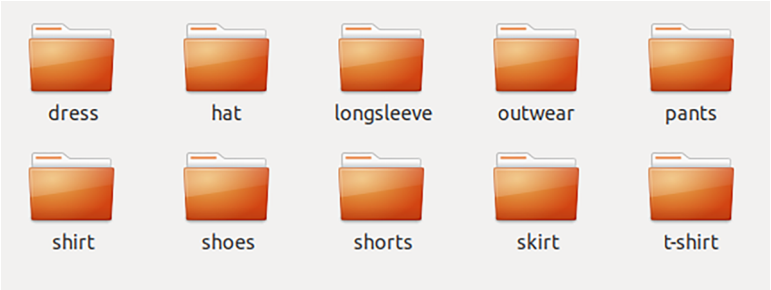


Figure 7.3 Images in the dataset are organized in subfolders.

As we see, this dataset contains 10 classes of clothes, from dresses and hats, to shorts and shoes.

Each subfolder contains images of only one class (figure 7.4).

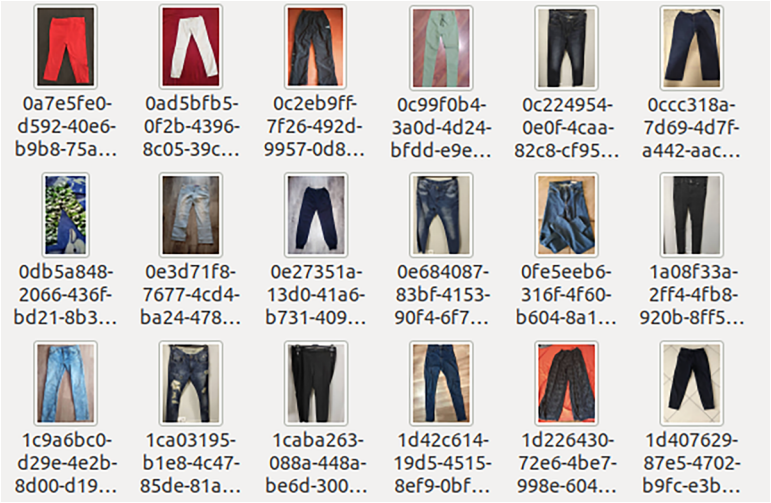


Figure 7.4 The content of the pants folder

In these pictures, the clothing items have different colors and the background is different. Some items are on the floor, some are spread out on a bed or a table, and some are hung in front of a neutral background.

With this variety of images, it's not possible to use the methods we previously covered. We need a special type of model: neural networks. This model also requires different tools, and we cover them next.

7.1.3 TensorFlow and Keras

If you use AWS SageMaker, you don't need to install anything: it already has all the required libraries.

But if you use your laptop with Anaconda, or run the code somewhere else, you need to install TensorFlow — a library for building neural networks.

Use `pip` to do it:

```
pip install tensorflow
```

TensorFlow is a low-level framework, and it's not always easy to use. In this chapter, we use Keras — a higher-level library built on top of TensorFlow. Keras makes training neural networks a lot simpler. It comes preinstalled together with TensorFlow, so we don't need to install anything extra.

> **NOTE** Previously, Keras was not a part of TensorFlow, and you can find many examples on the internet where it's still a separate library. However, the interface of Keras hasn't changed significantly, so most of the examples you may discover still work in the new Keras.

At the time of writing, the latest version of TensorFlow was 2.3.0 and AWS SageMaker used TensorFlow version 2.1.0. The difference in versions is not a problem; the code from this chapter works for both versions, and it will most likely work for all TensorFlow 2 versions.

We're ready to start and create a new notebook called chapter-07-neural-nets. As usual, we begin by importing NumPy and MatplotLib:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Next, import TensorFlow and Keras:

```
import tensorflow as tf
from tensorflow import keras
```

The preparation work is done, and now we can take a look at the images we have.

7.1.4 Loading images

Keras offers a special function for loading images called `load_img`. Let's import it:

```
from tensorflow.keras.preprocessing.image import load_img
```

> **NOTE** When Keras was a separate package, the imports looked like this:
>
> ```
> from keras.preprocessing.image import load_img
> ```
>
> If you find some old Keras code on the internet and want to use it with the latest versions of TensorFlow, simply add `tensorflow.` at the beginning when importing it. Most likely, it will be enough to make it work.

Let's use this function to take a look at one of the images:

```
path = './clothing-dataset-small/train/t-shirt'
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'
fullname = path + '/' + name
load_img(fullname)
```

After executing the cell, we should see an image of a T-shirt (figure 7.5).

```
path = './clothing-dataset-small/train/t-shirt'
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'
fullname = path + '/' + name
load_img(fullname)
```

Figure 7.5 An image of a T-shirt from the train set

To use this image in a neural network, we need to resize it because the models always expect images of a certain size. For example, the network we use in this chapter requires a 150 × 150 image or an 299 × 299 image.

To resize the image, specify the `target_size` parameter:

```
load_img(fullname, target_size=(299, 299))
```

As a result, the image becomes square and a bit squashed (figure 7.6).

```
load_img(fullname, target_size=(299, 299))
```

Figure 7.6 To resize an image, use the `target_size` parameter.

Let's now use a neural network to classify this image.

7.2 Convolutional neural networks

Neural networks are a class of machine learning models for solving classification and regression problems. Our problem is a classification problem — we need to determine the category of an image.

However, our problem is special: we're dealing with images. This is why we need a special type of neural network — a convolutional neural network, which can extract visual patterns from an image and use them to make predictions.

Pre-trained neural networks are available on the internet, so let's see how we can use one of them for this project.

7.2.1 Using a pretrained model

Training a convolutional neural network from scratch is a time-consuming process and requires a lot of data and powerful hardware. It may take weeks of nonstop training for large datasets like ImageNet with 14 million images. (Check image-net.org for more information.)

Luckily, we don't need to do it ourselves: we can use pretrained models. Usually, these models are trained on ImageNet and can be used for general-purpose image classification.

It's very simple, and we don't even need to download anything ourselves — Keras will take care of it automatically. We can use many different types of models (called *architectures*). You can find a good summary of available pretrained models in the official Keras documentation (https:// keras.io/api/applications/).

For this chapter, we'll use Xception, a relatively small model that has good performance. First, we need to import the model itself and some helpful functions:

```
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.xception import preprocess_input
from tensorflow.keras.applications.xception import decode_predictions
```

We imported three things:

- `Xception`: the actual model
- `preprocess_input`: a function for preparing the image to be used by the model
- `decode_prediction`: a function for decoding the model's prediction

Let's load this model:

```
model = Xception(
    weights='imagenet',
    input_shape=(299, 299, 3)
)
```

We specify two parameters here:

- `weights`: We want to use a pretrained model from ImageNet.
- `imput_shape`: The size of the input images: height, width, and the number of channels. We resize the images to 299 × 299, and each image has three channels: red, green and blue.

When we load it for the first time, it downloads the actual model from the internet. After it's done, we can use it.

Let's test it on the image we saw previously. First, we load it using the `load_img` function:

```
img = load_img(fullname, target_size=(299, 299))
```

The `img` variable is an `Image` object, which we need to convert to a NumPy array. It's easy to do:

```
x = np.array(img)
```

This array should have the same shape as the image. Let's check it:

```
x.shape
```

We see `(299, 299, 3)`. It contains three dimensions (figure 7.7):

- The width of the image: 299
- The height of the image: 299
- The number of channels: red, green, blue

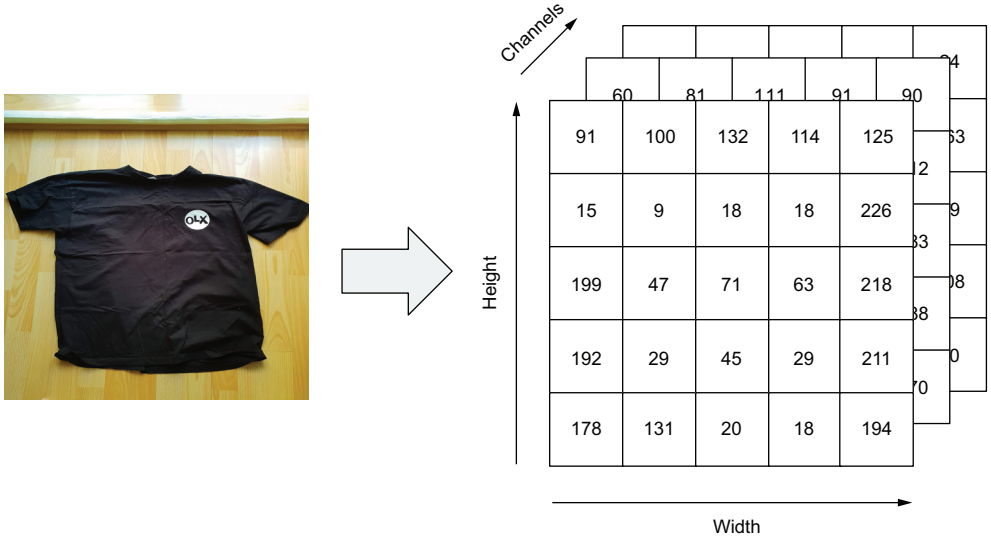


Figure 7.7 After converting, an image becomes a NumPy array of shape width × height × number of channels.

This matches the input shape we specified when loading the neural network. However, the model doesn't expect to get just a single image. It gets a *batch* of images — several images put together in one array. This array should have four dimensions:

- The number of images
- The width
- The height
- The number of channels

For example, for 10 images, the shape is `(10, 299, 299, 3)`. Because we have just one image, we need to create a batch with this single image:

```
X = np.array([x])
```

NOTE If we had several images, for example, `x`, `y` and `z`, we'd write

```
X = np.array([x, y, z])
```

Let's check its shape:

```
X.shape
```

As we see, it's `(1, 299, 299, 3)` — it's one image of size 299 × 299 with three channels.

Before we can apply the model to our image, we need to prepare it with the `preprocess_input` function:

```
X = preprocess_input(X)
```

This function converts the integers between 0 and 255 in the original array to numbers between –1 and 1.

Now, we're ready to use the model.

7.2.2 Getting predictions

To apply the model, use the `predict` method:

```
pred = model.predict(X)
```

Let's take a look at this array:

```
pred.shape
```

This array is quite large — it contains 1,000 elements (figure 7.8).

```
pred = model.predict(X)
```

```
pred.shape
```

```
(1, 1000)
```

```
pred[0, :10]
```

```
array([0.0003238 , 0.00015736, 0.00021406, 0.00015296, 0.00024657,
       0.00030446, 0.00032349, 0.00014726, 0.00020487, 0.00014866],
      dtype=float32)
```

Figure 7.8 The output of the pretrained Xception model

This Xception model predicts whether an image belongs to one of 1,000 classes, so each element in the prediction array is the probability of belonging to one of these classes.

We don't know what these classes are, so it's difficult to make sense from this prediction just by looking at the numbers. Luckily, we can use a function, `decode_predictions`, that decodes the prediction into meaningful class names:

```
decode_predictions(pred)
```

It shows the top five most likely classes for this image:

```
[[('n02667093', 'abaya', 0.028757658),
  ('n04418357', 'theater_curtain', 0.020734021),
  ('n01930112', 'nematode', 0.015735716),
  ('n03691459', 'loudspeaker', 0.013871926),
  ('n03196217', 'digital_clock', 0.012909736)]]
```

Not quite the result we expected. Most likely, images like this T-shirt are not common in ImageNet, and that's why the result isn't useful for our problem.

Even though these results aren't particularly helpful for us, we can use this neural network as a base model for solving our problem.

To understand how we can do it, we should first get a feeling for how convolutional neural networks work. Let's see what happens inside the model when we invoke the `predict` method.

7.3 Internals of the model

All neural networks are organized in layers. We take an image, pass it through all the layers, and, at the end, get the predictions (figure 7.9).

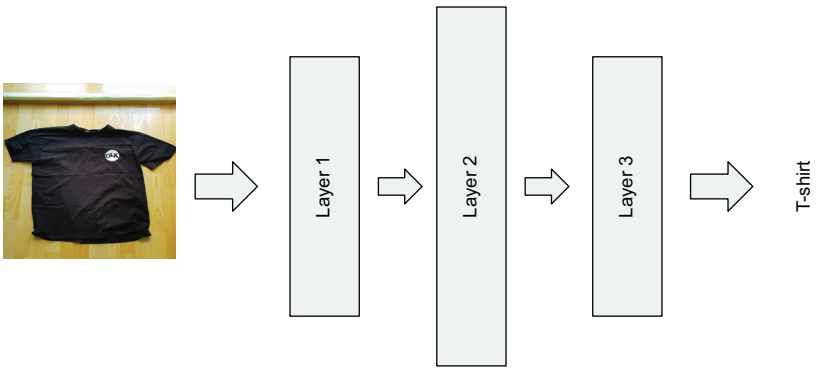


Figure 7.9 A neural network consists of multiple layers.

Usually, a model has a lot of layers. For example, the Xception model we use here has 71 layers. That's why these neural networks are called "deep" neural networks — because they have many layers.

For a convolutional neural network, the most important layers are

- Convolutional layers
- Dense layers

First, let's take a look at convolutional layers.

7.3.1 Convolutional layers

Even though "convolutional layer" sounds complicated, it's nothing more than a set of *filters* — small "images" with simple shapes like stripes (figure 7.10).

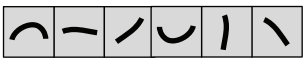


Figure 7.10 Examples of filters for a convolutional layer (not from a real network)

The filters in a convolutional layer are learned by the model during training. However, because we are using a pretrained neural network, we don't need to worry about it; we already have the filters.

To apply a convolutional layer to a picture, we slide each filter across this image. For example, we can slide it from left to right and from top to bottom (figure 7.11).

Figure 7.11 To apply a filter, we slide it over an image.

While sliding, we compare the content of the filter with the content of the image under the filter. For each comparison, we record the degree of similarity. This way, we get a *feature map* — an array with numbers, where a large number means a match between the filter and the image, and a low number means no match (figure 7.12).

So, a feature map tells us where on the image we can find the shape from the filter.

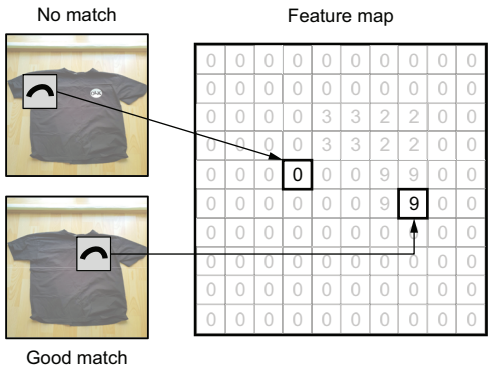


Figure 7.12 A feature map is a result of applying a filter to an image. A high value in the map corresponds to areas with a high degree of similarity between the image and the filter.

One convolutional layer consists of many filters, so we actually get multiple feature maps — one for each filter (figure 7.13).

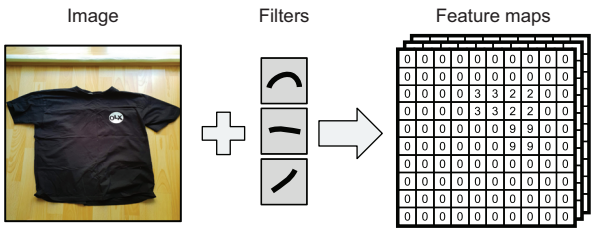


Figure 7.13 Each convolutional layer contains many filters, so we get a set of feature maps: one for each filter that we use.

Now we can take the output of one convolutional layer and use it as the input to the next layer.

From the previous layer we know the location of different stripes and other simple shapes. When two simple shapes occur in the same location, they form more complex patterns — crosses, angles, or circles.

That's what the filters of the next layer do: they combine shapes from the previous layer into more complex structures. The deeper we go down the network, the more complex patterns the network can recognize (figure 7.14).

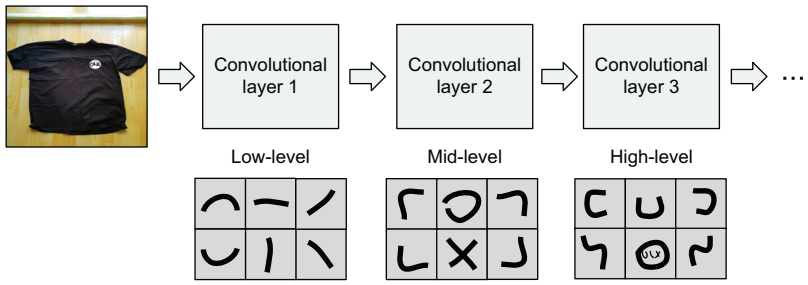


Figure 7.14 Deeper convolutional layers can detect progressively more complex features of the image.

We repeat this process to detect more and more complex shapes. This way, the network "learns" some distinctive features of the image. For clothes, it can be short or long sleeves or the type of neck. For animals, it can be pointy or floppy ears or the presence of whiskers.

At the end, we get a vector representation of an image: a one-dimensional array, where each position corresponds to some high-level visual features. Some parts of the

array may correspond to sleeves, whereas other parts represent ears and whiskers. At this level, it's usually difficult to make sense from these features, but they have enough discriminative power to distinguish between a T-shirt and pants or between a cat and a dog.

Now we need to use this vector representation to combine these high-level features and arrive at the final decision. For that, we use a different kind of layers — dense layers.

7.3.2 Dense layers

Dense layers process the vector representation of an image and translate these visual features to the actual class — T-shirt, dress, jacket, or other class (figure 7.15).

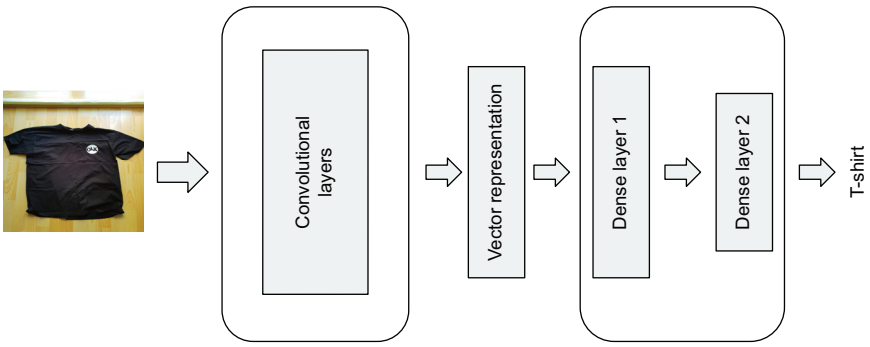


Figure 7.15 Convolutional layers transform an image into its vector representation, and dense layers translate the vector representation into the actual label.

To understand how it works, let's take a step back and think how we could use logistic regression for classifying images.

Suppose we want to build a binary classification model for predicting whether an image is a T-shirt. In this case, the input to logistic regression is the vector representation of an image — a feature vector x.

From chapter 3, we know that to make the prediction, we need to combine the features in x with the weights vector w and then apply the sigmoid function to get the final prediction:

$$\text{sigmoid}(x^T w)$$

We can show it visually by taking all the components of the vector x and connecting them to the output — the probability of being a T-shirt (figure 7.16).

What if we need to make predictions for multiple classes? For example, we may want to know if we have an image of a T-shirt, shirt, or dress. In this case, we can build multiple logistic regressions — one for each class (figure 7.17).

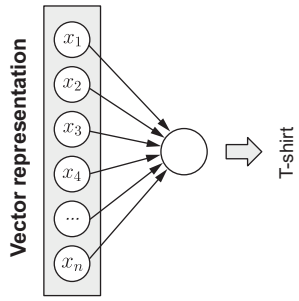


Figure 7.16 Logistic regression: we take all the components of the feature vector *x* and combine them to get the prediction.

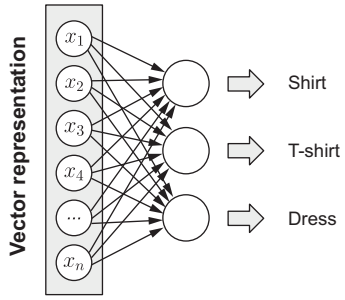


Figure 7.17 To predict multiple classes, we train multiple logistic regression models.

By putting together multiple logistic regression models, we just created a small neural network!

To make it visually simpler, we can combine the outputs into one layer — the output layer (figure 7.18).

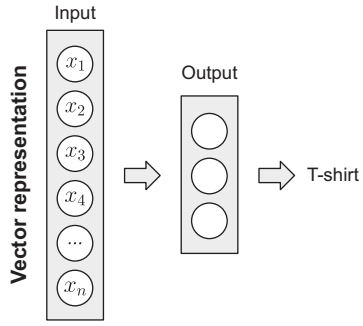


Figure 7.18 Multiple logistic regressions put together form a small neural network.

When we have 10 classes we want to predict, we have 10 elements in the output layer. To make a prediction, we look at each element of the output layer and take the one with the highest score.

In this case, we have a network with one layer: the layer that converts the input to the output.

This layer is called a *dense layer*. It's "dense" because it connects each element of the input with all the elements of its output. For this reason, these layers are sometimes called "fully connected" (figure 7.19).

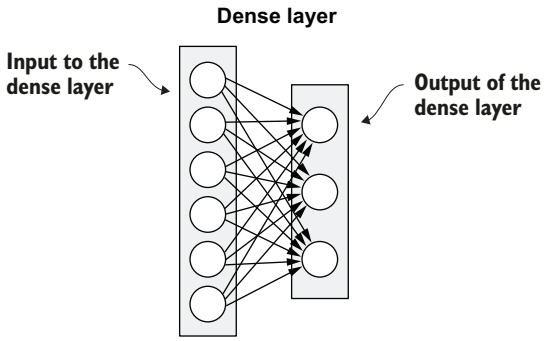


Figure 7.19 A dense layer connects every element of its input with every element of its output.

However, we don't have to stop at just one output layer. We can add more layers between the input and the final output (figure 7.20).

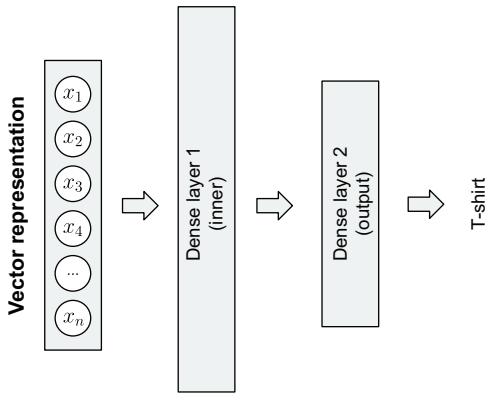


Figure 7.20 A neural network with two layers: one inner layer and one output layer

So, when we invoke `predict`, the image first goes through a series of convolutional layers. This way, we extract the vector representation of this image. Next, this vector representation goes through a series of dense layers, and we get the final prediction (figure 7.21).

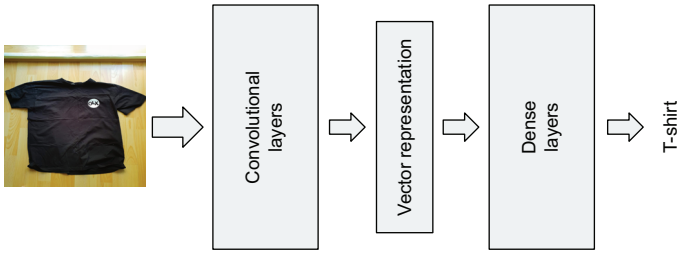


Figure 7.21 In a convolutional neural network, an image first goes through a series of convolutional layers, and then through a series of dense layers.

NOTE In this book, we give a simplified and high-level overview of the internals of convolutional neural networks. Many other layers exist in addition to convolutional layers and dense layers. For a more in-depth introduction to this topic, check the CS231n notes (cs231n.github.io/convolutional-networks).

Now let's get back to code and see how we can adjust a pretrained neural network for our project.

7.4 Training the model

Training a convolutional neural network takes a lot of time and requires a lot of data. But there's a shortcut: we can use *transfer learning*, an approach where we adapt a pretrained model to our problem.

7.4.1 Transfer learning

The difficulty in training usually comes from convolutional layers. To be able to extract a good vector representation from an image, the filters need to learn good patterns. For that, the network has to see many different images — the more, the better. But once we have a good vector representation, training dense layers is relatively easy.

This means that we can take a neural network pretrained on ImageNet and use it for solving our problem. This model has already learned good filters. So, we take this model and keep the convolutional layers, but drop the dense layers and instead train new ones (figure 7.22).

In this section, we do exactly that. But before we can start training, we need to get our dataset ready.

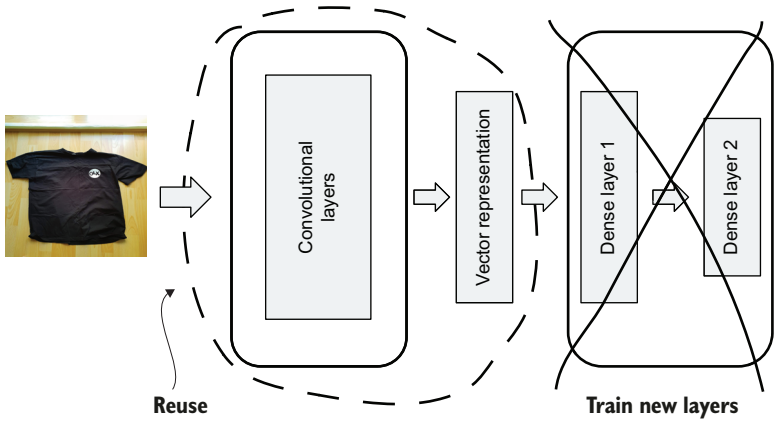


Figure 7.22 To adapt a pretrained model to a new domain, we keep the old convolutional layers but train new dense layers.

7.4.2 Loading the data

In previous chapters, we loaded the entire dataset into memory and used it to get X—the matrix with features. With images, it's more difficult: we may not have enough memory to keep all the images.

Keras comes with a solution — `ImageDataGenerator`. Instead of loading the entire dataset into memory, it loads the images from disk in small batches. Let's use it:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)
```

Applies the preprocess_input function to each image

We already know that images need to be preprocessed using the `preprocess_input` function. That's why we need to tell `ImageDataGenerator` how the data should be prepared.

We have a generator now, so we just need to point it to the directory with the data. For that, use the `flow_from_directory` method:

```
train_ds = train_gen.flow_from_directory(
    "clothing-dataset-small/train",
    target_size=(150, 150),
    batch_size=32,
)
```

Loads all the images from the train directory

Resizes the images to 150 × 150

Loads the images in batches of 32 images

For our initial experiments, we use small images of size 150 × 150. This way, it's faster to train the model. Also, the small size makes it possible to use a laptop for training.

We have 10 classes of clothing in our dataset, and images of each class are stored in a separate directory. For example, all T-shirts are stored in the t-shirt folder. The generator can use the folder structure to infer the label for each image.

When we execute the cell, it informs us how many images there are in the train dataset and how many classes:

```
Found 3068 images belonging to 10 classes.
```

Now we repeat the same process for the validation dataset:

```
validation_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

val_ds = validation_gen.flow_from_directory(
    "clothing-dataset-small/validation",
    target_size=image_size,
    batch_size=batch_size,
)
```

Like previously, we use the train dataset for training the model and the validation dataset for selecting the best parameters.

We have loaded the data, and now we're ready to train a model.

7.4.3 Creating the model

First, we need to load the base model — this is the pretrained model that we're using for extracting the vector representation from images. Like previously, we also use Xception, but this time, we include only the part with pretrained convolutional layers. After that, we add our own dense layers.

So, let's create the base model:

```
base_model = Xception(
    weights='imagenet',
    include_top=False
    input_shape=(150, 150, 3),
)
```

Uses the model pretrained on ImageNet

Keeps only the convolutional layers

Images should be 150 × 150 with three channels.

Note the `include_top` parameter: this way, we explicitly specify that we're not interested in the dense layers of the pretrained neural network, only in the convolutional layers. In Keras terminology, the "top" is the set of final layers of the network (figure 7.23).

We don't want to train the base model; attempting to do so will destroy all the filters. So, we "freeze" the base model by setting the `trainable` parameter to False:

```
base_model.trainable = False
```

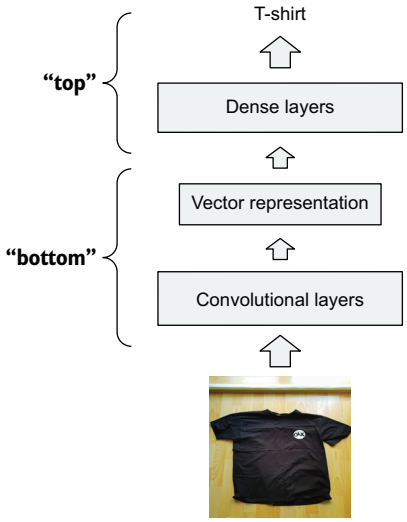


Figure 7.23 In Keras, the input to the network is on the bottom and the output is on the top, so `include_top=False` means "don't include the final dense layers."

Now let's build the clothing classification model:

```
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

outputs = keras.layers.Dense(10)(vector)

model = keras.Model(inputs, outputs)
```

Input images should be 150 × 150 with three channels.

Uses the base_model to extract the high-level features.

Extracts the vector representation: converts the output of base_model to a vector

Adds a dense layer of size 10: one element for each class

Combines the inputs and the outputs into a Keras model

The way we build the model is called the "functional style." It may be confusing at first, so let's take a look at each line individually.

First, we specify the input and the size of the arrays we expect:

```
inputs = keras.Input(shape=(150, 150, 3))
```

Next, we create the base model:

```
base = base_model(inputs, training=False)
```

Even though `base_model` is already a model, we use it as a function and give it two parameters — `inputs`, and `training=False`:

- The first parameter says what will be the input to `base_model`. It will come from `inputs`.
- The second parameter (`training=False`) is optional and says that we don't want to train the base model.

The result is `base`, which is a *functional component* (like `base_model`) that we can combine with other components. We use it as the input to the next layer:

```
vector = keras.layers.GlobalAveragePooling2D()(base)
```

Here, we create a pooling layer — a special construction that allows us to convert the output of a convolutional layer (a 3-D array) into a vector (a one-dimensional array).

After creating it, we immediately invoke it with `base` as the argument. This way, we say that the input to this layer comes from `base`.

This may be a bit confusing because we create a layer and immediately connect it to base. We can rewrite it to make it simpler to understand:

```
pooling = keras.layers.GlobalAveragePooling2D()
vector = pooling(base)
```

Creates a pooling layer first

Connects it to base

As a result, we get `vector`. This is another functional component that we connect to the next layer — a dense layer:

```
outputs = keras.layers.Dense(10)(vector)
```

Similarly, we first create the layer, and then connect it to `vector`. For now, we create a network with only one dense layer. It's enough to get started.

Now the result is `outputs` — the final result that we want to get out of the network.

So, in our case, the data comes into `inputs` and goes out of `outputs`. We just need to do one final step — wrap both `inputs` and `outputs` into a `Model` class:

```
model = keras.Model(inputs, outputs)
```

We need to specify two parameters here:

- What the model will get as input, which is `inputs` in our case
- What the output of the model is, which is `outputs`

Let's take a step back and look at the model definition code again, following the flow of data from `inputs` to `outputs` (figure 7.24).

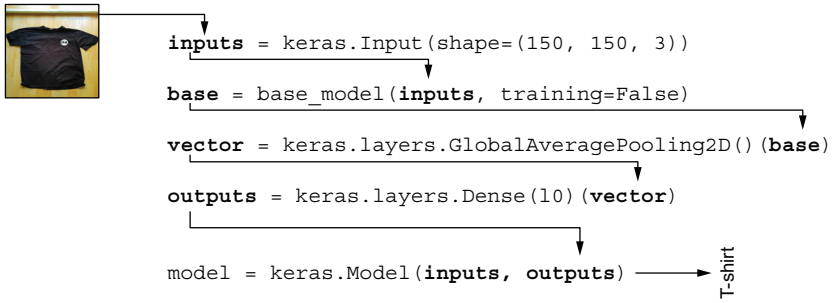


Figure 7.24 The flow of data: an image goes to `inputs`, then `base_model` converts it to `base`, then pooling converts it to `vector`, and then a dense layer converts it to `output`. At the end, `inputs` and `outputs` go to a Keras model.

To make it easier to visualize, we can think of every line of code as a block, which gets the data from the previous block, transforms it, and passes to the next block (figure 7.25).

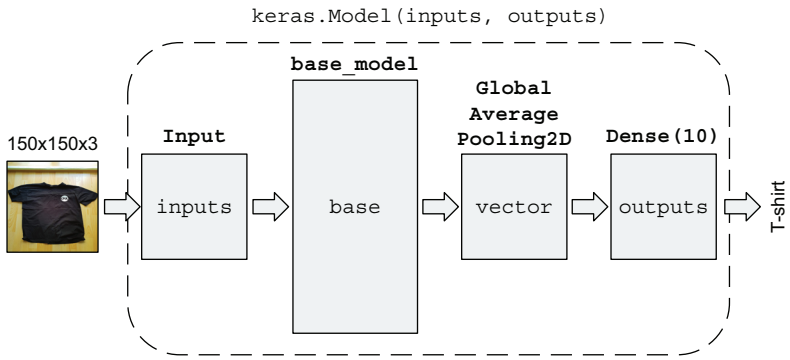


Figure 7.25 The flow of data: each line of Keras code as a block

So, we have created a model that can take in an image, get the vector representation using the base model, and make the final prediction with a dense layer.

Let's train it now.

7.4.4 Training the model

We have specified the model: the input, the elements of the model (the base model, the pooling layer), and the final output layer.

Now we need to train it. For that, we need an *optimizer*, which adjusts the weights of a network to make it better at doing its task.

We won't cover the details of how optimizers work — that's beyond the scope for this book, and it's not required to finish the project. But if you'd like to learn more about them, check the CS231n notes (https://cs231n.github.io/neural-networks-3/). You can see the list of available optimizers in the official documentation for Keras (https://keras.io/api/optimizers/).

For our project, we will use the Adam optimization algorithm — a good default choice, and in most cases, using it is sufficient.

Let's create it:

```
learning_rate = 0.01
optimizer = keras.optimizers.Adam(learning_rate)
```

Adam requires one parameter: the learning rate, which specifies how fast our network learns.

The learning rate may significantly affect the quality of our network. If we set it too high, the network learns too fast and may accidentally skip some important details. In this case, the predictive performance is not optimal. If we set it too low, the network takes too long to train, so the training process is highly ineffective.

We will later adjust this parameter. For now, we set it to 0.01 — a good default value to start with.

To train a model, the optimizer needs to know whether the model is doing well. For that, it uses a loss function, which becomes smaller as the network becomes better. The goal of the optimizer is to minimize this loss.

The `keras.losses` package offers many different losses. Here's a list of the most important ones:

- `BinaryCrossentropy`: For training a binary classifier
- `CategoricalCrossentropy`: For training a classification model with multiple classes
- `MeanSquaredError`: For training a regression model

Because we need to classify clothing into 10 different classes, we use the categorical cross-entropy loss:

```
loss = keras.losses.CategoricalCrossentropy(from_logits=True)
```

For this loss, we specify one parameter: `from_logits=True`. We need to do this because the last layer of our network outputs raw scores (called "logits"), not probabilities. The official documentation recommends doing this for numerical stability (https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy).

NOTE Alternatively, we could define the last layer of the network like this:

```
outputs = keras.layers.Dense(10, activation='softmax')(vector)
```

In this case, we explicitly tell the network to output probabilities: softmax is similar to sigmoid but for multiple classes. Then the output is not "logits" anymore, so we can drop this parameter:

```
loss = keras.losses.CategoricalCrossentropy()
```

Now let's put the optimizer and the loss together. For that, we use the `compile` method of our model:

```
model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=["accuracy"]
)
```

In addition to the optimizer and the loss, we also specify metrics we want to track during training. We're interested in accuracy: the percentage of images with correct predictions.

Our model is ready for training! To do it, use the `fit` method:

```
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

We specify three parameters:

- `train_ds`: The dataset for training
- `epochs`: The number of times it will go over the training data
- `validation_data`: The dataset for evaluation

One iteration over the entire training dataset is called an *epoch*. The more iterations we do, the better the network learns the training dataset.

At some point, it can learn the dataset so well that it starts overfitting. To know when this happens, we need to monitor the performance of our model on the validation dataset. That's why we specify the `validation_data` parameter.

When we start training, Keras informs us about the progress:

```
Train for 96 steps, validate for 11 steps
Epoch 1/10
96/96 [==============================] - 22s 227ms/step - loss: 1.2372 -
     accuracy: 0.6734 - val_loss: 0.8453 - val_accuracy: 0.7713
Epoch 2/10
96/96 [==============================] - 16s 163ms/step - loss: 0.6023 -
     accuracy: 0.8194 - val_loss: 0.7928 - val_accuracy: 0.7859
...
Epoch 10/10
96/96 [==============================] - 16s 165ms/step - loss: 0.0274 -
     accuracy: 0.9961 - val_loss: 0.9342 - val_accuracy: 0.8065
```

From that we can see

- The speed of training: how long each epoch takes.
- The accuracy on the train and validation datasets. We should monitor the accuracy on the validation set to make sure the model doesn't start overfitting. For

example, if the validation accuracy decreases for multiple epochs, it's a sign of overfitting.

- The loss on training and validation. We're not interested in loss — it's less intuitive and the values are harder to interpret.

NOTE Your results will likely be different. The overall predictive performance of the model should be similar, but the exact numbers will not be the same. With neural networks, it's a lot more difficult to ensure perfect reproducibility, even with fixing random seeds.

As you can see, the model quickly becomes 99% accurate on the train dataset, but the score on validation stays around 80% for all the epochs (figure 7.26).

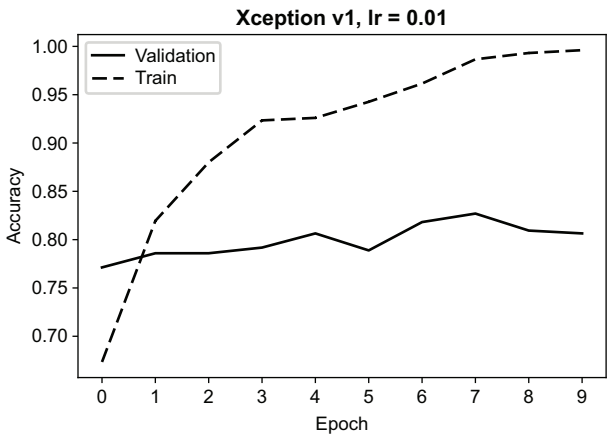


Figure 7.26 The accuracy on the train and validation datasets evaluated after each epoch

The perfect accuracy on the train data doesn't necessarily mean that our model overfits, but it's a good sign that we should adjust the learning rate parameter. We have previously mentioned that it's an important parameter, so let's tune it now.

Exercise 7.1

Transfer learning is the process of using a pretrained model (a base model) for converting an image to its vector representation and then training another model on top of it.

a True
b False

7.4.5 Adjusting the learning rate

We have started with a learning rate of 0.01. It's a good starting point, but it's not necessarily the best rate: we have seen that our model learns too fast and after a few epochs predicts the train set with 100% accuracy.

Let's experiment and try other values for this parameter.

First, to make it easier, we should put the logic for model creating in a separate function. This function takes learning rate as a parameter.

Listing 7.1 A function for creating a model

```
def make_model(learning_rate):
    base_model = Xception(
        weights='imagenet',
        input_shape=(150, 150, 3),
        include_top=False
    )

    base_model.trainable = False

    inputs = keras.Input(shape=(150, 150, 3))

    base = base_model(inputs, training=False)
    vector = keras.layers.GlobalAveragePooling2D()(base)

    outputs = keras.layers.Dense(10)(vector)

    model = keras.Model(inputs, outputs)

    optimizer = keras.optimizers.Adam(learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=["accuracy"],
    )

    return model
```

We've tried 0.01, so let's try 0.001:

```
model = make_model(learning_rate=0.001)
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

We can also try an even smaller value of 0.0001:

```
model = make_model(learning_rate=0.0001)
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

As we see (figure 7.27), for 0.001, the training accuracy doesn't go up as fast as with 0.01, but with 0.0001 it goes up very slowly. The network in this case learns too slow — it *underfits*.

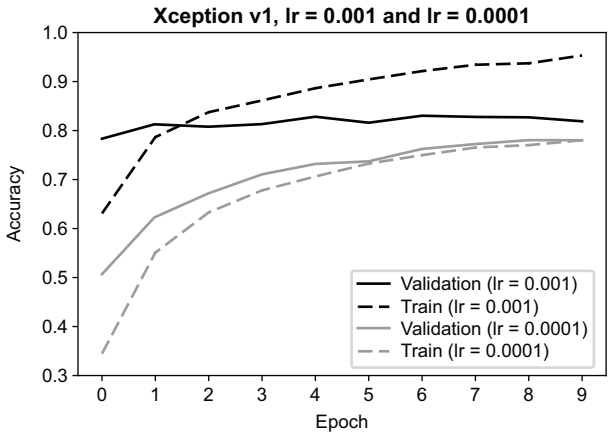


Figure 7.27 The performance of our model with learning rates of 0.001 and 0.0001

If we look at validation scores for all the learning rates (figure 7.28), we see that the learning rate of 0.001 is the best one.

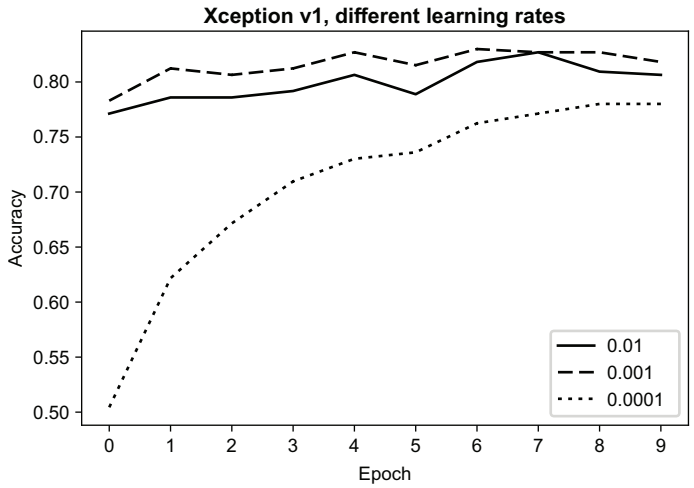


Figure 7.28 The accuracy of our model on the validation set for three different learning rates

For the learning rate of 0.001, the best accuracy is 83% (table 7.1).

Table 7.1 The validation accuracy with different values of dropout rate

Learning rate	0.01	0.001	0.0001
Validation accuracy	82.7%	83.0%	78.0%

NOTE Your numbers may be slightly different. It's also possible that in your experiments, the learning rate of 0.01 achieves slightly better results than 0.001.

The difference between 0.01 and 0.001 is not significant. But if we look at the accuracy on the training data, with 0.01, it overfits the training data a lot faster. At some point, it even achieves an accuracy of 100%. When the discrepancy between the performance on train and validation sets is high, the risk of overfitting is also high. So, we should prefer the learning rate of 0.001.

After training is done, we need to save the model. Now we'll see how to do it.

7.4.6 Saving the model and checkpointing

Once the model is trained, we can save it using the `save_weights` method:

```
model.save_weights('xception_v1_model.h5', save_format='h5')
```

We need to specify the following:

- The output file: `'xception_v1_model.h5'`
- The format: h5, which is a format for saving binary data

You may have noticed that while training, the performance of our model on the validation set jumps up and down. This way, after 10 iterations, we don't necessarily have the best model — maybe the best performance was achieved on iteration 5 or 6.

We can save the model after each iteration, but it generates too much data. And if we rent a server in the cloud, it can quickly take all the available space.

Instead, we can save the model only when it's better than the previous best score on validation. For example, if the previous best accuracy is 0.8, but we have improved it to 0.91, we save the model. Otherwise, we continue the training process without saving the model.

This process is called *model checkpointing*. Keras has a special class for doing it: `ModelCheckpoint`. Let's use it:

```
checkpoint = keras.callbacks.ModelCheckpoint(
    "xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5",
    save_best_only=True,
    monitor="val_accuracy"
)
```

Specifies the filename template for saving the models

Saves the model only when it's better than previous iterations

Uses the accuracy on validation for selecting the best model

The first parameter is a template for the filename. Let's take a look at it again:

```
"xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5"
```

It has two parameters inside:

- `{epoch:02d}` is replaced by the number of the epoch.
- `{val_accuracy:.3f}` is replaced by the validation accuracy.

Because we set `save_best_only` to `True`, `ModelCheckpoint` keeps track of the best accuracy and saves the results to disk each time the accuracy improves.

We implement `ModelCheckpoint` as a callback — a way to execute anything after each epoch finishes. In this particular case, the callback evaluates the model and saves the result if the accuracy gets better.

We can use it by passing it to the `callbacks` argument of the `fit` method:

```
model = make_model(learning_rate=0.001)

model.fit(
    train_ds,
    epochs=10,
    validation_data=val_ds,
    callbacks=[checkpoint]
)
```

Creates a new model

Specifies the list of callbacks to be used during training

After a few iterations, we already have some models saved to disk (figure 7.29).

Figure 7.29 Because the `ModelCheckpoint` callback saves the model only when it improves, we only have 4 files with our model, not 10.

We've learned how to store the best model. Now let's improve our model by adding more layers to the network.

7.4.7 Adding more layers

Previously, we trained a model with one dense layer:

```
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

outputs = keras.layers.Dense(10)(vector)

model = keras.Model(inputs, outputs)
```

We don't have to restrict ourselves to just one layer, so let's add another layer between the base model and the last layer with predictions (figure 7.30).

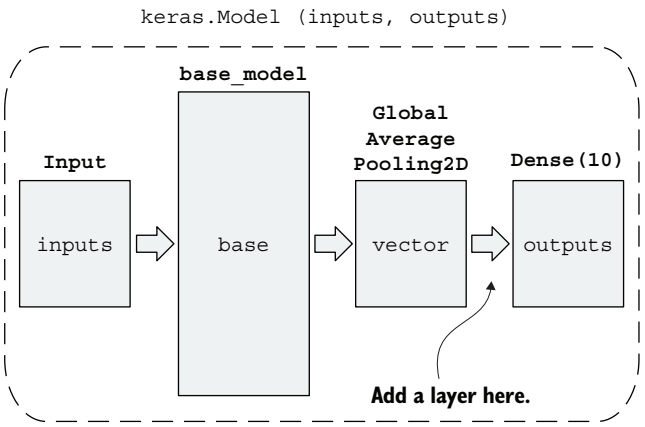


Figure 7.30 We add another dense layer between the vector representation and the output.

For example, we can add a dense layer of size 100:

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100, activation='relu')(vector)

outputs = keras.layers.Dense(10)(inner)

model = keras.Model(inputs, outputs)
```

Adds another dense layer of size 100

Instead of connecting outputs to vector, connects it to inner

NOTE There's no particular reason for selecting the size of 100 for the inner dense layer. We should treat it as a parameter: as with the learning rate, we can try different values and see which one leads to better performance on validation. In this chapter, we will not experiment with changing the size of the inner layer, but feel free to do so.

This way, we added a layer between the base model and the outputs (figure 7.31). Let's take another look at the line with the new dense layer:

```
inner = keras.layers.Dense(100, activation='relu')(vector)
```

Here, we set the `activation` parameter to `relu`.

Remember that we get a neural network by putting together multiple logistic regressions. In logistic regression, sigmoid is used for converting the raw score to probability.

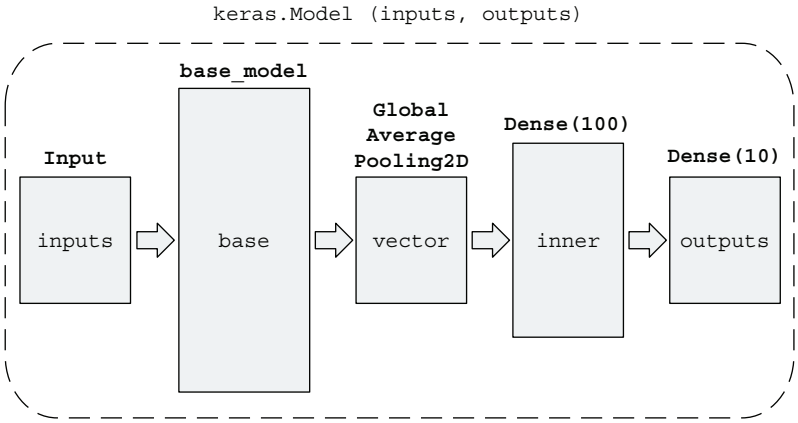


Figure 7.31 A new `inner` layer added between `vector` and `outputs`

But for inner layers, we don't need probabilities, and we can replace sigmoid with other functions. These functions are called *activation functions*. ReLU (Rectified Linear Unit) is one of them, and for inner layers, it's a better choice than sigmoid.

The sigmoid function suffers from the vanishing gradient problem, which makes training deep neural networks impossible. ReLU solves this problem. To read more about this problem, and about activation functions in general, please refer to the CS231n notes (https://cs231n.github.io/neural-networks-1/).

With another layer, our chances of overfitting increase significantly. To avoid that, we need to add regularization to our model. Next, we'll see how to do it.

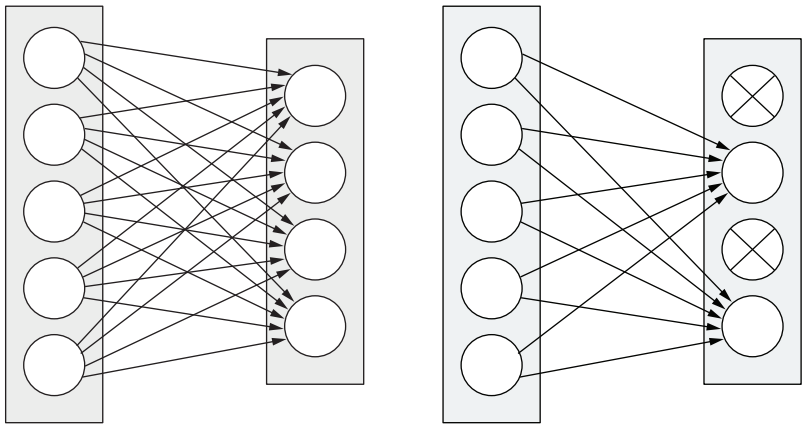
7.4.8 Regularization and dropout

Dropout is a special technique for fighting overfitting in neural networks. The main idea behind dropout is freezing a part of a dense layer when training. At each iteration, the part to freeze is chosen randomly. Only the unfrozen part is trained, and the frozen part is not touched at all.

If some parts of the network are ignored, the model overall is less likely to overfit. When the network goes over a batch of images, the frozen part of a layer doesn't see this data — it's turned off. This way, it's more difficult for the network to memorize the images (figure 7.32).

For every batch, the part to freeze is selected randomly, so the network learns to extract patterns from incomplete information, which makes it more robust and less likely to overfit.

We can control the strength of dropout by setting the dropout rate — the fraction of elements in a layer to be frozen at each step.



(A) Two dense layers without dropout (B) Two dense layers with dropout

Figure 7.32 With dropout, the connections to frozen nodes are dropped out.

To do this in Keras, we add a `Dropout` layer after the first `Dense` layer and set up the dropout rate:

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100, activation='relu')(vector)
drop = keras.layers.Dropout(0.2)(inner)
outputs = keras.layers.Dense(10)(drop)

model = keras.Model(inputs, outputs)
```

This way, we add another block in the network — the dropout block (figure 7.33).

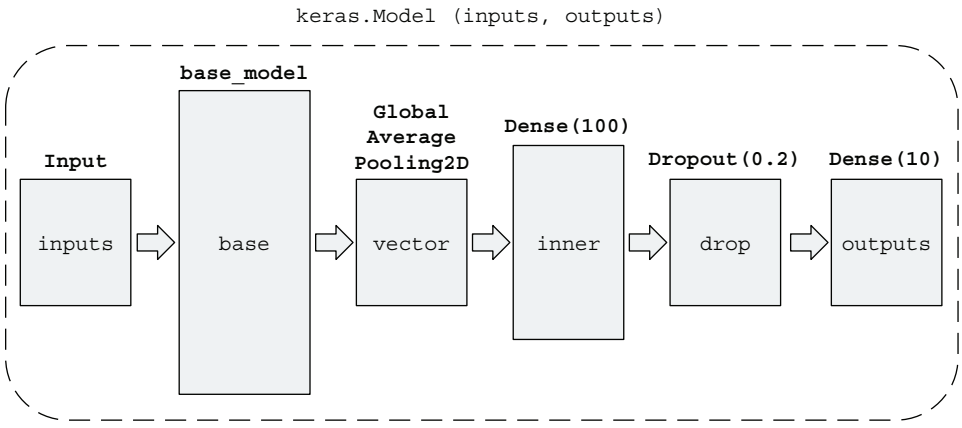


Figure 7.33 `Dropout` is another block between the `inner` layer and the `outputs` layer.

Let's train this model. To make it easier, we first need to update the `make_model` function and add another parameter there for controlling the dropout rate.

Listing 7.2 A function for creating a model with dropout

```
def make_model(learning_rate, droprate):
    base_model = Xception(
        weights='imagenet',
        input_shape=(150, 150, 3),
        include_top=False
    )

    base_model.trainable = False

    inputs = keras.Input(shape=(150, 150, 3))
    base = base_model(inputs, training=False)
    vector = keras.layers.GlobalAveragePooling2D()(base)

    inner = keras.layers.Dense(100, activation='relu')(vector)
    drop = keras.layers.Dropout(droprate)(inner)

    outputs = keras.layers.Dense(10)(drop)

    model = keras.Model(inputs, outputs)

    optimizer = keras.optimizers.Adam(learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=["accuracy"],
    )

    return model
```

Let's try four different values for the `droprate` parameter to see how the performance of our model changes:

- 0.0: Nothing gets frozen, so this is equivalent to not including the dropout layer at all.
- 0.2: Only 20% of the layer gets frozen,
- 0.5: Half of the layer is frozen.
- 0.8: Most of the layer (80%) is frozen.

With dropout, it takes more time to train a model: at each step, only a part of our network learns, so we need to make more steps. This means that we should increase the number of epochs when training.

So, let's train it:

```
model = make_model(learning_rate=0.001, droprate=0.0)
model.fit(train_ds, epochs=30, validation_data=val_ds)
```

Modifies droprate to experiment with different values

Trains a model for more epochs than previously

When it finishes, repeat this for other values of the `droprate` parameter by copying the code to another cell and changing the value to 0.2, 0.5, and 0.8.

From the results on the validation dataset, we see that there's no significant difference between 0.0, 0.2, and 0.5. However, 0.8 is worse — we made it really difficult for the network to learn anything (figure 7.34).

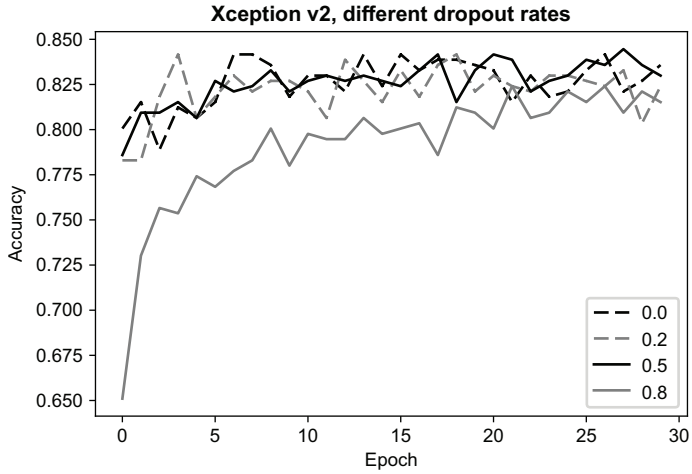


Figure 7.34 The accuracy on the validation set is similar for dropout rates of 0.0, 0.2, and 0.5. However, it's worse for 0.8.

The best accuracy we could achieve is 84.5% for the dropout rate of 0.5 (table 7.2).

Table 7.2 The validation accuracy with different values of dropout rate

Dropout rate	0.0	0.2	0.5	0.8
Validation accuracy	84.2%	84.2%	84.5%	82.4%

NOTE You may have different results, and it's possible that a different value for dropout rate achieves the best accuracy.

In cases like this, when there's no visible difference between accuracy on the validation dataset, it's useful to look at the accuracy on the train set as well (figure 7.35).

With no dropout, the model quickly memorizes the entire train dataset, and after 10 epochs, it becomes 99.9% accurate. With a dropout rate of 0.2, it needs more time

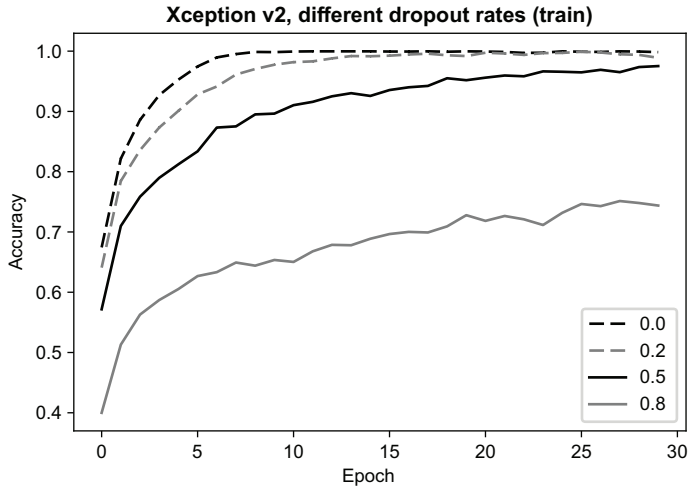


Figure 7.35 With a dropout rate of 0.0, the network overfits quickly, whereas the rate of 0.8 makes it really difficult to learn.

to overfit the training dataset, whereas for 0.5, it hasn't reached the perfect accuracy, even after 30 iterations. By setting the rate to 0.8, we make it really difficult for the network to learn anything, so the accuracy is low even on the training dataset.

We can see that with a dropout rate of 0.5, the network doesn't overfit as fast as others, while maintaining the same level of accuracy on the validation dataset as 0.0 and 0.2. Thus, we should prefer the model we trained with the dropout rate of 0.5 to other models.

By adding another layer and dropout, we have increased the accuracy from 83% to 84%. Even though this increase is not significant for this particular case, dropout is a powerful tool for fighting overfitting, and we should use it when making our models more complex.

In addition to dropout, we can use other ways to fight overfitting. For example, we can generate more data. In the next section, we'll see how to do it.

Exercise 7.2

In dropout, we

- a Remove a part of a model completely
- b Freeze a random part of a model, so it doesn't get updated during one iteration of training
- c Freeze a random part of a model, so it doesn't get used during the entire training process

7.4.9 Data augmentation

Getting more data is always a good idea, and it's usually the best thing we can do to improve the quality of our model. Unfortunately, it's not always possible to get more data.

For images, however, we can generate more data from existing images. For example:

- Flip an image vertically and horizontally.
- Rotate an image.
- Zoom in or out a bit.
- Change an image in other ways.

The process of generating more data from an existing dataset is called *data augmentation* (figure 7.36).

Figure 7.36 We can generate more training data by modifying existing images.

The easiest way to create a new image from an existing one is to flip it horizontally, vertically, or both (figure 7.37).

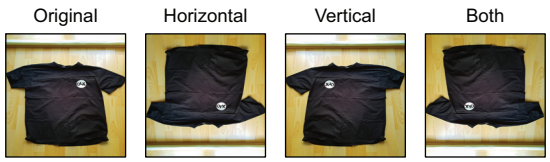


Figure 7.37 Flipping an image horizontally and vertically

In our case, horizontal flipping might not make much sense, but vertical flipping should be useful.

> **NOTE** If you're curious how these images are generated, check the notebook 07-augmentations.ipynb in the GitHub repository for this book.

Rotating is another image-manipulation strategy that we can use: we can generate a new image by rotating an existing one by some degree (figure 7.38).

Figure 7.38 Rotating an image. If the rotation degree is negative, the image is rotated counterclockwise.

Shear is another possible transformation. It skews the image by "pulling" it by one of its sides. When the shear is positive, we pull the right side down, and when it's negative, we pull the right side up (figure 7.39).

Figure 7.39 The shear transformation. We pull the image up or down by its right side.

At first glance, the effect of shear and rotation may look similar, but actually, they are quite different. Shear changes the geometrical shape of an image, but rotation doesn't: it only rotates an image (figure 7.40).

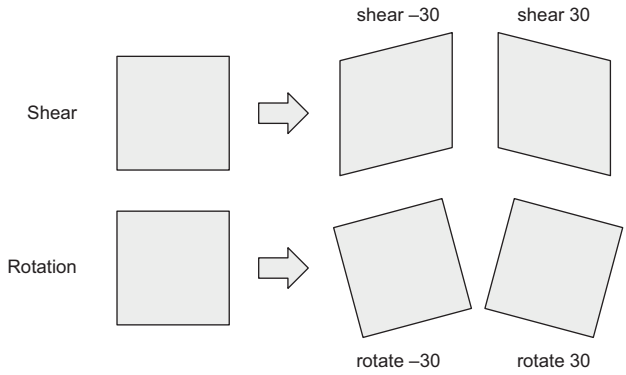


Figure 7.40 Shear changes the geometrical shape of an image by pulling it, so a square becomes a parallelogram. Rotation doesn't change the shape, so a square remains a square.

Next, we can shift an image horizontally (figure 7.41) or vertically (figure 7.42).

Figure 7.41 Shifting an image horizontally. Positive values shift the image to the left, whereas negative values shift it to the right.

Figure 7.42 Shifting an image vertically. Positive values shift the image to the top, whereas negative values shift it to the bottom.

Finally, we can zoom an image in or out (figure 7.43).

Figure 7.43 Zooming in or out. When the zoom factor is smaller than 1, we zoom in. If it's larger than 1, we zoom out.

What is more, we can combine multiple data augmentation strategies. For example, we can take an image, flip it horizontally, zoom out, and then rotate it.

By applying different augmentations to the same image, we can generate many more new images (figure 7.44).

Keras provides a built-in way of augmenting a dataset. It's based on `ImageDataGenerator`, which we have already used for reading the images.

The generator takes in many arguments. Previously, we used only `preprocessing_function` — it's needed to preprocess the images. Others are available, and many of them are responsible for augmenting the dataset.

Figure 7.44 10 new images generated from the same image

For example, we can create a new generator:

```
train_gen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=30.0,
    height_shift_range=30.0,
    shear_range=10.0,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=False,
    preprocessing_function=preprocess_input
)
```

Let's take a closer look at these parameters:

- `rotation_range=30`: Rotate an image by a random degree between –30 and 30.
- `width_shift_range=30`: Shift an image horizontally by a value between –30 and 30 pixels.
- `height_shift_range=30`: Shift an image vertically by a value between –30 and 30 pixels.
- `shear_range=10`: Apply the shear transformation by a value between –10 and 10 (also in pixels).
- `zoom_range=0.2`: Apply the zoom transformation using the zoom factor between 0.8 and 1.2 (1 – 0.2 and 1 + 0.2).
- `horizontal_flip=True`: Randomly flip an image horizontally.
- `vertical_flip=False`: Don't flip an image vertically.

For our project, let's take a small set of these augmentations:

```
train_gen = ImageDataGenerator(
    shear_range=10.0,
    zoom_range=0.1,
    horizontal_flip=True,
```

```
    preprocessing_function=preprocess_input,
)
```

Next, we use the generator in the same way as previously:

```
train_ds = train_gen.flow_from_directory(
    "clothing-dataset-small/train",
    target_size=(150, 150),
    batch_size=32,
)
```

We need to apply augmentations only to training data. We don't use it for validation: we want to make our evaluation consistent and be able to compare a model trained on the augmented dataset with a model trained without augmentations.

So, we load the validation dataset using exactly the same code as before:

```
validation_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

val_ds = validation_gen.flow_from_directory(
    "clothing-dataset-small/validation",
    target_size=image_size,
    batch_size=batch_size,
)
```

We're ready to train a new model now:

```
model = make_model(learning_rate=0.001, droprate=0.2)
model.fit(train_ds, epochs=50, validation_data=val_ds)
```

NOTE We omit the code for model checkpointing here for brevity. Add it if you want to save the best model.

To train this model, we need even more epochs than previously. Data augmentation is also a regularization strategy. Instead of training on the same image over and over again, the network sees a different variation of the same image for every epoch. This makes it more difficult for the model to memorize the data, and it decreases the chances of overfitting.

After training this model, we managed to improve the accuracy by 1%, from 84% to 85%.

This improvement is not really significant. But we have experimented a lot, and we could do this relatively quickly because we used small images of size 150 × 150. Now we can apply everything we have learned so far to larger images.

Exercise 7.3

Data augmentation helps fight overfitting because

a The model doesn't get to see the same images over and over again.
b It adds a lot of variety into the dataset — rotations and other image transformations.
c It generates examples of images that may exist, but the model overwise wouldn't have seen.
d All of the above.

7.4.10 Training a larger model

Even for people it may be challenging to understand what kind of item is in a small 150 × 150 image. It's also difficult for a computer: it's not easy to see the important details, so the model may confuse pants and shorts or T-shirts and shirts.

By increasing the size of images from 150 × 150 to 299 × 299, it'll be easier for the network to see more details and, therefore, achieve greater accuracy.

NOTE Training a model on larger images takes approximately four times longer than on small images. If you don't have access to a computer with a GPU, you don't have to run the code in this section. Conceptually, the process is the same, and the only difference is the input size.

So, let's modify our function for creating a model. For that, we need to take the code of `make_model` (listing 7.2) and adjust it in two places:

- The `input_shape` argument of Xception
- The `C` argument for input

In both these cases, we need to replace `(150, 150, 3)` with `(299, 299, 3)`.

Next, we need to adjust the `target_size` parameter for the train and validation generators. We replace `(150, 150)` by `(299, 299)`, and everything else stays the same.

Now we're ready to train a model!

```
model = make_model(learning_rate=0.001, droprate=0.2)
model.fit(train_ds, epochs=20, validation_data=val_ds)
```

NOTE To save the model, add checkpointing.

This model achieves accuracy of around 89% on the validation data. This is a considerable improvement over the previous model.

We have trained a model, so now it's time to use it.

7.5 Using the model

Previously, we trained multiple models. The best one is the model we trained on large images — it has 89% accuracy. The second-best model has an accuracy of 85%.

Let's now use these models to make predictions. To use a model, we first need to load it.

7.5.1 Loading the model

You can either use the model you trained yourself or download the model we trained for the book and use it.

To download them, go to the releases section of the book's GitHub repository and look for Models for Chapter 7: Deep learning (figure 7.45). Alternatively, go to this URL: https://github.com/alexeygrigorev/mlbookcamp-code/releases/tag/chapter7-model.

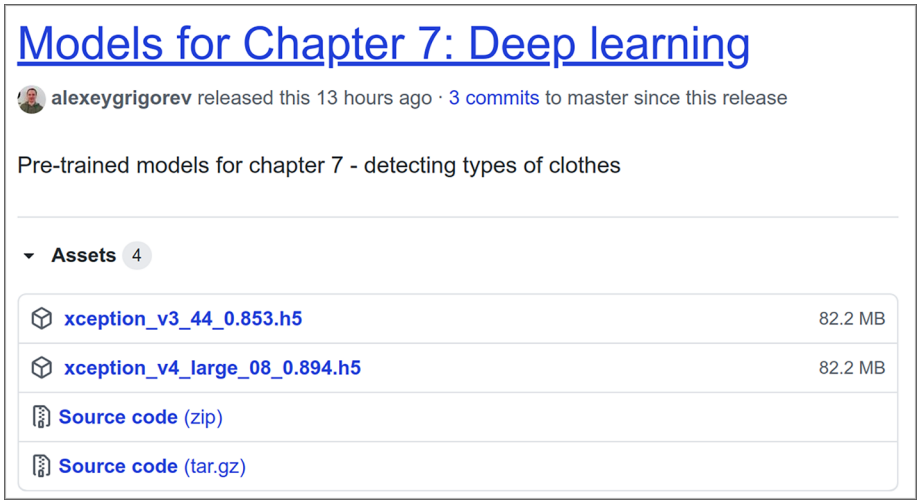


Figure 7.45 You can download the models we trained for this chapter from the book's GitHub repository.

Then download the large model trained on 299 × 299 images (xception_v4_large). To use it, load the model using the `load_model` function from the `models` package:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

We have already used both the train and validation datasets. We have finished the training process, so now it's time to evaluate this model on test data.

7.5.2 Evaluating the model

To load the test data, we follow the same approach: we use `ImageDataGenerator` but point to the test directory. Let's do it:

```
test_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

test_ds = test_gen.flow_from_directory(
    "clothing-dataset-small/test",
    shuffle=False,
    target_size=(299, 299),
    batch_size=32,
)
```

Use (150, 150) if you're using the small model.

Evaluating a model in Keras is as simple as invoking the `evaluate` method:

```
model.evaluate(test_ds)
```

It applies the model to all the data in the test folder and shows the evaluation metrics of loss and accuracy:

```
12/12 [==============================] - 70s 6s/step - loss: 0.2493 -
     accuracy: 0.9032
```

Our model shows 90% accuracy on the test dataset, which is comparable with the performance on the validation dataset (89%).

If we repeat the same process for the small dataset, we see that the performance is worse:

```
12/12 [==============================] - 15s 1s/step - loss: 0.6931 -
     accuracy: 0.8199
```

The accuracy is 82%, whereas on the validation it's 85%. The model performed worse on the test dataset.

It may be because of random fluctuations: the size of validation and test sets are not large, only 300 examples. So the model could get lucky on validation and unlucky on test.

However, it could be a sign of overfitting. By repeatedly evaluating the model on the validation dataset, we picked the model that got very lucky. Maybe this luck is not generalizable, and that's why the model performs worse on the data it hasn't seen previously.

Now let's see how we can apply the model to individual images to get predictions.

7.5.3 Getting the predictions

If we want to apply the model to a single image, we need to do the same thing `ImageDataGenerator` performs internally:

- Load an image.
- Preprocess it.

We already know how to load an image. We can use `load_img` for that:

```
path = 'clothing-dataset-small/test/pants/c8d21106-bbdb-4e8d-83e4-
     bf3d14e54c16.jpg'
img = load_img(path, target_size=(299, 299))
```

It's a picture of pants (figure 7.46).

```
img = load_img(path, target_size=(299, 299))
img
```

Figure 7.46 An image of pants from the train dataset

Next, we preprocess the image:

```
x = np.array(img)
X = np.array([x])
X = preprocess_input(X)
```

And, finally, we get the predictions:

```
pred = model.predict(X)
```

We can see the predictions for the image by checking the first row of predictions: `pred[0]` (figure 7.47).

```
pred = model.predict(X)
pred[0]
```

```
array([-2.8609202, -4.234048 , -1.5732546, -1.907885 , 10.247051 ,
       -2.2489133, -4.297381 ,  4.43905  , -4.4588056, -3.9616938],
      dtype=float32)
```

Figure 7.47 The predictions of our model. It's an array with 10 elements, one for each class.

The result is an array with 10 elements, where each element contains the score. The higher the score, the more likely the image is to belong to the respective class.

To get the element with the highest score, we can use the `argmax` method. It returns the index of the element with the highest score (figure 7.48).

```
pred[0].argmax()
```

```
4
```

Figure 7.48 The `argmax` function returns the element with the highest score.

To know which label corresponds to class 4, we need to get the mapping. It can be extracted from a data generator. But let's put it manually to a dictionary:

```
labels = {
    0: 'dress',
    1: 'hat',
    2: 'longsleeve',
    3: 'outwear',
    4: 'pants',
    5: 'shirt',
    6: 'shoes',
    7: 'shorts',
    8: 'skirt',
    9: 't-shirt'
}
```

To get the label, simply look it up in the dictionary:

```
labels[pred[0].argmax()]
```

As we see, the label is "pants," which is correct. Also note that the label "shorts" has a high positive score: pants and shorts are quite similar visually. But "pants" is clearly the winner.

We will use this code in the next chapter, where we will productionize our model.

7.6 Next steps

We've learned the basics we need for training a classification model for predicting the type of clothes. We've covered a lot of material, but there is a lot more to learn than we could fit into this chapter. You can explore this topic more by doing the exercises.

7.6.1 Exercises

- For deep learning, the more data we have, the better. But the dataset we used for this project is not large: we trained our model on only 3,068 images. To make it better, we can add more training data. You can find more pictures of clothes in other data sources; for example, at https://www.kaggle.com/dqmonn/zalando-store-crawl, https://www.kaggle.com/paramaggarwal/fashion-product-images-dataset, or https://www.kaggle.com/c/imaterialist-fashion-2019-FGVC6. Try adding more pictures to the training data, and see if it improves the accuracy on the validation dataset.
- Augmentations help us train better models. In this chapter, we used only the most basic augmentation strategies. You can further explore this topic and try other kinds of image modifications. For example, add rotations and shifting, and see if it helps the model achieve better performance.
- In addition to the built-in way of augmenting the dataset, we have special libraries for doing this. One of them is Albumentations (https://github.com/albumentations-team/albumentations), which contains many more image manipulation algorithms. You can also experiment with it and see which augmentations work well for this problem.
- Many different pretrained models are available. We used Xception, but many others are out there. You can try them and see if they give a better performance. With Keras, it's quite simple to use a different model: just import from a different package. For example, you can try ResNet50 and compare it with the results from Xception. Check the documentation for more information (https://keras.io/api/applications/).

7.6.2 Other projects

There are many image classification projects that you can do:

- Cats or dogs (https://www.kaggle.com/c/dogs-vs-cats)
- Hotdog or not hotdog (https://www.kaggle.com/dansbecker/hot-dog-not-hot-dog)
- Predicting the category of images from Avito's dataset (online classifieds) (https://www.kaggle.com/c/avito-duplicate-ads-detection). Note that many duplicates appear in this dataset, so be careful when splitting the data for validation. It might be a good idea to use the train/test split that the organizators prepared and do a bit of extra cleaning to make sure there are no duplicate images.

Summary

- TensorFlow is a framework for building and using neural networks. Keras is a library on top of TensorFlow that makes training models simpler.
- For image processing, we need a special kind of neural network: a convolutional neural network. It consists of a series of convolutional layers followed by a series of dense layers.
- The convolutional layers in a neural network convert an image to its vector representation. This representation contains high-level features. The dense layers use these features to make the prediction.
- We don't need to train a convolutional neural network from scratch. We can use pretrained models on ImageNet for general-purpose classification.
- Transfer learning is the process of adjusting a pretrained model to our problem. We keep the original convolutional layers but create new dense layers. This significantly reduces the time we need to train a model.
- We use dropout to prevent overfitting. At each iteration, it freezes a random part of the network, so only the other part can be used for training. This allows the network to generalize better.
- We can create more training data from existing images by rotating them, flipping them vertically and horizontally, and doing other transformations. This process is called data augmentation, and it adds more variability to the data and reduces the risk of overfitting.

In this chapter, we have trained a convolutional neural network for classifying images of clothing. We can save it, load it, and use it inside a Jupyter Notebook. This is not enough to use it in production.

In the next chapter, we show how to use it in production and talk about two ways of productionizing deep learning models: TensorFlow Lite in AWS Lambda and TensorFlow Serving in Kubernetes.

Answers to exercises

- Exercise 7.1 A) True
- Exercise 7.2 B) Freeze a random part of a model, so it doesn't get updated during one iteration of training.
- Exercise 7.3 D) All of the above

8 Serverless deep learning

This chapter covers

- Serving models with TensorFlow Lite — a lightweight environment for applying TensorFlow models
- Deploying deep learning models with AWS Lambda
- Exposing the lambda function as a web service via API Gateway

In the previous chapter, we trained a deep learning model for categorizing images of clothing. Now we need to deploy it, making the model available for other services.

We have many possible ways of doing this. We have already covered the basics of model deployment in chapter 5, where we talked about using Flask, Docker, and AWS Elastic Beanstalk for deploying a logistic regression model.

In this chapter, we'll talk about the serverless approach for deploying models — we'll use AWS Lambda.

8.1 Serverless: AWS Lambda

AWS Lambda is a service from Amazon. Its main promise is that you can "run code without thinking about servers."

It lives up to the promise: in AWS Lambda, we just need to upload some code. The service takes care of running it and scales it up and down according to the load.

Additionally, you only need to pay for the time when the function is actually used. When nobody uses the model and invokes our service, you don't pay for anything.

In this chapter, we use AWS Lambda for deploying the model we trained previously. For doing that, we'll also use TensorFlow Lite — a lightweight version of TensorFlow that has only the most essential functions.

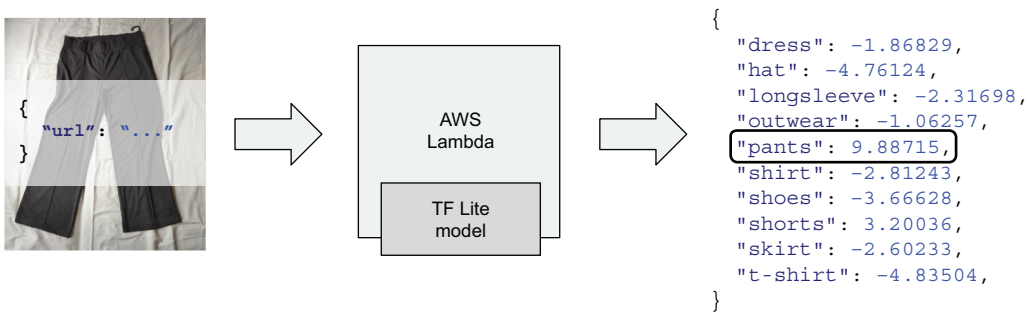


Figure 8.1 Overview of the service: it gets the URL of an image, applies the model, and returns the predictions.

We want to build a web service that

- Gets the URL in the request
- Loads the image from this URL
- Uses TensorFlow Lite to apply the model to the image and get the predictions
- Responds with the results (figure 8.1)

To create this service, we will need to

- Convert the model from Keras to the TensorFlow Lite format
- Preprocess the images — resize them and apply the preprocessing function
- Package the code in a Docker image, and upload it to ECR (the Docker registry from AWS)
- Create and test the lambda function on AWS
- Make the lambda function available to everyone with AWS API Gateway

We assume you have an AWS account and have configured the AWS CLI tool. For details, please refer to appendix A.

> **NOTE** At the time of writing, AWS Lambda is covered by the AWS free tier. This means that it's possible to do all the experiments in this chapter for free. To check the conditions, refer to the AWS documentation (https:// aws.amazon.com/free/).

We use AWS here, but this approach also works for other serverless platforms.

The code for this chapter is available in the book's GitHub repository (https://github.com/alexeygrigorev/mlbookcamp-code/) in the chapter-08-serverless folder.

Let's start by discussing TensorFlow Lite.

8.1.1 TensorFlow Lite

TensorFlow is a great framework with a rich set of features. However, most of these features are not needed for model deployment, and they take up a lot of space: when compressed, TensorFlow takes up more than 1.5 GB of space.

TensorFlow Lite (usually abbreviated as "TF Lite"), on the other hand, takes up only 50 MB of space. It's optimized for mobile devices and contains only the essential parts. With TF Lite, you can use the model only for making predictions and nothing else, including training new models.

Even though it was originally created for mobile devices, it's applicable for more cases. We can use it whenever we have a TensorFlow model but can't afford to take the entire TensorFlow package with us.

> **NOTE** The TF Lite library is under active development and changes rather quickly. It's possible that the way you install this library has changed since the book was published. Please refer to the official documentation for up-to-date instructions (https://www.tensorflow.org/lite/guide/python).

Let's install the library now. We can do so with `pip`:

```
pip install --extra-index-url https://google-coral.github.io/py-repo/
     tflite_runtime
```

When running `pip install`, we add the `extra-index-url` parameter. The library we install is not available in the central repository with Python packages, but it's available in a different repository. We need to point to this repository.

> **NOTE** For non-Debian-based Linux distributions, like CentOS, Fedora, or Amazon Linux, the library installed this way might not work: you might get an error when trying to import the library. If that's the case, you need to compile this library yourself. Refer to the instructions here for more details: https://github.com/alexeygrigorev/serverless-deep-learning. For MacOS and Windows, it should work as expected.

TF Lite uses a special optimized format for storing models. To use our model with TF Lite, we need to convert our model to this format. We'll do that next.

8.1.2 Converting the model to TF Lite format

We used the h5 format for saving the model from the previous chapter. This format is suitable for storing Keras models, but it won't work for TF Lite. So, we need to convert our model to TF-Lite format.

If you don't have the model from the previous chapter, go ahead and download it:

```
wget https://github.com/alexeygrigorev/mlbookcamp-code/releases/download/
     chapter7-model/xception_v4_large_08_0.894.h5
```

Now let's create a simple script for converting this model — convert.py.

First, start with imports:

```
import tensorflow as tf
from tensorflow import keras
```

Next, load the Keras model:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

And finally, convert it to TF Lite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)

tflite_model = converter.convert()

with tf.io.gfile.GFile('clothing-model-v4.tflite', 'wb') as f:
    f.write(tflite_model)
```

Let's run it:

```
python convert.py
```

After running it, we should have a file named clothing-model-v4.tflite in our directory.

We're ready to use this model now for image classification, applying the model to images of clothing to understand if a given image is a T-shirt, pants, a skirt, or something else. However, remember that before we can use a model for classifying an image, the image needs to be preprocessed. We'll see how to do that next.

8.1.3 Preparing the images

Previously, when testing the model in Keras, we preprocessed each image using the `preprocess_input` function. This is how we imported it in the previous chapter:

```
from tensorflow.keras.applications.xception import preprocess_input
```

And then we applied this function to images before we put them into models.

However, we can't use the same function when deploying our model. This function is a part of the TensforFlow package, and there's no equivalent in TF Lite. We don't want to depend on TensorFlow just for this simple preprocessing function.

Instead, we can use a special library that has only the code we need: `keras_image_helper`. This library was written to simplify the explanation in this book. If you want to know how images are pre-processed in more detail, check the source code. It's available at https://github.com/alexeygrigorev/keras-image-helper. This library can load an image, resize it, and apply other preprocessing transformations that Keras models require.

Let's install it with `pip`:

```
pip install keras_image_helper
```

Next, open Jupyter, and create a notebook called chapter-08-model-test.

We start by importing the `create_preprocessor` function from the library:

```
from keras_image_helper import create_preprocessor
```

The function `create_preprocessor` takes two arguments:

- `name`: The name of the model. You can see the list of available models at https://keras.io/api/applications/.
- `target_size`: The size of the image that the neural network expects to get.

We used the Xception model, and it expects an image of size 299 × 299. Let's create a preprocessor for our model:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Now let's get a picture of pants (figure 8.2), and prepare it:

```
image_url = 'http://bit.ly/mlbookcamp-pants'
X = preprocessor.from_url(image_url)
```

Figure 8.2 The picture of pants that we use for testing

The result is a NumPy array of shape (1, 299, 299, 3):

- It's a batch of one image only.
- 299 × 299 is the size of the image.
- There are three channels: red, green, and blue.

We have prepared the image, and we're ready to use the model for classifying it. Let's see how we can do this with TF Lite.

8.1.4 Using the TensorFlow Lite model

We have the array `X` from the previous step, and now we can use TF Lite for classifying it.

First, import TF Lite:

```
import tflite_runtime.interpreter as tflite
```

Load the model we previously converted:

```
interpreter = tflite.Interpreter(model_path='clothing-model-v4.tflite')
interpreter.allocate_tensors()
```

Creates the TF Lite interpreter

Initializes the interpreter with the model

To be able to use the model, we need to get its input (where `X` will go) and the output (where we get the predictions from):

```
input_details = interpreter.get_input_details()
input_index = input_details[0]['index']
```

Gets the input: the part of the network that takes in the array X

```
output_details = interpreter.get_output_details()
output_index = output_details[0]['index']
```

Gets the output: the part of the network with final predictions

To apply the model, take the `X` we previously prepared, put it into the input, invoke the interpreter, and get the results from the output:

```
interpreter.set_tensor(input_index, X)
interpreter.invoke()

preds = interpreter.get_tensor(output_index)
```

Puts X into the input

Runs the model to get predictions

Gets the predictions from the output

The `preds` array contains the predictions:

```
array([[-1.8682897, -4.7612453, -2.316984 , -1.0625705,  9.887156 ,
        -2.8124316, -3.6662838,  3.2003622, -2.6023388, -4.8350453]],
      dtype=float32)
```

Now we can do the same thing with it as previously — assign the label to each element of this array:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

results = dict(zip(labels, preds[0]))
```

It's done! We have the predictions in the `results` variable:

```
{'dress': -1.8682897,
 'hat': -4.7612453,
 'longsleeve': -2.316984,
 'outwear': -1.0625705,
 'pants': 9.887156,
 'shirt': -2.8124316,
 'shoes': -3.6662838,
 'shorts': 3.2003622,
 'skirt': -2.6023388,
 't-shirt': -4.8350453}
```

We see that the `pants` label has the highest score, so this must be a picture of pants.

Let's now use this code for our future AWS Lambda function!

8.1.5 Code for the lambda function

In the previous section, we wrote all the code we need for the lambda function. Let's put it together in a single script — lambda_function.py.

As usual, start with imports:

```
import tflite_runtime.interpreter as tflite
from keras_image_helper import create_preprocessor
```

Then, create the preprocessor:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Next, load the model, and get the output and input:

```
interpreter = tflite.Interpreter(model_path='clothing-model-v4.tflite')
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
input_index = input_details[0]['index']
```

```
output_details = interpreter.get_output_details()
output_index = output_details[0]['index']
```

To make it a bit cleaner, we can put all the code for making a prediction together in one function:

```
def predict(X):
    interpreter.set_tensor(input_index, X)
    interpreter.invoke()
    preds = interpreter.get_tensor(output_index)
    return preds[0]
```

Next, let's make another function for preparing the results:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

def decode_predictions(pred):
    result = {c: float(p) for c, p in zip(labels, pred)}
    return result
```

Finally, put everything together in one function — `lambda_handler` — which is the function invoked by the AWS Lambda environment. It will use all the things we defined previously:

```
def lambda_handler(event, context):
    url = event['url']
    X = preprocessor.from_url(url)
    preds = predict(X)
    results = decode_predictions(preds)
    return results
```

In this case, the `event` parameter contains all the information we pass to the lambda function in our request (figure 8.3). The `context` parameter is typically not used.

We're ready to test it now! To do it locally, we need to put this code into the Python Docker container for AWS Lambda.

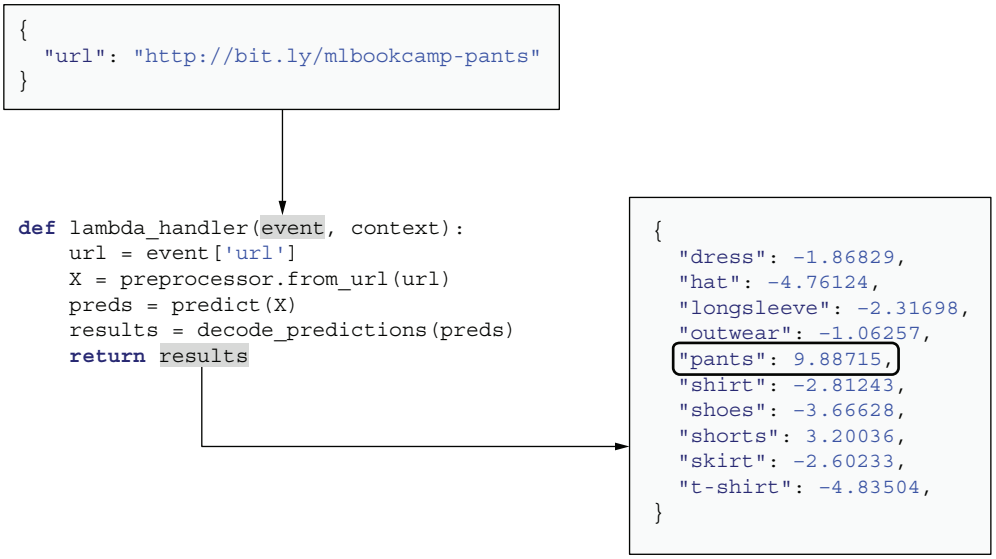


Figure 8.3 The input and the output of the lambda function: the input goes to the `event` parameter, and the predictions are returned as the output.

8.1.6 Preparing the Docker image

First, create a file named Dockerfile:

```
FROM public.ecr.aws/lambda/python:3.7

RUN pip3 install keras_image_helper --no-cache-dir
RUN pip3 install https://raw.githubusercontent.com/alexeygrigorev/serverless-
deep-learning/master/tflite/tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl
--no-cache-dir

COPY clothing-model-v4.tflite clothing-model-v4.tflite
COPY lambda_function.py lambda_function.py

CMD [ "lambda_function.lambda_handler" ]
```

1 Uses the official Docker image
2 Installs keras_image_helper
3 Installs TF Lite
4 Copies the model
5 Copies the lambda function
6 Defines the location of the lambda function

Let's take a look at each line of the file. First 1, we use the official Python 3.7 Docker image for Lambda from AWS. You can see other available images here: https://gallery.ecr.aws/. Then 2, we install the keras_image_helper library.

Next 3, we install a special version of TF Lite that was compiled to work with Amazon Linux. The installation instructions we used in this chapter earlier don't work for Amazon Linux, only for Ubuntu (and other Debian-based distributions). That's why we need to use a special version. You can read more about it here: https://github.com/alexeygrigorev/serverless-deep-learning.

Then ❹, we copy the model to the image. When we do so, the model becomes a part of the image. This way, it's simpler to deploy the model. We could use an alternative approach — the model can be put to S3 and loaded when the script starts. It's more complex but also more flexible. For the book, we went with the simpler approach.

Then ❺, we copy the code of the lambda function we prepared earlier.

Finally ❻, we tell the `lambda` environment that it needs to look for the file named lambda_function and look for the function `lambda_handler` inside this function. This is the function we prepared in the previous section.

Let's build this image:

```
docker build -t tf-lite-lambda .
```

Next, we need to check that the lambda function works. Let's run the image:

```
docker run --rm -p 8080:8080 tf-lite-lambda
```

It's running! We can test it now.

We can continue using the Jupyter Notebook we created earlier, or we can create a separate Python file named test.py. It should have the following content — and you'll note it's very similar to the code we wrote in chapter 5 for testing our web service:

```
import requests

data = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = "http://localhost:8080/2015-03-31/functions/function/invocations"

results = requests.post(url, json=data).json()
print(results)
```

❶ Prepares the request

Specifies the URL ❷

❸ Sends a POST request to the service

First, we define the `data` variable in ❶ — this is our request. Then we specify the URL of the service in ❷ — this is the location where the function is currently deployed. Finally, in ❸, we use the POST method to submit the request and get back the predictions in the `results` variable.

When we run it, we get the following response:

```
{
  "dress": -1.86829,
  "hat": -4.76124,
  "longsleeve": -2.31698,
  "outwear": -1.06257,
  "pants": 9.88715,
  "shirt": -2.81243,
  "shoes": -3.66628,
  "shorts": 3.20036,
```

```
  "skirt": -2.60233,
  "t-shirt": -4.83504
}
```

The model works!

We're almost ready to deploy it to AWS. To do that, we first need to publish this image to ECR — the Docker container registry from AWS.

8.1.7 Pushing the image to AWS ECR

To publish this Docker image to AWS, we first need to create a registry using the AWS CLI tool:

```
aws ecr create-repository --repository-name lambda-images
```

It will return back an URL that looks like this:

```
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/lambda-images
```

You'll need this URL.

Alternatively, it's possible to create the registry using the AWS Console.

Once the registry is created, we need to push the image there. Because this registry belongs to our account, we first need to authenticate our Docker client. On Linux and MacOS, you can do this:

```
$(aws ecr get-login --no-include-email)
```

On Windows, run `aws ecr get-login --no-include-email`, copy the output, enter it into the terminal, and execute it manually.

Now let's use the registry URL to push the image to ECR:

```
REGION=eu-west-1
ACCOUNT=XXXXXXXXXXX
REMOTE_NAME=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/lambda-images:tf-lite-
    lambda
docker tag tf-lite-lambda ${REMOTE_NAME}
docker push ${REMOTE_NAME}
```

Specify the region and your AWS account ID.

Now it's pushed, and we can use it to create a lambda function in AWS.

8.1.8 Creating the lambda function

This step is easier to do with the AWS Console, so open it, go to services, and select Lambda.

Next, click Create Function. Select Container Image (figure 8.4).

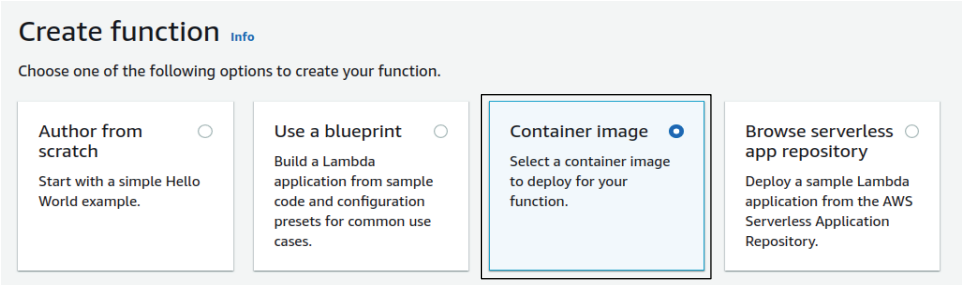


Figure 8.4 When creating a `lambda` function, select Container Image.

After that, fill in the details (figure 8.5).

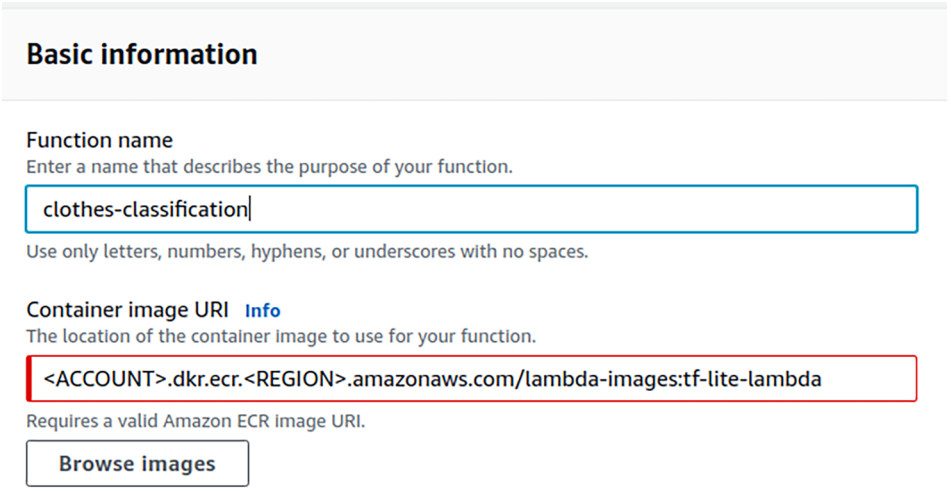


Figure 8.5 Enter the function name and the container image URI.

The container image URI should be the image we created earlier and pushed to ECR:

```
<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/lambda-images:tf-lite-lambda
```

You can use the Browse Images button to find it (figure 8.5). Keep the rest unchanged, and click Create Function. The function is created!

Now we need to give our function more memory and let it run for a longer time without timing out. For that, select the Configuration tab, choose General Configuration, and then click Edit (figure 8.6).

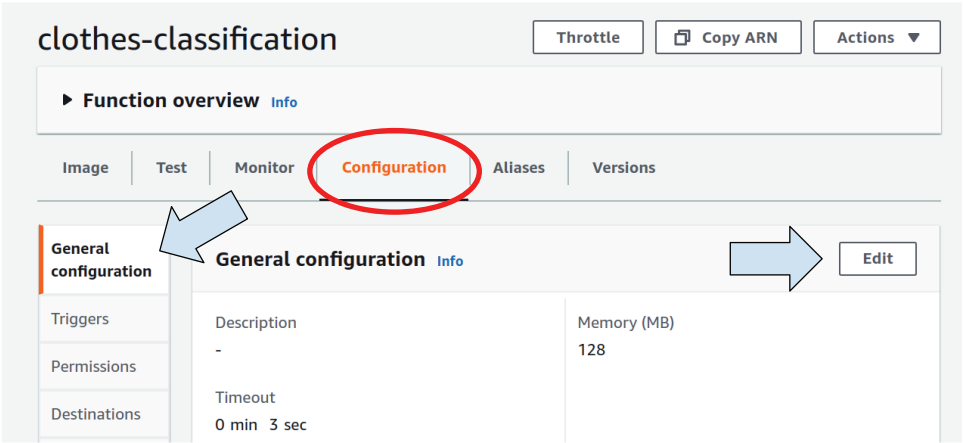


Figure 8.6 The default settings of a lambda function: the default amount of memory (128 MB) is not enough, so we need to increase it. Click Edit to do so.

The default settings are not good for deep learning models. We need to configure this function to give it more RAM and allow it to take more time.

For that, click the Edit button, give it 1024 MB of RAM, and set the timeout to 30 seconds (figure 8.7).

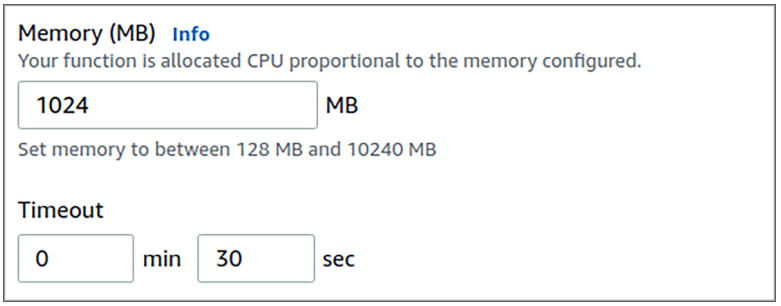


Figure 8.7 Increase the amount of memory to 1024 MB and set the timeout to 30 seconds.

Save it.

It's ready! To test it, go to the Test tab (figure 8.8).

It'll suggest creating a test event. Give it a name (for example, test), and put the following content in the request body:

```
{
    "url": "http://bit.ly/mlbookcamp-pants"
}
```

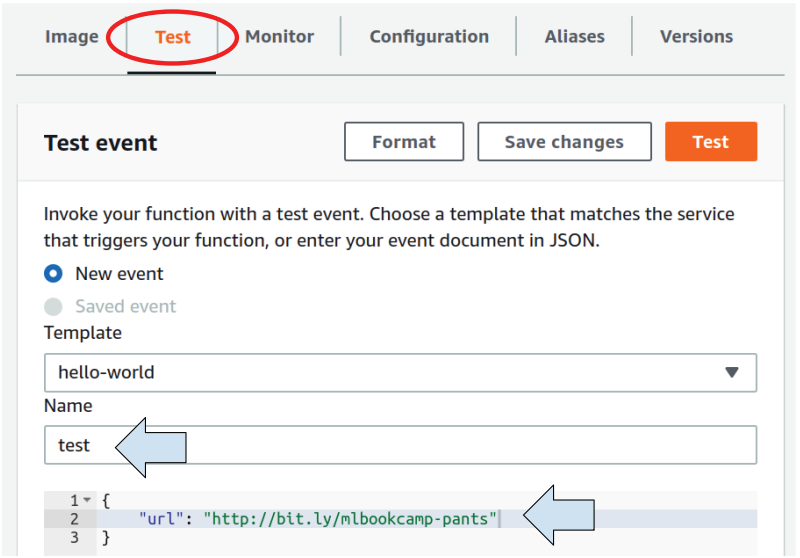


Figure 8.8 The Test button is located at the top of the screen. Click it to test the function.

Save it and click the Test button again. After approximately 15 seconds, you should see "Execution results: succeeded" (figure 8.9).

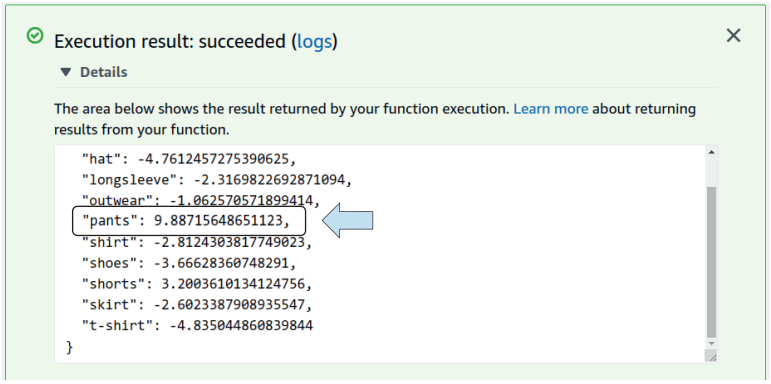


Figure 8.9 The predictions of our model. The prediction for `pants` has the highest score.

When we run the test for the first time, it needs to pull the image from ECR, load all the libraries in memory, and do some other things to "warm up." But once it's done it, the consequent invocations take less time — approximately two seconds for this model.

We have successfully deployed our model to AWS Lambda, and it's working!

Also, remember that you pay only when the function is invoked, so you don't need to worry about deleting this function if it's not used. And you don't need to worry about managing EC2 instances at all — AWS Lambda takes care of everything for us.

It's already possible to use this model for many things: AWS Lambda integrates well with a lot of other services from AWS. But if we want to use it as a web service and send requests over HTTP, we need to expose it through API Gateway.

We'll see how to do this next.

8.1.9 Creating the API Gateway

In the AWS Console, find the API Gateway service. Create a new API: select REST API, and click Build.

Then select New API, and call it clothes-classification (figure 8.10). Click Create API.

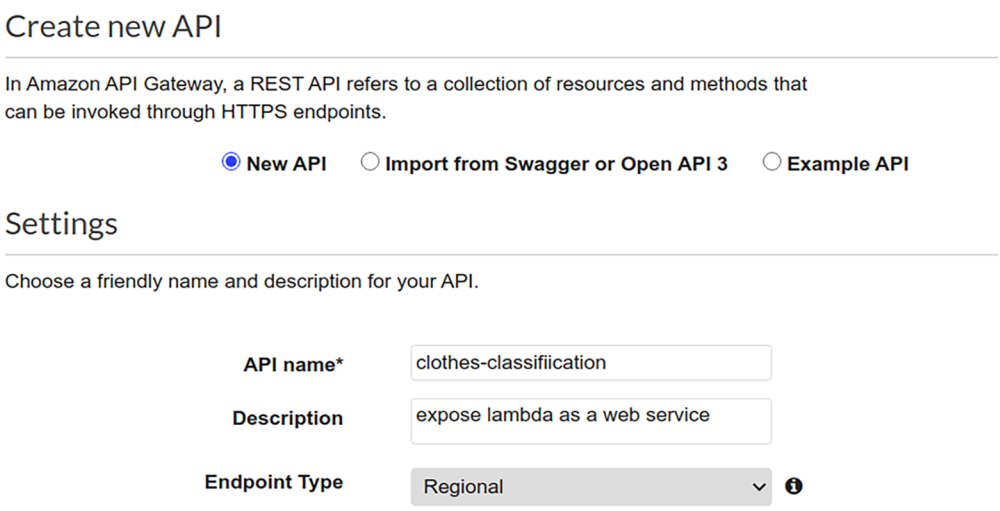


Figure 8.10 Creating a new REST API Gateway in AWS

Next, click the Actions button and select Resource. Then, create a resource predict (figure 8.11).

> **NOTE** The name predict doesn't follow the REST naming standards: usually resources should be nouns. However, it's common to name endpoints for predictions as predict; that's why we don't follow the REST convention.

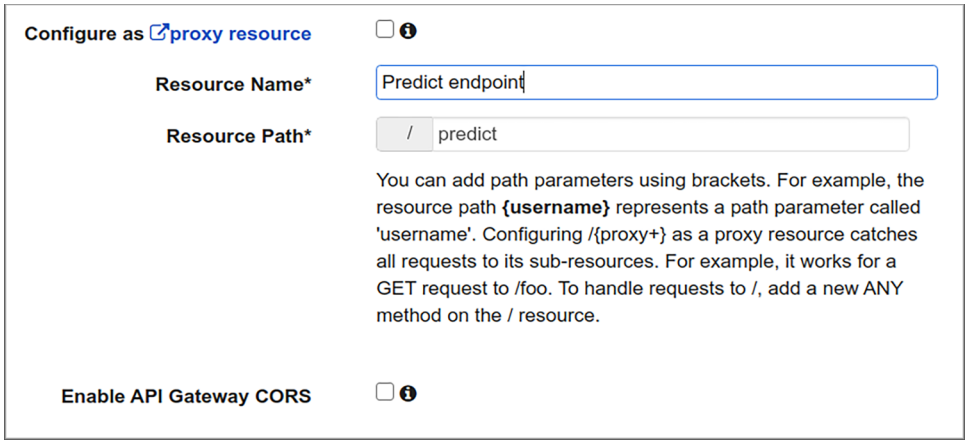


Figure 8.11 Creating a predict resource

After creating the resource, create a POST method for it (figure 8.12):

1. Click Predict.
2. Click Actions.
3. Select Create Method.
4. Choose POST from the list.
5. Click the tick button.

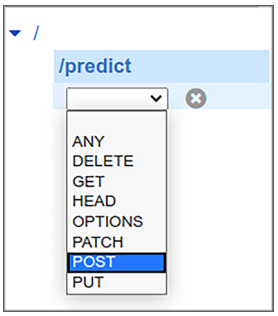


Figure 8.12 Create a POST method for the predict resource.

We're almost ready!

Now select Lambda Function as the integration type and enter the name of your lambda function (figure 8.13).

NOTE Make sure you don't use proxy integration — this checkbox should remain unchecked. If you use this option, API Gateway adds some extra information to the request, and we would need to adjust the lambda function.

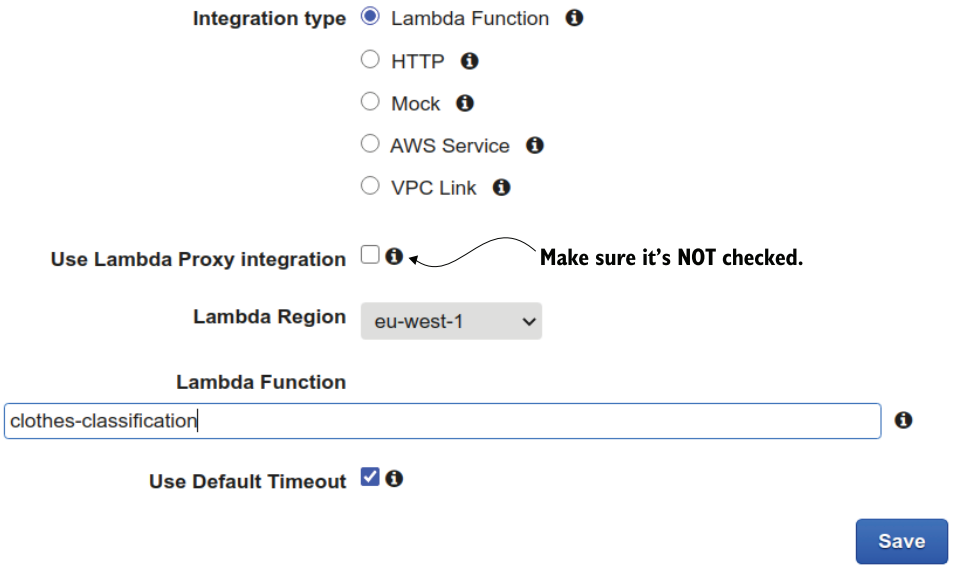


Figure 8.13 Configuring the POST action for the predict resource. Make sure Proxy Integration is not checked.

After doing this, we should see the integration (figure 8.14).

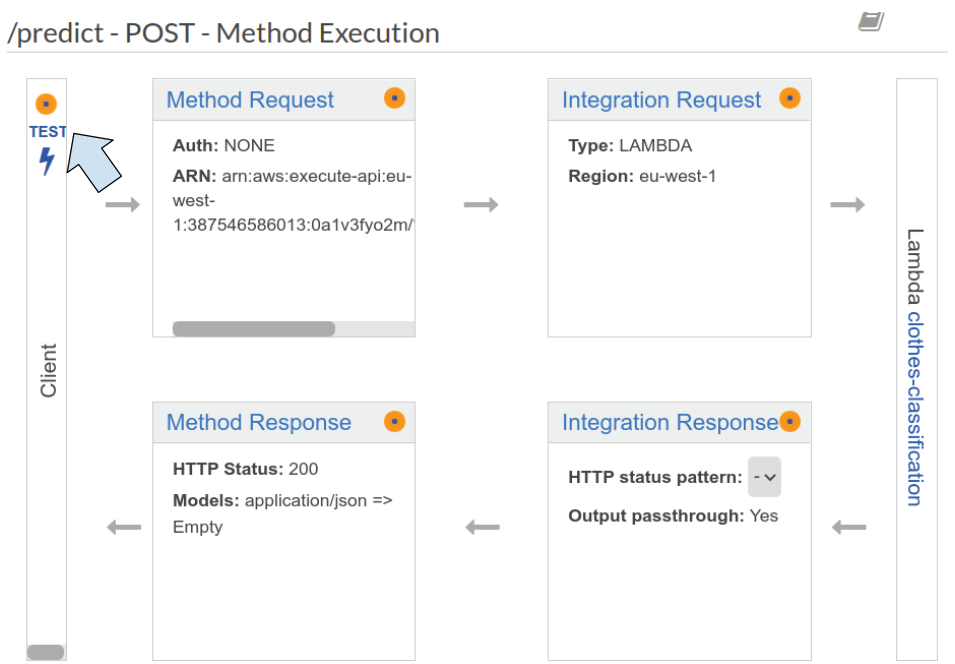


Figure 8.14 Deploying the API

Let's test it. Click TEST, and put the same request in the request body as previously:

```
{
    "url": "http://bit.ly/mlbookcamp-pants"
}
```

The response is the same: the predicted class is `pants` (figure 8.15).

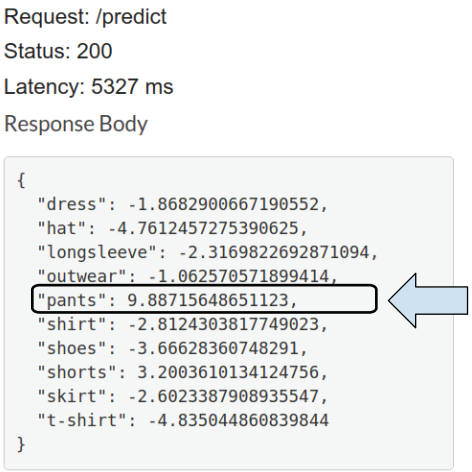


Figure 8.15 The response from the lambda function. The `pants` category has the highest score.

To use it externally, we need to deploy the API. Select Deploy API from the list of actions (figure 8.16).

Next, create a new stage test (figure 8.17).

By clicking Deploy, we deploy the API. Now find the Invoke URL field. It should look like this:

https://0a1v3fyo2m.execute-api.eu-west-1.amazonaws.com/test

All we need to do now to invoke the lambda function is to add "/predict" at the end of this URL.

Let's take the test.py script we created previously and replace the URL there:

```
import requests

data = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = "https://0a1v3fyo2m.execute-api.eu-west-1.amazonaws.com/test/predict"

results = requests.post(url, json=data).json()

print(results)
```

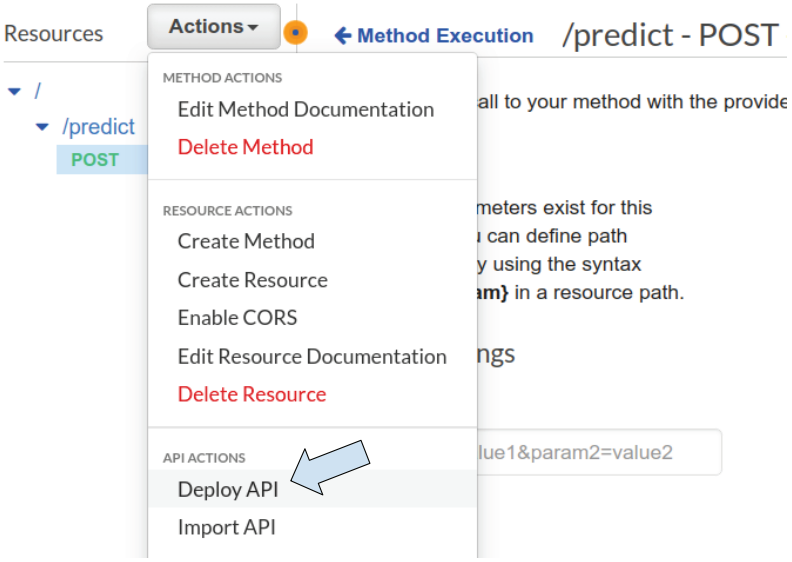


Figure 8.16 The function `clothes-classification` is now connected to the POST method of the predict resource in the API Gateway. The TEST button helps with verifying that the connection with lambda works.

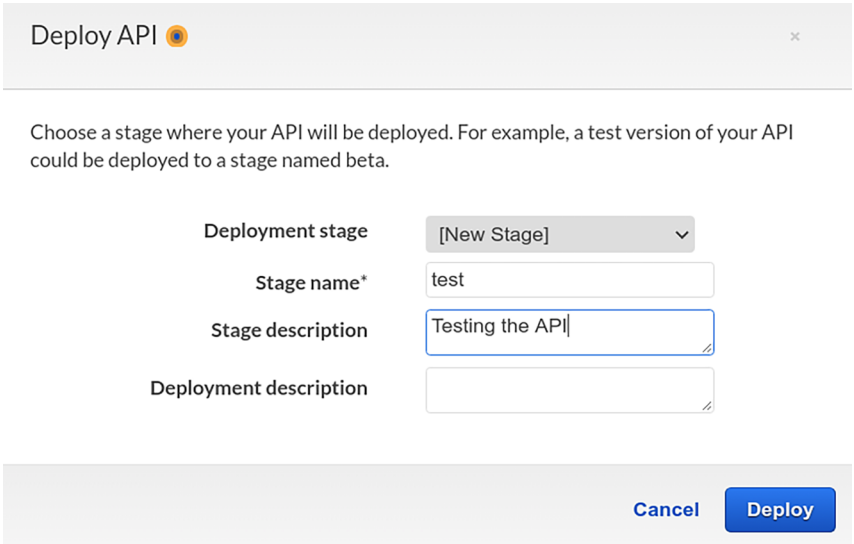


Figure 8.17 Configuring the stage for the API

Run it:

```
python test.py
```

The response is the same as previously:

```
{
  "dress": -1.86829,
  "hat": -4.76124,
  "longsleeve": -2.31698,
  "outwear": -1.06257,
  "pants": 9.88715,
  "shirt": -2.81243,
  "shoes": -3.66628,
  "shorts": 3.20036,
  "skirt": -2.60233,
  "t-shirt": -4.83504
}
```

Now our model is exposed with a web service that we can use from anywhere.

8.2 Next steps

8.2.1 Exercises

Try the following to further explore the topics of serverless model deployment:

- AWS Lambda is not the only serverless environment. You can also experiment with cloud functions in Google Cloud and Azure functions on Azure.
- SAM (Serverless Application Model) is a tool from AWS for making the process of creating AWS Lambda functions easier (https://aws.amazon.com/serverless/sam/). You can use it to reimplement the project from this chapter.
- Serverless (https://www.serverless.com/) is a framework similar to SAM. It's not specific to AWS and works for other cloud providers. You can experiment with it and deploy the project from this chapter.

8.2.2 Other projects

You can do many other projects:

- AWS Lambda is a convenient platform for hosting machine learning models. In this chapter, we deployed a deep learning model. You can also experiment with it more and deploy the models we trained in the previous chapters as well as the models you developed as a part of the exercises.

Summary

- TensorFlow Lite is a lightweight alternative to "full" TensorFlow. It contains only the most important parts that are needed for using deep learning models. Using it makes the process of deploying models with AWS Lambda faster and simpler.

- Lambda functions can be run locally using Docker. This way, we can test our code without deploying it to AWS.
- To deploy a lambda function, we need to put its code in Docker, publish the Docker image to ECR, and then use the URI of the image when creating a lambda function.
- To expose the lambda function, we use API Gateway. This way, we make the lambda function available as a web service, so it could be used by anyone.

In this chapter, we've used AWS Lambda — the serverless approach for deploying deep learning models. We didn't want to worry about servers and let the environment worry about it instead.

In the next chapter, we actually think about servers, and we use a Kubernetes cluster for deploying a model.

9 Serving models with Kubernetes and Kubeflow

This chapter covers

- Understanding different methods of deploying and serving models in the cloud
- Serving Keras and TensorFlow models with TensorFlowServing
- Deploying TensorFlow Serving to Kubernetes
- Using Kubeflow and KFServing for simplifying the deployment process

In the previous chapter, we talked about model deployment with AWS Lambda and TensorFlow Lite.

In this chapter, we discuss the "serverful" approach to model deployment: we serve the clothing classification model with TensorFlow Serving on Kubernetes. Also, we talk about Kubeflow, an extension for Kubernetes that makes model deployment easier.

We're going to cover a lot of material in this chapter, but Kubernetes is so complex that it's simply not possible to go deep into detail. Because of that, we often refer to external resources for a more in-depth coverage of some topics. But don't worry; you will learn enough to feel comfortable deploying your own models with it.

9.1 Kubernetes and Kubeflow

Kubernetes is a container orchestration platform. It sounds complex, but it's nothing other than a place where we can deploy Docker containers. It takes care of exposing these containers as web services and scales these services up and down as the amount of requests we receive changes.

Kubernetes is not the easiest tool to learn, but it's very powerful. It's likely that you will need to use it at some point. That's why we decided to cover it in this book.

Kubeflow is another popular tool built on top of Kubernetes. It makes it easier to use Kubernetes to deploy machine learning models. In this chapter, we cover both Kubernetes and Kubeflow.

In the first part, we talk about TensorFlow Serving and plain Kubernetes. We discuss how we can use these technologies for model deployment. The plan for the first part is

- First, we convert the Keras model into the special format used by TensorFlow Serving.
- Then we use TensorFlow Serving to run the model locally.
- After that, we create a service for preprocessing images and communicating with TensorFlow Serving.
- Finally, we deploy both the model and the preprocessing service with Kubernetes.

NOTE This chapter doesn't attempt to cover Kubernetes in depth. We show only how to use Kubernetes for deploying models and often refer to more specialized sources that cover it in more detail.

In the second part, we use Kubeflow, a tool on top of Kubernetes that makes deployment easier:

- We use the same model we've prepared for TensorFlow Serving and deploy it with KFServing — the part of Kubeflow that takes care of serving.
- Then we create a transformer for preprocessing the images and postprocessing the predictions.

The code for this chapter is available in the book's GitHub repository (https://github.com/alexeygrigorev/mlbookcamp-code/) in the chapter-09-kubernetes and chapter-09-kubeflow folders.

Let's get started!

9.2 Serving models with TensorFlow Serving

In chapter 7, we used Keras for predicting the classes of images. In chapter 8, we converted the model to TF Lite and used it for making predictions from AWS Lambda. In this chapter, we do this with TensorFlow Serving.

TensorFlow Serving, usually abbreviated as "TF Serving," is a system designed for serving TensorFlow models. Unlike TF Lite, which is made for mobile devices, TF Serving focuses on servers. Often, the servers have GPUs, and TF Serving knows how to make use of them.

AWS Lambda is great for experimenting and for dealing with small amounts of images — fewer than one million per day. But when we grow past that amount and get more images, AWS Lambda becomes expensive. Then deploying models with Kubernetes and TF Serving is a better option.

Just using TF Serving for deploying models is not sufficient, however. We also need another service for preparing the images. Next, we'll discuss the architecture of the system that we will build.

9.2.1 Overview of the serving architecture

TF Serving focuses on only one thing — serving the model. It expects that the data it receives is already prepared: images are resized, preprocessed, and sent in the right format.

This is why simply putting the model into TF Serving is not enough. We need an additional service that takes care of preprocessing the data.

We need two components for a system for serving a deep learning model (figure 9.1):

- Gateway: The preprocessing part. It gets the URL for which we need to make the prediction, prepares it, and sends it further to the model. We will use Flask for creating this service.
- Model: The part with the actual model. We will use TF Serving for this.

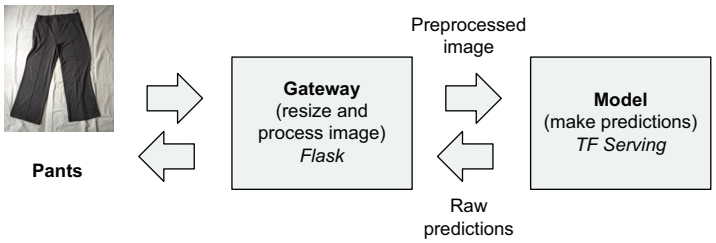


Figure 9.1 The two-tier architecture of our system. The gateway gets the user requests and prepares the data, and TF Serving uses the data to make the predictions.

Creating a system with two components instead of one may seem like an unnecessary complication. In the previous chapter, we didn't need to do it. We had only one part — the lambda function.

In principle, we could take the code from that lambda function, put it in Flask, and use it for serving the model. This approach will indeed work, but it won't be the most effective one. If we need to process millions of images, being able to properly utilize the resources is important.

Having two separate components instead of a single one makes it easier to select the right resource for each part:

- The gateway spends a lot of time downloading the images in addition to doing preprocessing. It doesn't need a powerful computer for that.
- The TF Serving component requires a more powerful machine, often with a GPU. It would be wasteful to use this powerful machine for downloading images.
- We might require many gateway instances and only a few TF Serving instances. By separating them into different components, we can scale each independently.

We will start with the second component — TF Serving.

In chapter 7, we trained a Keras model. To use it with TF Serving, we need to convert it to the special format used by TF Serving, which is called *saved_model*. We do this next.

9.2.2 The saved_model format

The Keras model we trained previously was saved in the h5 format. TF Serving cannot read h5: it expects models to be in the saved_model format. In this section, we convert the h5 model to a saved_model file.

In case you don't have the model from chapter 7, you can download it with `wget`:

```
wget https://github.com/alexeygrigorev/mlbookcamp-
code/releases/download/chapter7-model/xception_v4_large_08_0.894.h5
```

Now let's convert it. We can do this from a Jupyter Notebook or from a Python script.

In either case, we start with imports:

```
import tensorflow as tf
from tensorflow import keras
```

Then load the model:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

And, finally, save it in the saved_model format:

```
tf.saved_model.save(model, 'clothing-model')
```

That's it. After running this code, we have the model saved in the clothing-model folder.

To be able to use this model later, we need to know a few things:

- The name of the model signature. The model signature describes the model's inputs and outputs. You can read more about model signatures here: https://www.tensorflow.org/tfx/serving/signature_defs.
- The name of the input layer.
- The name of the output layer.

When using Keras, we didn't need to worry about it, but TF Serving requires us to have this information.

TensorFlow comes with a special utility for analyzing the models in the saved_model format — `saved_model_cli`. We don't need to install anything extra. We will use the `show` command from this utility:

```
saved_model_cli show --dir clothing-model --all
```

Let's take a look at the output:

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

...

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_8'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 299, 299, 3)
      name: serving_default_input_8:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense_7'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 10)
      name: StatefulPartitionedCall:0
  Method name is: tensorflow/serving/predict
```

The signature definition — serving_default

The input name — input_8

The output name — dense_7

In this output, we're interested in three things:

- The signature definition (signature_def) of the model. In this case, it's `serving_default`.
- The input (`input_8`). The name of the input of the model.
- The output (`dense_7`). tThe name of the output layer of the model.

NOTE Take note of these names — we'll need them later when invoking this model.

The model is converted, and now we're ready to serve it with TF Serving.

9.2.3 Running TensorFlow Serving locally

One of the easiest ways of running TF Serving locally is to use Docker. You can read more about it in the official documentation: https://www.tensorflow.org/tfx/serving/docker. Refer to chapter 5 for more information about Docker.

All we need to do is invoke the `docker run` command specifying the path to the model and its name:

```
docker run -it --rm \
    -p 8500:8500 \
    -v "$(pwd)/clothing-model:/models/clothing-model/1" \
    -e MODEL_NAME=clothing-model \
    tensorflow/serving:2.3.0
```

1 Opens port 8500 for serving

2 Mounts our model

3 Specifies the name of the model

4 Uses the TensorFlow Serving image, version 2.3.0

When running it, we use three parameters:

- `-p`: To map port 8500 on the host machine (the computer where we run Docker) to port 8500 inside the container ❶.
- `-v`: To put the model files inside the Docker image ❷. The model is put in /models/clothing-model/1, where clothing-model is the name of the model and 1 is the version.
- `-e`: To set the `MODEL_NAME` variable to `clothing-model` ❸, which is the directory name from ❷.

To learn more about the `docker run` command, refer to the official Docker documentation (https://docs.docker.com/engine/reference/run/).

After running this command, we should see logs in the terminal:

```
2020-12-26 22:56:37.315629: I tensorflow_serving/core/loader_harness.cc:87]
Successfully loaded servable version {name: clothing-model version: 1}
2020-12-26 22:56:37.321376: I tensorflow_serving/model_servers/server.cc:371]
Running gRPC ModelServer at 0.0.0.0:8500 ...
[evhttp_server.cc : 238] NET_LOG: Entering the event loop ...
```

The `Entering the event loop` message tells us that TF Serving has started successfully and is ready to receive requests.

But we cannot use it yet. To prepare a request, we need to load an image, preprocess it, and convert it to a special binary format. Next, we'll see how we can do it.

9.2.4 Invoking the TF Serving model from Jupyter

For communication, TF Serving uses gRPC — a special protocol designed for high-performance communication. This protocol relies on protobuf, a format for effective data transfer. Unlike JSON, it's binary, which makes the requests significantly more compact.

To understand how to use it, let's first experiment with these technologies from a Jupyter Notebook. We connect to our model deployed with TF Serving using gRPC and protobuf. After that, we can put this code into a Flask application in the next section.

Let's start. We need to install a couple of libraries:

- grpcio: For gRPC support in Python
- tensorflow-serving-api: For using TF Serving from Python

Install them with `pip`:

```
pip install grpcio==1.32.0 tensorflow-serving-api==2.3.0
```

We also need the keras_image_helper library for preprocessing the images. We already used this library in chapter 8. If you haven't installed it yet, use `pip` for that:

```
pip install keras_image_helper==0.0.1
```

Next, create a Jupyter Notebook. We can call it chapter-09-image-preparation. As usual, we start with imports:

```
import grpc
import tensorflow as tf

from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
```

We imported three things:

- gRPC: for communicating with TF Serving
- TensorFlow: for protobuf definitions (We'll see later how it's used.)
- A couple of functions from TensorFlow Serving

Now we need to define the connection to our service:

```
host = 'localhost:8500'
channel = grpc.insecure_channel(host)
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

> **NOTE** We use an insecure channel — a channel that requires no authentication. All the communication between the services in this chapter happens inside the same network. This network is closed to the outside world, so using an insecure channel does not cause any security vulnerabilities. Setting a secure channel is possible but outside the scope of this book.

For preprocessing the images, we use the keras_image_helper library, like previously:

```
from keras_image_helper import create_preprocessor

preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Let's use the same image of pants we used in chapter 8 (figure 9.2).

Figure 9.2 The picture of pants that we use for testing

Let's convert it to a NumPy array:

```
url = "http://bit.ly/mlbookcamp-pants"
X = preprocessor.from_url(url)
```

We have a NumPy array in `X`, but we can't use it as is. For gRPC, we need to convert it to protobuf. TensorFlow has a special function for that: `tf.make_tensor_proto`.

This is how we use it:

```
def np_to_protobuf(data):
    return tf.make_tensor_proto(data, shape=data.shape)
```

This function takes two arguments:

- A NumPy array: `data`,
- The dimensions of this array: `data.shape`

NOTE In this example, we use TensorFlow for converting a NumPy array to protobuf. TensorFlow is a large library, so depending on it for just one small function is unreasonable. In this chapter, we do it for simplicity, but you shouldn't do it in production, because using Docker with big images can create problems: it takes more time to download the image and they occupy more space. Check this repository to see what you can do instead: https://github.com/alexeygrigorev/tensorflow-protobuf.

Now we can use the `np_to_protobuf` function to prepare a gRPC request:

```
pb_request = predict_pb2.PredictRequest()

pb_request.model_spec.name = 'clothing-model'
pb_request.model_spec.signature_name = 'serving_default'
pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X))
```

1. If there is one item, bringing in the one item
2. Sets the model name to clothing-model
3. Specifies the signature name: serving_default
4. Converts X to protobuf, and assigns it to input_8

Let's take a look at each line. First, in 1, we create a request object. TF Serving uses the information from this object to determine how to process the request.

In 2, we specify the name of the model. Recall that when running TF Serving in Docker, we specified the `MODEL_NAME` parameter — we set it to `clothing-model`. Here we say that we want to send the request to that model.

In 3, we specify which signature we want to query. When we analyzed the saved_model file, the signature name was `serving_default`, so this is what we use here. You can read more about signatures in the official TF Serving documentation (https://www.tensorflow.org/tfx/serving/signature_defs).

In ❹, we do two things. First, we convert `X` to protobuf. Then, we set the results to the input named `input_8`. This name also comes from our analysis of the saved_model file.

Let's execute it:

```
pb_result = stub.Predict(pb_request, timeout=20.0)
```

This sends a request to the TF Serving instance. Then TF Serving applies the model to the request and sends back the results. The results are saved to the `pb_result` variable. To get the predictions from there, we need to access one of the outputs:

```
pred = pb_result.outputs['dense_7'].float_val
```

Note that we need to refer to a specific output by name — `dense_7`. When analyzing the signatures of the saved_model file, we also took a note of it — and now used it to get the predictions.

The `pred` variable is a list of floats — the predictions:

```
[-1.868, -4.761, -2.316, -1.062, 9.887, -2.812, -3.666, 3.200, -2.602, -4.835]
```

We need to turn this list of numbers into something that we can understand — we need to connect it to the labels. We use the same approach as in the previous chapters:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

result = {c: p for c, p in zip(labels, pred)}
```

This gives us the final result:

```
{'dress': -1.868,
 'hat': -4.761,
 'longsleeve': -2.316,
 'outwear': -1.062,
 'pants': 9.887,
 'shirt': -2.812,
 'shoes': -3.666,
 'shorts': 3.200,
 'skirt': -2.602,
 't-shirt': -4.835}
```

We see that the `pants` label has the highest score.

We successfully managed to connect to the TF Serving instance from a Jupyter Notebook, and we used gRPC and protobuf for that. Now let's put this code into a web service.

9.2.5 Creating the Gateway service

We already have all the code we need for communicating with our model deployed with TF Serving.

This code, however, is not convenient to use. The users of our model shouldn't need to worry about downloading the image, doing the preprocessing, converting it to protobuf, and all other things we did. They should be able to send an URL of an image and get back the predictions.

To make it easier for our users, we'll put all this code into a web service. The users will interact with the service, and the service will talk to TF Serving. So, the service will act as a gateway to our model. This is why we can simply call it "Gateway" (figure 9.3).

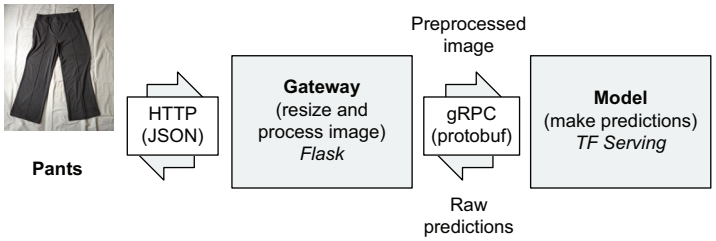


Figure 9.3 The Gateway service is a Flask app that gets an URL to the image and prepares it. Then it uses gRPC and protobuf to communicate with TF Serving.

We use Flask to create this service. We already used Flask previously; you can refer to chapter 5 for more details.

The Gateway service needs to do these things:

- Take the URL of an image in the request.
- Download the image, preprocess it, and convert it to a NumPy array.
- Convert the NumPy array to protobuf, and use gRPC to communicate with TF Serving.
- Postprocess the results — convert the raw list with numbers to human-understandable form.

So, let's create it! Start by creating a file model_server.py — we'll put all this logic there.

First, we get the same imports as we have in the notebook:

```
import grpc
import tensorflow as tf
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc

from keras_image_helper import create_preprocessor
```

Now we need to add Flask imports:

```
from flask import Flask, request, jsonify
```

Next, create the connection gRPC stub:

```
host = os.getenv('TF_SERVING_HOST', 'localhost:8500')
channel = grpc.insecure_channel(host)
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

Makes the TF Serving URL configurable

Instead of simply hardcoding the URL of the TF Serving instance, we make it configurable via the environment variable `TF_SERVING_HOST`. If the variable is not set, we use the default value `'localhost:8500'`.

Now let's create the preprocessor:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Also, we need to define the names of our classes:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]
```

Instead of simply copying and pasting the code from the notebook, we can make the code more organized and put it into two functions:

- `make_request`: For creating a gRPC request from a NumPy array
- `process_response`: For attaching the class labels to the predictions

Let's start with `make_request`:

```
def np_to_protobuf(data):
    return tf.make_tensor_proto(data, shape=data.shape)

def make_request(X):
    pb_request = predict_pb2.PredictRequest()
```

```
    pb_request.model_spec.name = 'clothing-model'
    pb_request.model_spec.signature_name = 'serving_default'
    pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X))
    return pb_request
```

Next, create process_response:

```
def process_response(pb_result):
    pred = pb_result.outputs['dense_7'].float_val
    result = {c: p for c, p in zip(labels, pred)}
    return result
```

And finally, let's put everything together:

```
def apply_model(url):
    X = preprocessor.from_url(url)
    pb_request = make_request(X)
    pb_result = stub.Predict(pb_request, timeout=20.0)
    return process_response(pb_result)
```

Preprocesses an image from the provided URL

Converts the NumPy array into a gRPC request

Executes the request

Processes the response, and attaches the labels to predictions

All the code is ready. We only need to do one last thing: create a Flask app and the `predict` function. Let's do it:

```
app = Flask('clothing-model')

@app.route('/predict', methods=['POST'])
def predict():
    url = request.get_json()
    result = apply_model(url['url'])
    return jsonify(result)

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Now we're ready to run the service. Execute this command in the terminal:

```
python model_server.py
```

Wait until it's ready. We should see the following in the terminal:

```
 * Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
```

Let's test it! Like in chapter 5, we use the requests library for that. You can open any Jupyter Notebook. For example, you can continue in the same notebook where we experimented with connecting to TF Serving with gRPC.

We need to send a request with a URL and show the response. This is how we do it with requests:

```
import requests

req = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = 'http://localhost:9696/predict'

response = requests.post(url, json=req)
response.json()
```

Here, we send a POST request to our service and display the results. The response is the same as previously:

```
{'dress': -1.868,
 'hat': -4.761,
 'longsleeve': -2.316,
 'outwear': -1.062,
 'pants': 9.887,
 'shirt': -2.812,
 'shoes': -3.666,
 'shorts': 3.200,
 'skirt': -2.602,
 't-shirt': -4.835}
```

The service is ready and it works locally. Let's deploy it with Kubernetes!

9.3 Model deployment with Kubernetes

Kubernetes is an orchestration system for automating container deployment. We can use it to host any Docker containers. In this section, we'll see how we can use Kubernetes to deploy our application.

First, we'll start by going over some Kubernetes basics.

9.3.1 Introduction to Kubernetes

The main unit of abstraction in Kubernetes is a *pod*. A pod contains a single Docker image, and when we want to serve something, pods do the actual job.

Pods live on a *node* — this is an actual machine. A node usually contains one or more pods.

To deploy an application, we define a *deployment*. We specify how many pods the application should have and which image should be used. When our application starts to get more requests, sometimes we want to add more pods to our deployment to handle the increase in traffic. This can also happen automatically — this process is called *horizontal autoscaling*.

A *service* is the entry point to the pods in a deployment. The clients interact with the service, not individual pods. When a service gets a request, it routes it to one of the pods in the deployment.

Clients outside of the Kubernetes cluster interact with the services inside the cluster through *ingress*.

Suppose we have a service — Gateway. For this service, we have a deployment (Gateway Deployment) with three pods — pod A, pod B on Node 1, and pod D on Node 2 (figure 9.4). When a client wants to send a request to the service, it's first processed by the ingress, and then the service routes the request to one of the pods. In this example, it's pod A deployed on node 1. The service on pod A processes the request, and the client receives the response.

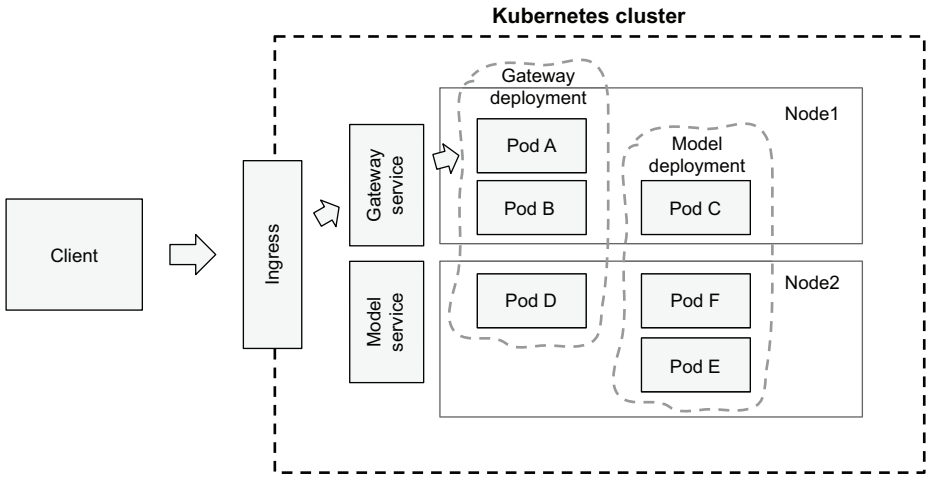


Figure 9.4 The anatomy of a Kubernetes cluster. Pods are instances of our application. They live on nodes — the actual machines. Pods that belong to the same application are grouped in a deployment. The client communicates to services, and the services route the request to one of the pods in a deployment.

This is a very short introduction to the key vocabulary of Kubernetes, but it should be sufficient to get started. To learn more about Kubernetes, refer to the official documentation (https://kubernetes.io/).

In the next section, we'll see how we can create our own Kubernetes cluster on AWS.

9.3.2 Creating a Kubernetes cluster on AWS

To be able to deploy our services to a Kubernetes cluster, we need to have one. We have multiple options:

- You can create a cluster in the cloud. All the major cloud providers make it possible to set up a Kubernetes cluster in the cloud.
- You can set it up locally using Minikube or MicroK8S. You can read more about it here: https://mlbookcamp.com/article/local-k8s.html.

In this section, we use EKS from AWS. EKS, which stands for Elastic Kubernetes Service, is a service from AWS that lets us create a Kubernetes cluster with a minimal

amount of effort. Alternatives are GKE (Google Kubernetes Engine) from Google Cloud and AKS (Azure Kubernetes Service) from Azure.

For this section, you need to use three command-line tools:

- AWS CLI: Manages AWS resources. Refer to appendix A for more information.
- `eksctl`: Manages EKS clusters (https://docs.aws.amazon.com/eks/latest/userguide eksctl.html).
- `kubectl`: Manages resources in a Kubernetes cluster (https://kubernetes.io/docs/tasks/tools/install-kubectl/). It works for any cluster, not just EKS.

The official documentation is sufficient for installing these tools, but you can also refer to the book's website for more information (https://mlbookcamp.com/article/eks).

If you don't use AWS but do use a different cloud provider, you need to use their tools for setting up a Kubernetes cluster. Because Kubernetes is not tied to any particular vendor, most of the instructions in this chapter will work, regardless of where you have the cluster.

Once you have installed `eksctl` and AWS CLI, we can create an EKS cluster.

First, prepare a file with the cluster configuration. Create a file in your project directory and call it cluster.yaml:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: ml-bookcamp-eks
  region: eu-west-1
  version: "1.18"

nodeGroups:
  - name: ng
    desiredCapacity: 2
    instanceType: m5.xlarge
```

After creating the config file, we can use `eksctl` for spinning up a cluster:

```
eksctl create cluster -f cluster.yaml
```

NOTE Creating a cluster takes 15–20 minutes, so be patient.

With this configuration, we create a cluster with Kuberbetes version 1.18 deployed in the eu-west-1 region. The name of the cluster is ml-bookcamp-eks. If you want to deploy it to a different region, you can change it. This cluster will use two m5.xlarge machines. You can read more about this type of instance here: https://aws.amazon.com/ec2/instance-types/m5/. This is sufficient for the experiments we need to do in this chapter for both Kubernetes and Kubeflow.

NOTE EKS is not covered by the AWS free tier. You can learn more about the costs in the official documentation of AWS (https://aws.amazon.com/eks/pr-icing/).

Once it's created, we need to configure `kubectl` to be able to access it. For AWS, we do this with the AWS CLI:

```
aws eks --region eu-west-1 update-kubeconfig --name ml-bookcamp-eks
```

This command should generate a `kubectl` config file in the default location. On Linux and MacOS, this location is ~/.kube/config.

Now let's check that everything works, and that we can connect to our cluster using `kubectl`:

```
kubectl get service
```

This command returns the list of currently running services. We haven't deployed anything, so we expect to see only one service — Kubernetes itself. This is the result you should see:

```
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.100.0.1   <none>        443/TCP   6m17s
```

The connection works, and now we can deploy a service. To do that, we first need to prepare a Docker image with the actual service. Let's do that next.

9.3.3 Preparing the Docker images

In the previous sections, we created two components of the serving system:

- TF-Serving: The component with the actual model
- Gateway: The component for image preprocessing that communicates with TF Serving

Now we deploy them. We start first with deploying the TF Serving image.

The TensorFlow Serving image

As in chapter 8, we first need to publish our image to ECR — the Docker registry of AWS. Let's create a registry called model-serving:

```
aws ecr create-repository --repository-name model-serving
```

It should return a path like this:

```
<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-serving
```

IMPORTANT Take note — we'll need this path later.

When running a Docker image of TF Serving locally, we used this command (you don't need to run it now):

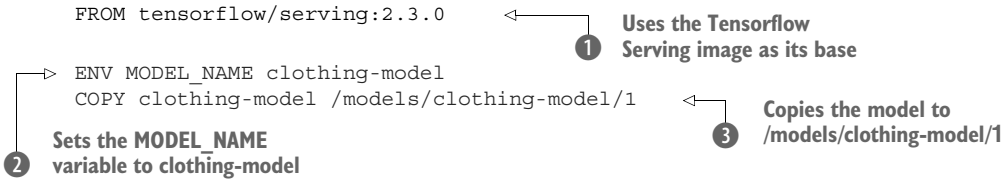
```
docker run -it --rm \
    -p 8500:8500 \
    -v "$(pwd)/clothing-model:/models/clothing-model/1" \
```

```
    -e MODEL_NAME=clothing-model \
    tensorflow/serving:2.3.0
```

We used the `-v` parameter to mount the model from the `clothing-model` to the /models/clothing-model/1 directory within the image.

It's also possible to do it with Kubernetes, but in this chapter, we follow a simpler approach and include the model into the image itself, similar to what we did in chapter 8.

Let's create a Dockerfile for that. We can name it tf-serving.dockerfile:



We base our image on the TensorFlow Serving image in ❶. Next, in ❷, we set the environment variable `MODEL_NAME` to `clothing-model`, which is the equivalent of the `-e` parameter. Next, we copy the model to /models/clothing-model/1 in ❸, which is the equivalent of using the `-v` parameter.

> **NOTE** If you want to use a computer with a GPU, use the tensorflow/serving :2.3.0-gpu image (commented as ❶ in the Dockerfile).

Let's build it:

```
IMAGE_SERVING_LOCAL="tf-serving-clothing-model"
docker build -t ${IMAGE_SERVING_LOCAL} -f tf-serving.dockerfile .
```

Next, we need to publish this image to ECR. First, we need to authenticate with ECR using AWS CLI:

```
$(aws ecr get-login --no-include-email)
```

> **NOTE** You need to include the "$" when typing the command. The command inside the parentheses returns another command. Using the "`$()`", we execute this command.

Next, tag the image with the remote URI:

```
ACCOUNT=XXXXXXXXXXXX
REGION=eu-west-1
REGISTRY=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/model-serving
IMAGE_SERVING_REMOTE=${REGISTRY}:${IMAGE_SERVING_LOCAL}
docker tag ${IMAGE_SERVING_LOCAL} ${IMAGE_SERVING_REMOTE}
```

Be sure to change the `ACCOUNT` and `REGION` variables.

Now we're ready to push the image to ECR:

```
docker push ${IMAGE_SERVING_REMOTE}
```

It's pushed! Now we need to do the same with the Gateway component.

THE GATEWAY IMAGE

Now let's prepare the image for the Gateway component. Gateway is a web service, and it relies on a number of Python libraries:

- Flask and Gunicorn
- keras_image_helper
- grpcio
- TensorFlow
- TensorFlow-Serving-API

Remember that in chapter 5, we used Pipenv for managing dependencies. Let's use it here as well:

```
pipenv install flask gunicorn \
    keras_image_helper==0.0.1 \
    grpcio==1.32.0 \
    tensorflow==2.3.0 \
    tensorflow-serving-api==2.3.0
```

Running this command creates two files: Pipfile and Pipfile.lock.

> **WARNING** Even though we already mentioned it, it's important enough to repeat it. Here, we rely on TensorFlow for only one function. In a production environment, it's better not to install TensorFlow. In this chapter, we do it for simplicity. Instead of depending on TensorFlow, we can take only the protobuf files we need and reduce the size of our Docker image significantly. Refer to this repository for instructions: https://github.com/alexeygrigorev/tensorflow-protobuf.

Now let's create a Docker image. Start with creating a Dockerfile named gateway.dockerfile with the following content:

```
FROM python:3.7.5-slim

ENV PYTHONUNBUFFERED=TRUE

RUN pip --no-cache-dir install pipenv

WORKDIR /app

COPY ["Pipfile", "Pipfile.lock", "./"]
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache
```

```
COPY "model_server.py" "model_server.py"

EXPOSE 9696

ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "model_server:app"]
```

This Dockerfile is very similar to the file we had previously. Refer to chapter 5 for more information about it.

Let's build this image now:

```
IMAGE_GATEWAY_LOCAL="serving-gateway"
docker build -t ${IMAGE_GATEWAY_LOCAL} -f gateway.dockerfile .
```

And push it to ECR:

```
IMAGE_GATEWAY_REMOTE=${REGISTRY}:${IMAGE_GATEWAY_LOCAL}
docker tag ${IMAGE_GATEWAY_LOCAL} ${IMAGE_GATEWAY_REMOTE}

docker push ${IMAGE_GATEWAY_REMOTE}
```

> **NOTE** For verifying that these images work well together locally, you need to use Docker Compose (https://docs.docker.com/compose/). This is a very useful tool, and we recommend that you spend time learning it, but we will not cover it here.

We have published both images to ECR, and now we're ready to deploy the services to Kubernetes! We'll do that next.

9.3.4 Deploying to Kubernetes

Before we deploy, let's revisit the basics of Kubernetes. We have the following objects living inside a cluster:

- Pod: The smallest unit in Kubernetes. It's a single process, and we have one Docker container in one pod.
- Deployment: A group of multiple related pods.
- Service: What sits in front of a deployment and routes the requests to individual pods.

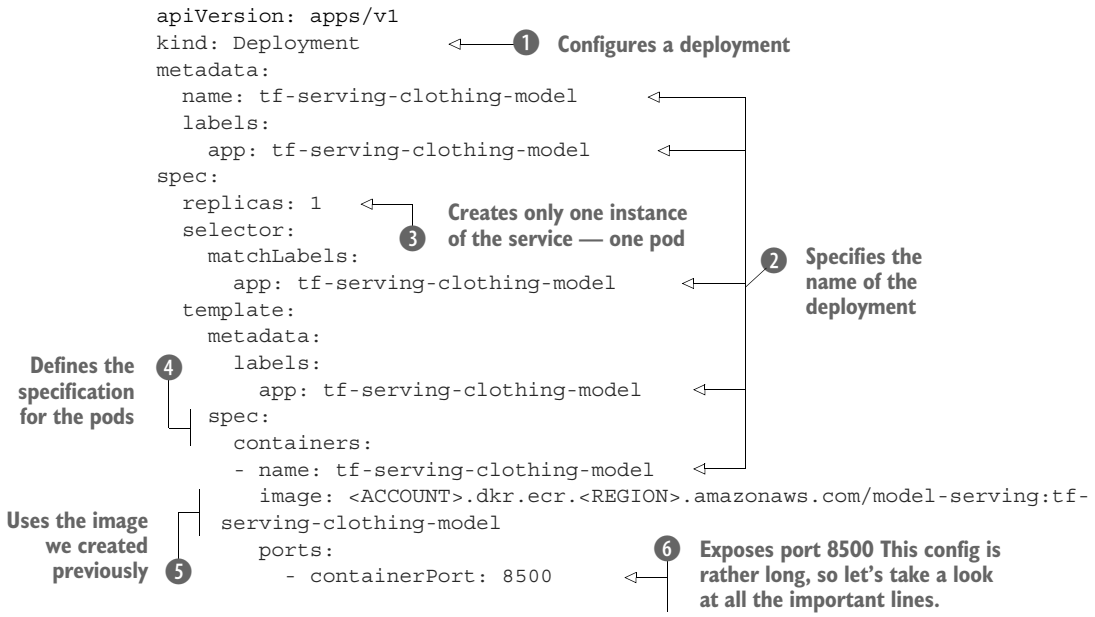
To deploy an application to Kubernetes, we need to configure two things:

- A deployment: It specifies how the pods of this deployment will look.
- A service: Specifies how to access the service and how the service connects to the pods.

Let's start with configuring the deployment for TF Serving.

DEPLOYMENT FOR TF SERVING

In Kubernetes, we usually configure everything with YAML files. For configuring a deployment, we create a file named tf-serving-clothing-model-deployment.yaml in our project directory with the following content:



In ❶, we specify the type of the Kubernetes object we want to configure in this YAML file — it's a deployment.

In ❷, we define the name of the deployment as well as set some metadata information. We need to repeat it multiple times: one time for setting the name of the deployment ("name") and a few more times ("labels: app") for the service that we'll configure later.

In ❸, we set the number of instances — pods — we want to have in the deployment.

In ❹, we specify the configuration for the pods — we set the parameters that all of the pods will have.

In ❺, we set the URI for the Docker image. The pod will use this image. Don't forget to put your account ID there as well as the correct region.

Finally, in ❻, we open port 8500 on the pods of this deployment. This is the port that TF Serving uses.

To learn more about configuring deployments in Kubernetes, check the official documentation (https://kubernetes.io/docs/concepts/workloads/controllers/deployment).

We have a config. Now we need to use it to create a Kubernetes object — a deployment in our case. We do it by using the `apply` command from `kubectl`:

```
kubectl apply -f tf-serving-clothing-model-deployment.yaml
```

The `-f` parameter tells `kubectl` that it needs to read the configuration from the config file.

To verify that it's working, we need to check if a new deployment appeared. This is how we can get the list of all active deployments:

```
kubectl get deployments
```

The output should look similar to this:

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
tf-serving-clothing-model    1/1     1            1           41s
```

We see that our deployment is there. Also, we can get the list of pods. It's quite similar to getting the list of all deployments:

```
kubectl get pods
```

We should see something like that in the output:

```
NAME                                          READY   STATUS    RESTARTS   AGE
tf-serving-clothing-model-56bc84678d-b6n4r    1/1     Running   0          108s
```

Now we need to create a service on top of this deployment.

SERVICE FOR TF SERVING

We want to invoke TF Serving from Gateway. For that, we need to create a service in front of the TF Serving deployment.

Like with a deployment, we start by creating a configuration file for the service. It's also a YAML file. Create a file called tf-serving-clothing-model-service.yaml with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: tf-serving-clothing-model
  labels:
    app: tf-serving-clothing-model
spec:
  ports:
    - port: 8500
      targetPort: 8500
      protocol: TCP
      name: http
  selector:
    app: tf-serving-clothing-model
```

Configures the name of the service

Specification of the service — the port that will be used

Connects the service to the deployment by specifying the label of the deployment

We apply it in the same way — by using the `apply` command:

```
kubectl apply -f tf-serving-clothing-model-service.yaml
```

To check that it works, we can get the list of all services and see if our service is there:

```
kubectl get services
```

We should see something like

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE
kubernetes          ClusterIP   10.100.0.1       <none>        443/TCP     84m
tf-serving-         ClusterIP   10.100.111.165   <none>        8500/TCP    19s
clothing-model
```

In addition to the default Kubernetes service, we also have tf-serving-clothing-model, which is the service we just created.

To access this service, we need to get its URL. The internal URLs typically follow this pattern:

```
<service-name>.<namespace-name>.svc.cluster.local
```

The `<service-name>` part is tf-serving-clothing-model.

We haven't used any specific namespace for this service, so Kubernetes automatically put the service in the "default" namespace. We won't cover namespaces here, but you can read more about them in the official documentation (https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/).

This is the URL for the service we just created:

```
tf-serving-clothing-model.default.svc.cluster.local
```

We'll need this URL later, when configuring Gateway.

We've created a deployment for TF Serving as well as a service. Now let's create a deployment for Gateway.

Deployment for Gateway

Like previously, we start by creating a YAML file with configuration. Create a file named serving-gateway-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: serving-gateway
  labels:
    app: serving-gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: serving-gateway
  template:
    metadata:
      labels:
        app: serving-gateway
    spec:
      containers:
      - name: serving-gateway
        image: <ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-
     serving:serving-gateway
```

```
          ports:
            - containerPort: 9696
Sets the value for the TF_SERVING_HOST environment variable
          env:
          - name: TF_SERVING_HOST
            value: "tf-serving-clothing-model.default.svc.cluster.local:8500"
```

Replace `<ACCOUNT>` and `<REGION>` in the image URL with your values.

The configuration of this deployment is very similar to the deployment of TF Serving, with one important difference: we specify the value of the `TF_SERVING_HOST` variable by setting it to the URL of the service with our model (shown in bold in the listing).

Let's apply this configuration:

```
kubectl apply -f serving-gateway-deployment.yaml
```

This should create a new pod and a new deployment. Let's take a look at the list of pods:

```
kubectl get pod
```

Indeed, a new pod is there:

```
NAME                                          READY   STATUS    RESTARTS   AGE
tf-serving-clothing-model-56bc84678d-b6n4r    1/1     Running   0          1h
serving-gateway-5f84d67b59-lx8tq              1/1     Running   0          30s
```

WARNING Gateway uses gRPC to communicate with TF Serving. When deploying multiple instances of TF Serving, you may run into a problem with distributing the load between these instances (https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/). To solve it, you will need to install a service mesh tool like Linkerd, Istio, or something similar. Talk to your operations team to see how you can do it at your company.

We've created a deployment for Gateway. Now we need to configure the service for. We do it next.

SERVICE FOR GATEWAY

We've created a deployment for Gateway, and now we need to create a service. This service is different from the service we created for TF Serving — it needs to be publicly accessible, so services outside of our Kubernetes cluster can use it. For that, we need to use a special type of service — LoadBalancer. It creates an external load balancer, which is available outside of the Kubernetes cluster. In the case of AWS, it uses ELB, the Elastic Load Balancing service.

Let's create a config file named serving-gateway-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: serving-gateway
```

```
  labels:
    app: serving-gateway
spec:
   type: LoadBalancer
  ports:
    - port: 80
      targetPort: 9696
      protocol: TCP
      name: http
  selector:
    app: serving-gateway
```

❶ Uses the LoadBalancer type

❷ Maps port 9696 in the pod to port 80 n the service

In ❶, we specify the type of the service — LoadBalancer.

In ❷, we connect port 80 in the service to port 9696 in pods. This way, we don't need to specify the port when connecting to the service — it will use the default HTTP port, which is 80.

Let's apply this config:

```
kubectl apply -f serving-gateway-service.yaml
```

To see the external URL of the service, use the `describe` command:

```
kubectl describe service serving-gateway
```

It will output some information about the service:

```
Name:                     serving-gateway
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=serving-gateway
Type:                     LoadBalancer
IP Families:              <none>
IP:                       10.100.100.24
IPs:                      <none>
LoadBalancer Ingress:     ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-
     1.elb.amazonaws.com
Port:                     http  80/TCP
TargetPort:               9696/TCP
NodePort:                 http  32196/TCP
Endpoints:                <none>
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                  Message
  ----    ------                ----  ----                  -------
  Normal  EnsuringLoadBalancer   4s   service-controller    Ensuring load
                                                            balancer
  Normal  EnsuredLoadBalancer    2s   service-controller    Ensured load
                                                            balancer
```

We're interested in the line with `LoadBalancer Ingress`. This is the URL we need to use to access the Gateway service. In our case, this is the URL:

```
ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-1.elb.amazonaws.com
```

The Gateway service is ready to use. Let's do it!

9.3.5 Testing the service

When running TF Serving and Gateway locally, we prepared a simple snippet of Python code for testing our service. Let's reuse it. Go to the same notebook, and replace the local IP address by the URL we got from the previous section:

```
import requests

req = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = 'http://ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-
     1.elb.amazonaws.com/predict'

response = requests.post(url, json=req)
response.json()
```

Run it. As a result, we get the same predictions as previously:

```
{'dress': -1.86829,
 'hat': -4.76124,
 'longsleeve': -2.31698,
 'outwear': -1.06257,
 'pants': 9.88716,
 'shirt': -2.81243,
 'shoes': -3.66628,
 'shorts': 3.20036,
 'skirt': -2.60233,
 't-shirt': -4.83504}
```

It's working — and it means we just successfully deployed our deep learning model with TF Serving and Kubernetes!

> **IMPORTANT** If you finished experimenting with EKS, don't forget to shut down the cluster. If you don't turn it off, you'll need to pay for it, even if it's idle and you don't use it. You will find the instructions for that at the end of this chapter.

In this example, we covered Kubernetes from the user's perspective only, not from an operational standpoint. We haven't talked about autoscaling, monitoring, alerting, and other important topics required for productionizing machine learning models.

For more details about these topics, consult a Kubernetes book or the official documentation of Kubernetes.

You probably noticed that we needed to do quite a lot of things for deploying a single model: create a Docker image, push it to ECR, create a deployment, create a service. Doing this for a couple of models is not a problem, but if you need to do it for tens or hundreds of models, it becomes problematic and repetitive.

There's a solution — Kubeflow. It makes deployment easier. In the next section, we'll see how we can use it for serving Keras models.

9.4 Model deployment with Kubeflow

Kubeflow is a project that aims to simplify the deployment of machine learning services on Kubernetes.

It consists of a set of tools, each of which aims at solving a particular problem. For example:

- Kubeflow Notebooks Server: Makes it easier to centrally host Jupyter Notebooks
- Kubeflow Pipelines: Automates the training process
- Katib: Selects the best parameters for the model
- Kubeflow Serving (abbreviated as "KFServing"): Deploys machine learning models

And many others. You can read more about its components here: https://www.kubeflow.org/docs/components/.

In this chapter, we focus on model deployment, so we'll need to use only one component of Kubeflow — KFServing.

If you want to install the entire Kubeflow project, refer to the official documentation. It has the installation instructions for the major cloud providers such as Google Cloud Platform, Microsoft Azure, and AWS (https://www.kubeflow.org/docs/aws/aws-e2e/).

For the instructions about installing only KFServing without the rest of Kubeflow on AWS, refer to the book's website: https://mlbookcamp.com/article/kfserving-eks-install. We used this article for setting up the environment for the rest of this chapter, but the code here should work with any Kubeflow installation with minor changes.

> **NOTE** The installation may be nontrivial for you, especially if you haven't done anything similar in the past. If you are not sure about some things, ask somebody from the operations team to help you set it up.

9.4.1 Preparing the model: Uploading it to S3

To deploy a Keras model with KFServing, we first need to convert it to the saved_model format. We already did this previously, so we can just use the converted files.

Next, we need to create a bucket in S3, where we will put our models. Let's call it mlbookcamp-models-<NAME>, where <NAME> could be anything — for example, your name. Bucket names must be unique across the entire AWS. That's why we need

to add some suffix to the name of the bucket. It should be in the same region as our EKS cluster. In our case, it's eu-west-1.

We can create it with the AWS CLI:

```
aws s3api create-bucket \
    --bucket mlbookcamp-models-alexey \
    --region eu-west-1 \
    --create-bucket-configuration LocationConstraint=eu-west-1
```

After creating a bucket, we need to upload the model there. Use the AWS CLI for that:

```
aws s3 cp --recursive clothing-model s3://mlbookcamp-models-alexey/clothing-
     model/0001/
```

Note that there's "0001" at the end. This is important — KFServing, like TF Serving, needs a version of the model. We don't have any previous versions of this model, so we add "0001" at the end.

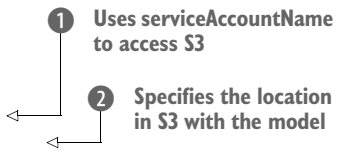
Now we're ready to deploy this model.

9.4.2 Deploying TensorFlow models with KFServing

Previously, when deploying our model with plain Kubernetes, we needed to configure a deployment and then a service. Instead of doing it, KFServing defines a special kind of Kubernetes object — InferenceService. We need to configure it only once, and it will take care of creating all other Kubernetes objects — including a service and a deployment — automatically.

First, create another YAML file (tf-clothes.yaml) with the following content:

```
apiVersion: "serving.kubeflow.org/v1beta1"
kind: "InferenceService"
metadata:
  name: "clothing-model"
spec:
  default:
    predictor:
      serviceAccountName: sa
      tensorflow:
        storageUri: "s3://mlbookcamp-models-alexey/clothing-model"
```



When accessing a model from S3, we need to specify the service account name to be able to get the model. This tells KFServing how to access the S3 bucket — and we specify it in ❶. The article about installing KFServing on EKS covers this as well (https://mlbookcamp.com/article/kfserving-eks-install).

Like with usual Kubernetes, we use `kubectl` to apply this config:

```
kubectl apply -f tf-clothing.yaml
```

Because it creates an InferenceService object, we need to get the list of such objects using the `get` command from `kubectl`:

```
kubectl get inferenceservice
```

We should see something like this:

```
NAME             URL                          READY      AGE
clothing-model   http://clothing-model...  True   ...  97s
```

If our service `READY` is not yet `True`, we need to wait a bit before it becomes ready. It may take 1–2 minutes.

Now take note of the URL and the name of the model:

- The URL: https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/clothing-model. In your configuration, the host will be different, so the entire URL will also be different.
- The model name: clothing-model.

NOTE It may take some time for the URL to become reachable from our laptop. Changes in DNS may need some time to propagate.

9.4.3 Accessing the model

The model is deployed. Let's use it! For that, we can start a Jupyter Notebook or create a Python script file.

KFServing uses HTTP and JSON, so we use the requests library for communicating with it. So let's start by importing it:

```
import requests
```

Next, we need to use the image preprocessor for preparing the images. It's the same one we used previously:

```
from keras_image_helper import create_preprocessor

preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Now, we need an image for testing. We use the same image of pants as in the previous section and use the same code for getting it and preprocessing it:

```
image_url = "http://bit.ly/mlbookcamp-pants"
X = preprocessor.from_url(image_url)
```

The `X` variable contains a NumPy array. We need to convert it to a list before we can send the data to KFServing:

```
data = {
    "instances": X.tolist()
}
```

We have the request. As the next step, we need to define the URL where we will send this request. We already have it from the previous section, but we need to modify it slightly:

- Use HTTPS instead of HTTP.
- Add ":predict" at the end of the URL.

With these changes, this is how the URL appears:

```
url = 'https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/
     clothing-model:predict'
```

We're ready to post the request:

```
resp = requests.post(url, json=data)
results = resp.json()
```

Let's take a look at the results:

```
{'predictions': [[-1.86828923,
   -4.76124525,
   -2.31698346,
   -1.06257045,
   9.88715553,
   -2.81243205,
   -3.66628242,
   3.20036,
   -2.60233665,
   -4.83504581]]}
```

Like we did previously, we need to translate the predictions into human-readable form. We do it by assigning a label to each element of the result:

```
pred = results['predictions'][0]

labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

result = {c: p for c, p in zip(labels, pred)}
```

Here's the result:

```
{'dress': -1.86828923,
 'hat': -4.76124525,
 'longsleeve': -2.31698346,
 'outwear': -1.06257045,
 'pants': 9.88715553,
 'shirt': -2.81243205,
 'shoes': -3.66628242,
 'shorts': 3.20036,
 'skirt': -2.60233665,
 't-shirt': -4.83504581}
```

We have deployed our model, and it can be used.

But we cannot expect that the people who use our model will be happy about having to prepare the images themselves. In the next section, we'll talk about transformers — they can take away the burden of preprocessing the images.

9.4.4 KFServing transformers

In the previous section, we introduced the Gateway service. It was sitting between the client and the model, and it took care of transforming the requests from the clients to the format the model expects (figure 9.5).

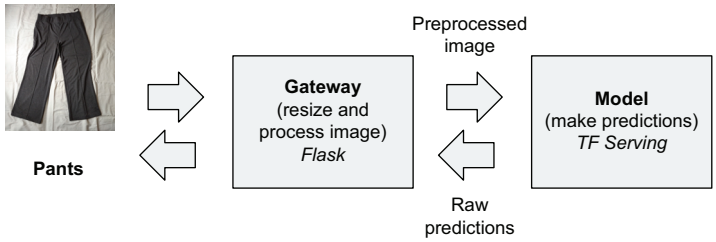


Figure 9.5 The Gateway service takes care of preprocessing the image, so the clients of our applications don't need to do it.

Fortunately for us, we don't have to introduce another Gateway service for KFServing. Instead, we can use a *transformer*.

Transformers take care of

- Preprocessing the request coming from the client and converting it to the format our model expects
- Postprocessing the output of the model — converting it to the format the client needs

We can put all the preprocessing code from the previous section into a transformer (figure 9.6).

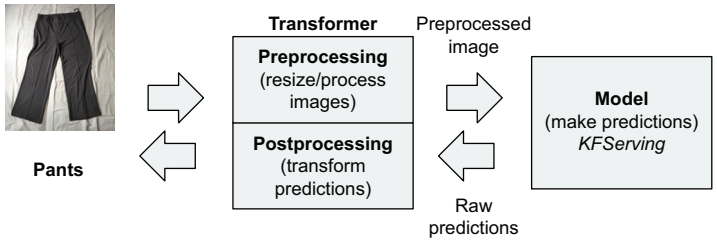


Figure 9.6 The KFServing transformer can download the image and prepare it in the preprocessing step, as well as attach labels to the output of the model in the postprocessing step.

Like the Gateway service we created manually, transformers in KFServing are deployed separately from the model. This means that they can scale up and down independently. It's a good thing — they perform a different kind of work:

- Transformers are doing I/O work (downloading the image).
- Models are doing CPU-intensive work (applying the neural network to make predictions).

To create a transformer, we need to install the KFServing library for Python and create a class that extends the `KFModel` class.

It looks like this:

```
class ImageTransformer(kfserving.KFModel):
    def preprocess(self, inputs):
        # implement pre-processing logic

    def postprocess(self, inputs):
        # implement post-processing logic
```

We will not go into details about building your own transformer, but if you'd like to know how to do it, check out this article: https://mlbookcamp.com/article/kfserving-transformers. Instead, for this book, we've prepared a transformer that uses the keras_image_helper library. You can check its source code here: https://github.com/alexeygrigorev/kfserving-keras-transformer.

Let's use it. First, we need to delete the old inference service:

```
kubectl delete -f tf-clothes.yaml
```

Then, update the config file (tf-clothes.yaml), and include the transformer section (in bold) there:

```
apiVersion: "serving.kubeflow.org/v1alpha2"
kind: "InferenceService"
metadata:
  name: "clothing-model"
```

```
spec:
  default:
    predictor:
      serviceAccountName: sa
      tensorflow:
        storageUri: "s3://mlbookcamp-models-alexey/clothing-model"
    transformer:
      custom:
        container:
          image: "agrigorev/kfserving-keras-transformer:0.0.1"
          name: user-container
          env:
            - name: MODEL_INPUT_SIZE
              value: "299,299"
            - name: KERAS_MODEL_NAME
              value: "xception"
            - name: MODEL_LABELS
              value: "dress,hat,longsleeve,outwear,pants,
     shirt,shoes,shorts,skirt,t-shirt"
```

Defines the model in the predictor section

Defines the transformer in the transformer section

Sets the image for the transformer

Configures it — specifies input size, model name, and labels

In addition to the "predictor" section, which we had previously, we add another one — "transformer." The transformer we use is a publicly available image at agrigorev/kfserving-keras-transformer:0.0.1.

It relies on the keras_image_helper library to do the transformation. For that, we need to set three parameters:

- `MODEL_INPUT_SIZE`: The size of the input that the model expects: 299 x 299
- `KERAS_MODEL_NAME`: The name of the architecture from Keras applications (https://keras.io/api/applications/) that we used for training the model
- `MODEL_LABELS`: The classes that we want to predict

Let's apply this config:

```
kubectl apply -f tf-clothes.yaml
```

Wait a couple of minutes before it becomes ready — use `kubectl get inferenceservice` to check the status.

After it's deployed (`READY` is `True`), we can test it. We'll do that next.

9.4.5 Testing the transformer

With a transformer, we don't need to worry about preparing the image: it's enough to just send the URL of an image. The code becomes much simpler.

This is how it looks:

```
import requests

data = {
    "instances": [
        {"url": "http://bit.ly/mlbookcamp-pants"},
    ]
}
```

```
url = 'https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/
     clothing-model:predict'
result = requests.post(url, json=data).json()
```

The URL of the service stays the same. The result contains the predictions:

```
{'predictions': [{'dress': -1.8682, 'hat': -4.7612, 'longsleeve': -2.3169,
'outwear': -1.0625, 'pants': 9.8871, 'shirt': -2.8124, 'shoes': -3.6662,
'shorts': 3.2003, 'skirt': -2.6023, 't-shirt': -4.8350}]}
```

And that's all! Now we can use the model.

9.4.6 Deleting the EKS cluster

After experimenting with EKS, don't forget to shut down the cluster. Use `eksctl` for that:

```
eksctl delete cluster --name ml-bookcamp-eks
```

To verify that the cluster was removed, you can check the EKS service page in AWS Console.

9.5 Next steps

You've learned the basics you need for training a classification model for predicting the type of clothes. We've covered a lot of material, but there is a lot more to learn than we could fit in this chapter. You can explore this topic more by doing the exercises.

9.5.1 Exercises

- Docker Compose is a tool for running applications with multiple containers. In our example, Gateway needs to communicate with the TF Serving model; that is why we need to be able to link them. Docker Compose can help with that. Experiment with it for running TF Serving and Gateway locally.
- In this chapter, we used EKS from AWS. For learning Kubernetes, it's beneficial to experiment with Kubernetes locally. Use Minikube or Microk8s to reproduce the example with TF Serving and Gateway locally.
- For all experiments in this chapter, we used the default Kubernetes namespace. In practice, we typically use different namespaces for different groups of applications. Learn more about namespaces in Kubernetes and then deploy our services in a different namespace. For example, you can call it "models."
- KFServing transformers are a powerful tool for preprocessing the data. We haven't discussed how we can implement them ourselves and instead used an already-implemented transformer. To learn more about them, implement this transformer yourself.

9.5.2 Other projects

There are many projects that you can do to learn Kubernetes and Kubeflow better:

- In this chapter, we covered a deep learning model. It's quite complex, and we ended up creating two services. Other models we developed before chapter 7 are less complex and only require a simple Flask app for hosting them. You can deploy the models from chapters 2, 3, and 6 using Flask and Kubernetes.
- KFServing can be used for deploying other types of models, not just TensorFlow. Use it for deploying the Scikit-learn models from chapters 3 and 6.

Summary

- TensorFlow-Serving is a system for deploying Keras and TensorFlow models. It uses gRPC and protobuf for communication, and it's highly optimized for serving.
- When using TensorFlow Serving, we typically need a component for preparing the user request into the format the model expects. This component hides the complexity of interacting with TensorFlow Serving and makes it easier for the clients to use the model.
- To deploy something on Kubernetes, we need to create a deployment and a service. The deployment describes what should be deployed: the Docker image and its configuration. The service sits in front of a deployment and routes requests to individual containers.
- Kubeflow and KFServing make the deployment process simpler: we need to specify only the location to the model, and they take care of creating a deployment, a service, and other important things automatically.
- KFServing transformers make it easier to preprocess data coming to the model and postprocess the results. With transformers, we don't need to create a special Gateway service for preprocessing.

appendix A
Preparing the environment

A.1 Installing Python and Anaconda

For the projects in this book, we will use Anaconda, a Python distribution that comes with most of the required machine learning packages that you'll need to use: NumPy, SciPy, Scikit-learn, Pandas, and many more.

A.1.1 Installing Python and Anaconda on Linux

The instructions in this section will work regardless of whether you're installing Anaconda on a remote machine or your laptop. Although we tested it only on Ubuntu 18.04 LTS and 20.04 LTS, this process should work fine for most Linux distributions.

> **NOTE** Using Ubuntu Linux is recommended for the examples in this book. It's not a strict requirement, however, and you should not have problems running the examples in other operating systems. If you don't have a computer with Ubuntu, it's possible to rent one online in the cloud. Please refer to the "Renting a server on AWS" section for more detailed instructions.

Almost every Linux distribution comes with a Python interpreter installed, but it's always a good idea to have a separate installation of Python to avoid trouble with the system Python. Using Anaconda is a great option: it's installed in the user directory and it doesn't interfere with the system Python.

To install Anaconda, you first need to download it. Go to https://www.anaconda.com and click Get Starter. Then select Download Anaconda Installer. This should take you to https://www.anaconda.com/products/individual.

Select 64-Bit (x86) installer and the latest available version — 3.8 at the moment of writing (figure A.1).

Next, copy the link to the installation package. In our case it was https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh.

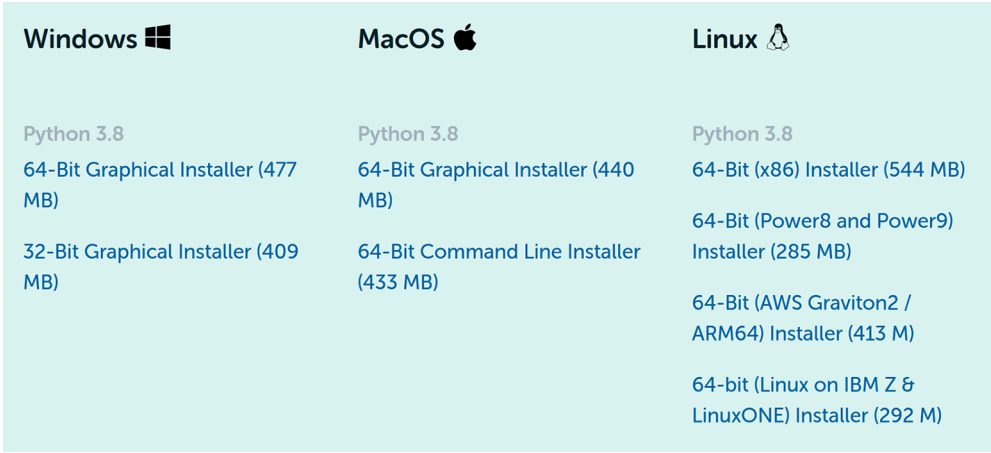


Figure A.1 Downloading the Linux Installer for Anaconda

NOTE If a newer version of Anaconda is available, you should install it instead. All the code will work on newer versions without problems.

Now go to the terminal to download it:

```
wget  https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh
```

Then install it:

```
bash Anaconda3-2021.05-Linux-x86_64.sh
```

Read the agreement, type "yes" if you accept it, and then select the location where you want to install Anaconda. You can use the default location, but you don't have to.

During the installation, you'll be asked if you want to initialize Anaconda. Type "yes," and it will do everything automatically:

```
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>> yes
```

If you don't want to let the installer initialize it, you can do it manually by adding the location with Anaconda's binaries to the `PATH` variable. Open the .bashrc file in the home directory and add this line at the end:

```
export PATH=~/anaconda3/bin:$PATH
```

After the installation has completed, you can delete the installer:

```
rm Anaconda3-2021.05-Linux-x86_64.sh
```

Next, open a new terminal shell. If you're using a remote machine, you can simply exit the current session by pressing Ctrl-D and then log in again using the same `ssh` command as previously.

Now everything should work. You can test that your system picks the right binary by using the `which` command:

```
which python
```

If you're running on an EC2 instance from AWS, you should see something similar to this:

```
/home/ubuntu/anaconda3/bin/python
```

Of course, the path may be different, but it should be the path to the Anaconda installation.

Now you're ready to use Python and Anaconda.

A.1.2 Installing Python and Anaconda on Windows

LINUX SUBSYSTEM FOR WINDOWS

The recommended way to install Anaconda on Windows is to use the Linux Subsystem for Windows.

To install Ubuntu on Windows, open the Microsoft Store and look for ubuntu in the search box; then select Ubuntu 18.04 LTS (figure A.2).

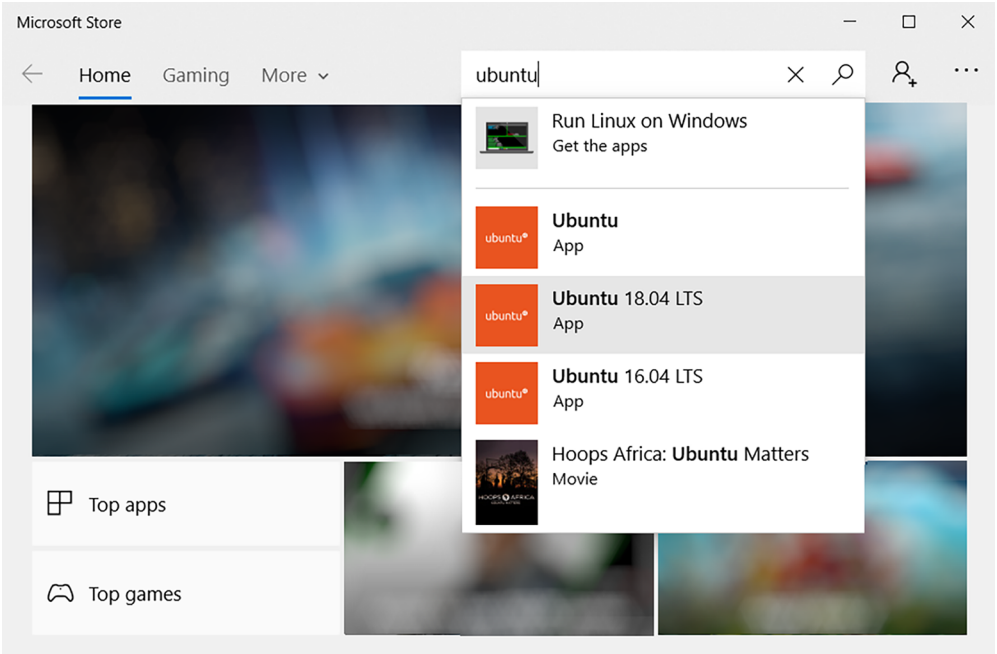


Figure A.2 Use Microsoft Store to install Ubuntu on Windows.

To install it, simply click Get in the next window (figure A.3).

Figure A.3 To install Ubuntu 18.04 for Windows, click Get.

Once it's installed, we can use it by clicking the Launch button (figure A.4).

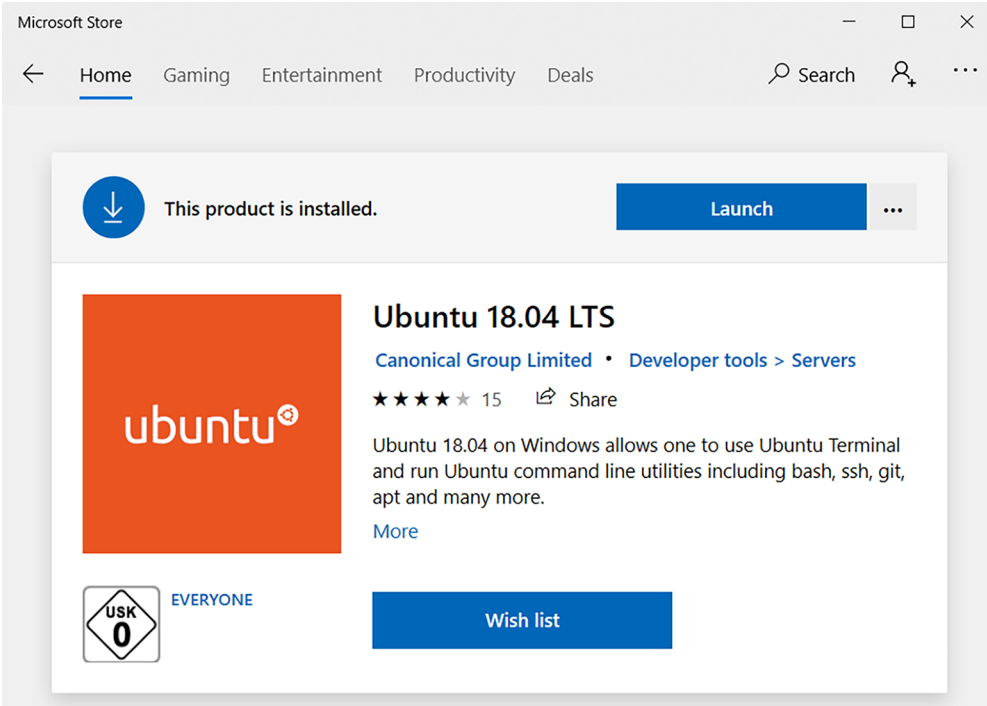


Figure A.4 Click Launch to run the Ubuntu terminal.

When running it for the first time, it will ask you to specify the username and password (figure A.5). After that, the terminal is ready to use.

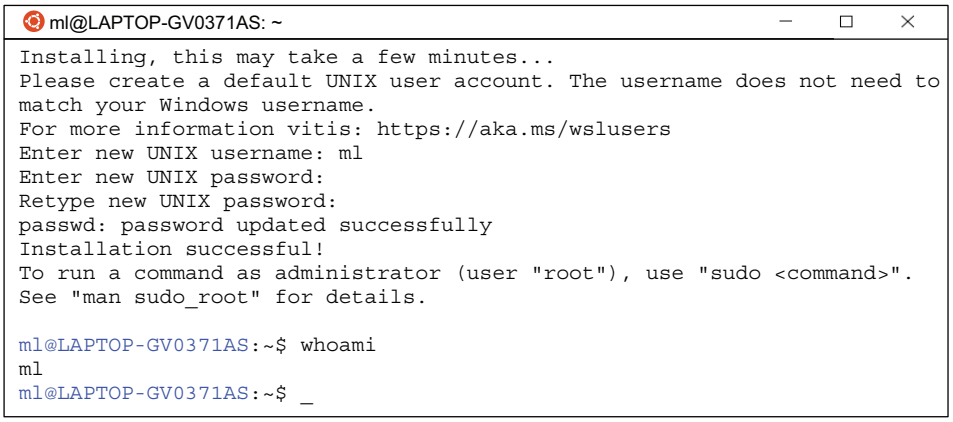


Figure A.5 The Ubuntu Terminal running on Windows

Now you can use the Ubuntu Terminal and follow the instructions for Linux to install Anaconda.

ANACONDA WINDOWS INSTALLER

Alternatively, we can use the Windows Installer for Anaconda. First, we need to download it from https://anaconda.com/distribution (figure A.6). Navigate to the Windows Installer section and download the 64-Bit Graphical Installer (or the 32-bit version, if you're using an older computer).

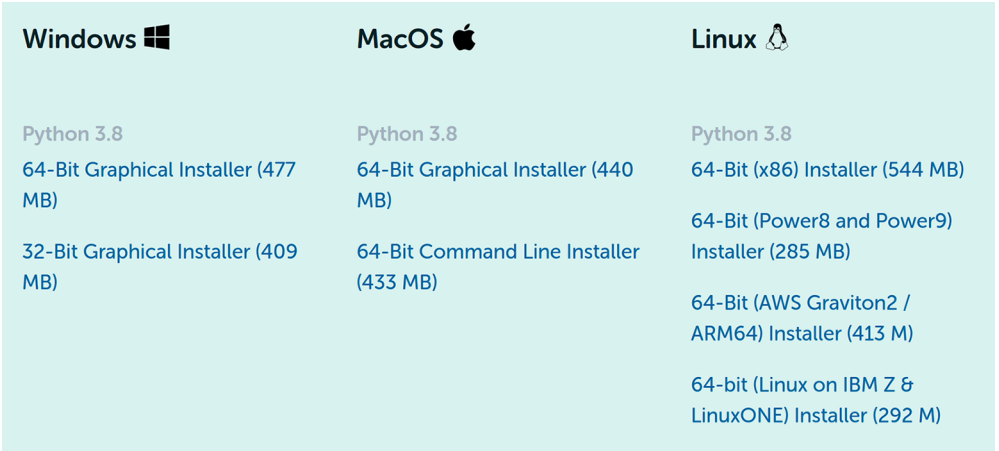


Figure A.6 Downloading the Windows Installer for Anaconda

Once you've downloaded the installer, simply run it and follow the setup guide (figure A.7).

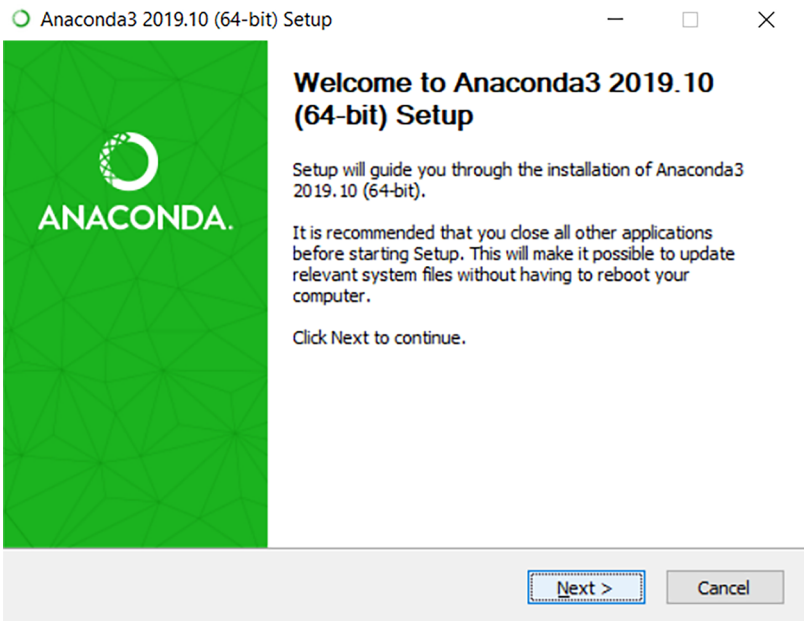


Figure A.7 The installer for Anaconda

It's pretty straightforward and you should have no problems running it. After the installation is successful, you should be able to run it by choosing Anaconda Navigator from the start menu.

A.1.3 Installing Python and Anaconda on macOS

The instructions for macOS should be similar to Linux and Windows: select the installer with the latest version of Python and execute it.

A.2 Running Jupyter

A.2.1 Running Jupyter on Linux

Once Anaconda is installed, you can run Jupyter. First, you need to create a directory that Jupyter will use for all the notebooks:

```
mkdir notebooks
```

Then `cd` to this directory to run Jupyter from there:

```
cd notebooks
```

It will use this directory for creating notebooks. Now let's run Jupyter:

```
jupyter notebook
```

This should be enough if you want to run Jupyter on a local computer. If you want to run it on a remote server, such as an EC2 instance from AWS, you need to add a few extra command-line options:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

In this case, you must specify two things:

- The IP address Jupyter will use to accept incoming HTTP requests (`--ip=0.0.0.0`). By default it uses localhost, meaning that it's possible to access the Notebook service only from within the computer.
- The `--no-browser` parameter, so Jupyter won't attempt to use the default web browser to open the URL with the notebooks. Of course, there's no web browser on the remote machine, only a terminal.

NOTE In the case of EC2 instances on AWS, you will also need to configure the security rules to allow the instance to receive requests on the port 8888. Please refer to the "Renting a server on AWS" section for more details.

When you run this command, you should see something similar to this:

```
[C 04:50:30.099 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-3510-open.html
    Or copy and paste one of these URLs:
        http://(ip-172-31-21-255 or 127.0.0.1):8888/
     ?token=670dfec7558c9a84689e4c3cdbb473e158d3328a40bf6bba
```

When starting, Jupyter generates a random token. You need this token to access the web page. This is for security purposes, so no one can access the Notebook service but you.

Copy the URL from the terminal, and replace (ip-172-31-21-255 or 127.0.0.1) with the instance URL. You should end up with something like this:

http://ec2-18-217-172-167.us-east-2.compute.amazonaws.com:8888/?token=f04317713e74e65289fe5a43dac43d5bf164c144d05ce613

This URL consists of three parts:

- The DNS name of the instance: if you use AWS, you can get it from the AWS console or by using the AWS CLI.
- The port (8888, which is the default port for the Jupyter notebooks service).
- The token you just copied from the terminal.

After that, you should be able to see the Jupyter Notebooks service and create a new notebook (figure A.8).

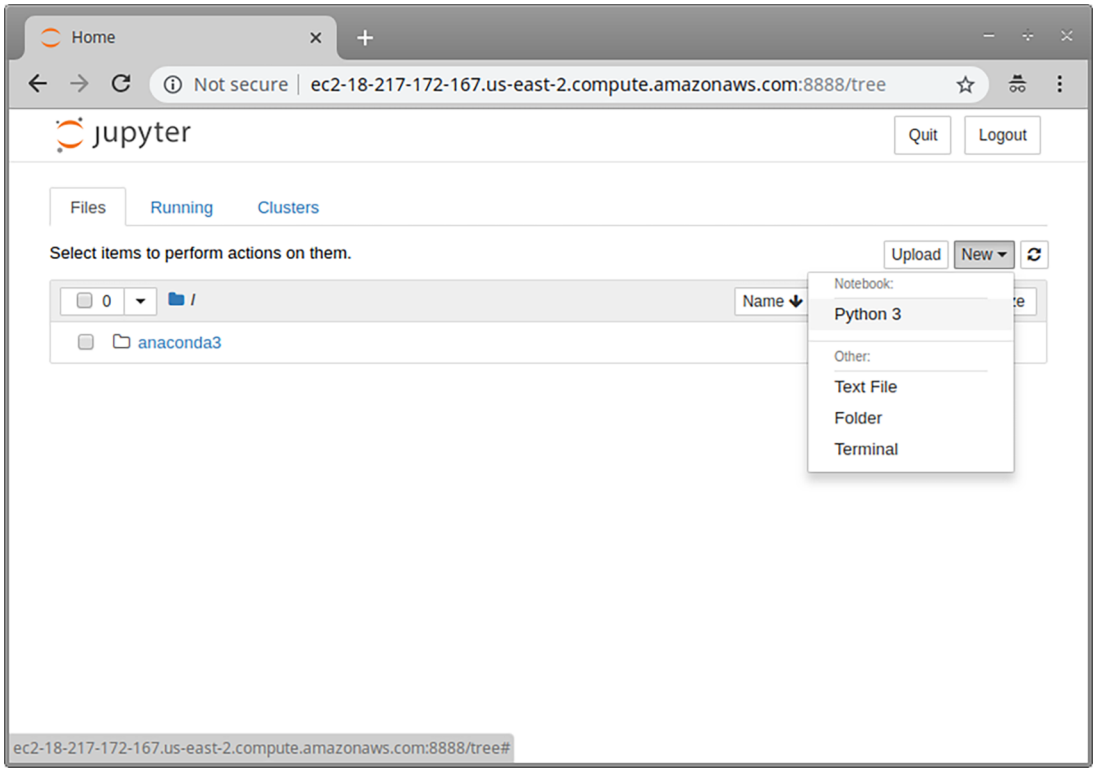


Figure A.8 The Jupyter Notebook service. Now you can create a new notebook.

If you're using a remote machine, when you exit the SSH session the Jupyter Notebook service will stop working. The internal process is attached to the SSH session, and it will be terminated. To avoid this, you can run the service inside screen, a tool for managing multiple virtual terminals:

```
screen -R jupyter
```

This command will attempt to connect to a screen with the name jupyter, but if no such screen exists, it will create one.

Then, inside the screen, you can type the same command for starting Jupyter Notebook:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

Check that it's working by trying to access it from your web browser. After verifying that it works, you can detach the screen by pressing Ctrl-A followed by D: first press

Ctrl-A, wait a bit, and then press D (for macOS, first press Ctrl-A and then press Ctrl-D). Anything running inside the screen is not attached to the current SSH session, so when you detach the screen and exit the session, the Jupyter process will keep running.

You can now disconnect from SSH (by pressing Ctrl-D) and verify that the Jupyter URL is still working.

A.2.2 Running Jupyter on Windows

As with Python and Anaconda, if you use the Linux Subsystem for Windows to install Jupyter, the instructions for Linux should work for Windows too.

By default, there's no browser configured to run in the Linux Subsystem. So we need to use the following command for launching Jupyter:

```
jupyter notebook --no-browser
```

Alternatively, we can set the `BROWSER` variable to point it to a browser from Windows:

```
export BROWSER='/mnt/c/Windows/explorer.exe'
```

However, if you didn't use the Linux Subsystem and installed Anaconda using the Windows Installer, starting the Jupyter Notebook service is different.

First, we need to open the Anaconda Navigator in the start menu. Once it's open, find Jupyter in the Applications tab and click Launch (figure A.9).

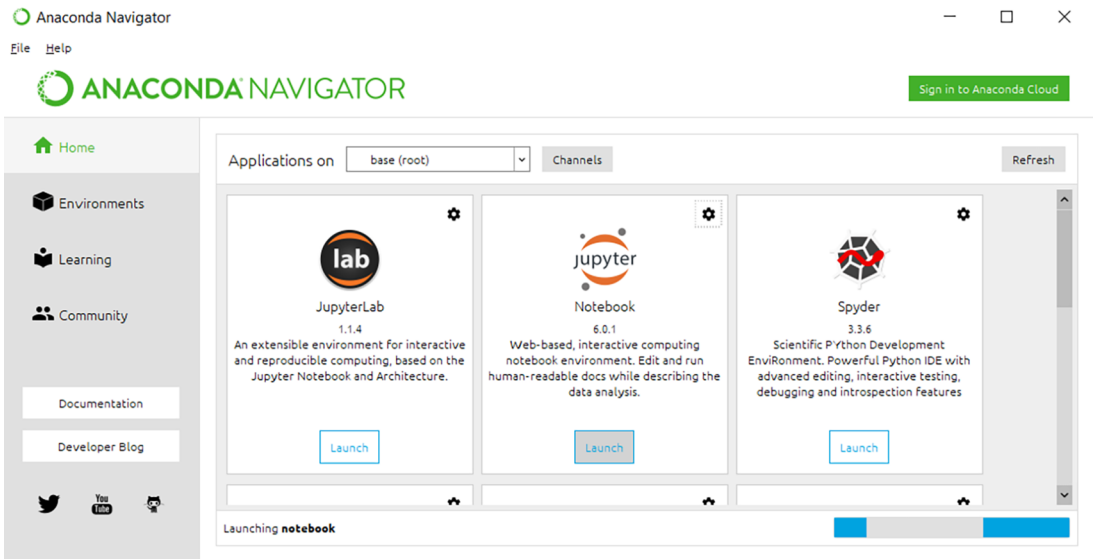


Figure A.9 To run the Jupyter Notebook service, find Jupyter in the applications tab and click Launch.

After the service launches successfully, the browser with Jupyter should open automatically (figure A.10).

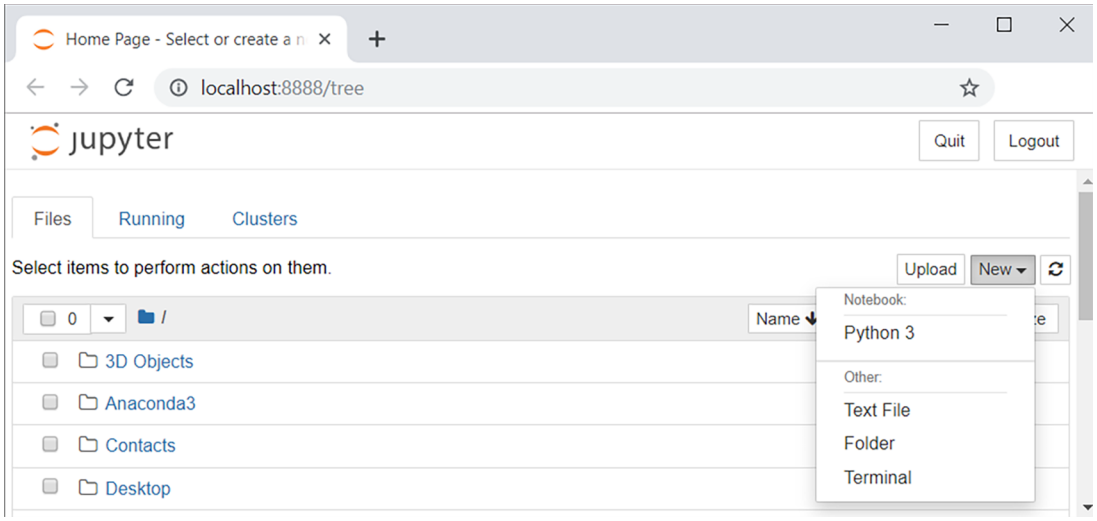


Figure A.10 The Jupyter Notebook service launched using Anaconda Navigator

A.2.3 Running Jupyter on MacOS

The instructions for Linux should also work for macOS, with no additional changes.

A.3 Installing the Kaggle CLI

The Kaggle CLI is the command-line interface for accessing the Kaggle platform, which includes data from Kaggle competitions and Kaggle datasets.

You can install it using pip:

```
pip install kaggle --upgrade
```

Then you need to configure it. First, you need to get credentials from Kaggle. For that, go to your Kaggle profile (create one if you don't have one yet), located at https://www.kaggle.com/<username>/account. The URL will be something like https://www.kaggle.com/agrigorev/account.

In the API section, click Create New API Token (figure A.11).

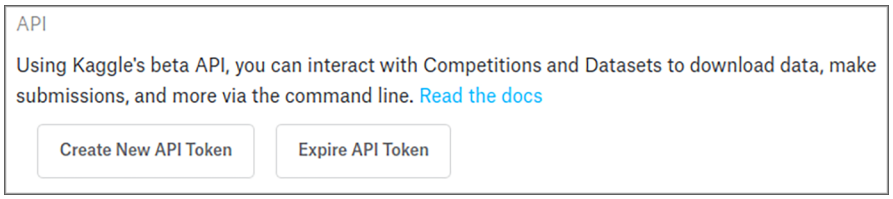


Figure A.11 To generate an API token to use from the Kaggle CLI, click Create New API Token on your Kaggle account page.

This will download a file called kaggle.json, which is a JSON file with two fields: `username` and `key`. If you're configuring the Kaggle CLI on the same computer you used to download the file, you should simply move this file to the location where the Kaggle CLI expects it:

```
mkdir ~/.kaggle
mv kaggle.json ~/.kaggle/kaggle.json
```

If you're configuring it on a remote machine, such as an EC2 instance, you need to copy the content of this file and paste it into the terminal. Open the file using nano (this will create the file if it doesn't exist):

```
mkdir ~/.kaggle
nano ~/.kaggle/kaggle.json
```

Paste in the content of the kaggle.json file you downloaded. Save the file by pressing Ctrl-O and exit nano by pressing Ctrl-X.

Now test that it's working by trying to list the available datasets:

```
kaggle datasets list
```

You can also test that it can download datasets by trying the dataset from chapter 2:

```
kaggle datasets download -d CooperUnion/cardataset
```

It should download a file called cardataset.zip.

A.4 Accessing the source code

We've stored the source code for this book on GitHub, a platform for hosting source code. You can see it here: https://github.com/alexeygrigorev/mlbookcamp-code.

GitHub uses Git to manage code, so you'll need a Git client to access the code for this book.

Git comes preinstalled in all the major Linux distributions. For example, the AMI we used for creating an instance with Ubuntu on AWS already has it.

If your distribution doesn't have Git, it's easy to install it. For example, for Debian-based distributions (such as Ubuntu), you need to run the following command:

```
sudo apt-get install git
```

On macOS, to use Git you need to install Command Line Tools or, alternatively, download the installer at https://sourceforge.net/projects/git-osx-installer/.

For Windows, you can download Git at https://git-scm.com/download/win.

Once you have Git installed, you can use it to get the book's code. To access it, you need to run the following command:

```
git clone https://github.com/alexeygrigorev/mlbookcamp-code.git
```

Now you can run Jupyter Notebook:

```
cd mlbookcamp-code
jupyter notebook
```

If you don't have Git and don't want to install it, it's also possible to access the code without it. You can download the latest code in a zip archive and unpack it. On Linux, you can do that by executing these commands:

```
wget -O mlbookcamp-code.zip \
    https://github.com/alexeygrigorev/mlbookcamp-code/archive/master.zip
unzip mlbookcamp-code.zip
rm mlbookcamp-code.zip
```

You can also just use your web browser: type the URL, download the zip archive, and extract the content.

A.5 Installing Docker

In chapter 5, we use Docker to package our application in an isolated container. It's quite easy to install.

A.5.1 Installing Docker on Linux

These steps are based on the official instructions for Ubuntu from the Docker website (https://docs.docker.com/engine/install/ubuntu/).

First, we need to install all the prerequisites:

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl software-
     properties-common
```

Next, we add the repository with the Docker binaries:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/
     ubuntu $(lsb_release -cs) stable"
```

Now we can install it:

```
sudo apt-get update
sudo apt-get install docker-ce
```

Finally, if we want to execute Docker commands without sudo, we need to add our user to the `docker` user group:

```
sudo adduser $(whoami) docker
```

Now you'll need to reboot your system. In the case of EC2 or another remote machine, just logging off and on is enough.

To test that everything works fine, run the `hello-world` container:

```
docker run hello-world
```

You should see a message saying that everything works:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

A.5.2 Installing Docker on Windows

To install Docker on Windows, you need to download the installer from the official website (https://hub.docker.com/editions/community/docker-ce-desktop-windows/) and simply follow the instructions.

A.5.3 Installing Docker on MacOS

Like with Windows, installing Docker on MacOS is simple: first, download the installer from the official website (https://hub.docker.com/editions/community/docker-ce-desktop-mac/), and then follow the instructions.

A.6 Renting a server on AWS

Using a cloud service is the easiest way of getting a remote machine that you can use for following the examples in the book.

There are quite a few options nowadays, including cloud computing providers like Amazon Web Services (AWS), Google Cloud Platform, Microsoft Azure, and Digital Ocean. Rather than having to rent a server for a long time, in the cloud you can use it for a short period and typically pay per hour, per minute, or even per second. You can select the best machine for your needs in terms of computing power (number of CPUs or GPUs) and RAM.

It's also possible to rent a dedicated server for a longer time and pay per month. If you intend to use the server for a long time — say, six months or more — renting a dedicated server will be cheaper. Hetzner.com might be a good option in this case. They also offer servers with GPUs.

To make it easier for you to set up the environment with all the required libraries for the book, we provide instructions here for setting up an EC2 (Elastic Compute Cloud) machine on AWS. EC2 is part of AWS and allows you to rent a server of any configuration for any duration of time.

> **NOTE** We're not affiliated with Amazon or AWS. We chose to use it in this book because at the time of writing it's the most commonly used cloud provider.

If you don't have an AWS account or only recently created it, you're eligible for the free tier: you have a 12-month trial period in which to check out most of the AWS products for free. We try to use the free tier whenever possible, and we will specifically mention if something isn't covered by this tier.

Note that the instructions in this section are optional, and you don't have to use AWS or any other cloud. The code should work on any Linux machine, so if you have

a laptop with Linux, it should be enough to work through the book. A Mac or Windows computer should also be fine, but we haven't tested the code thoroughly on these platforms.

A.6.1 Registering on AWS

The first thing you need to do is create an account. To do this, go to https://aws.amazon.com and click the Create an AWS Account button (see figure A.12).

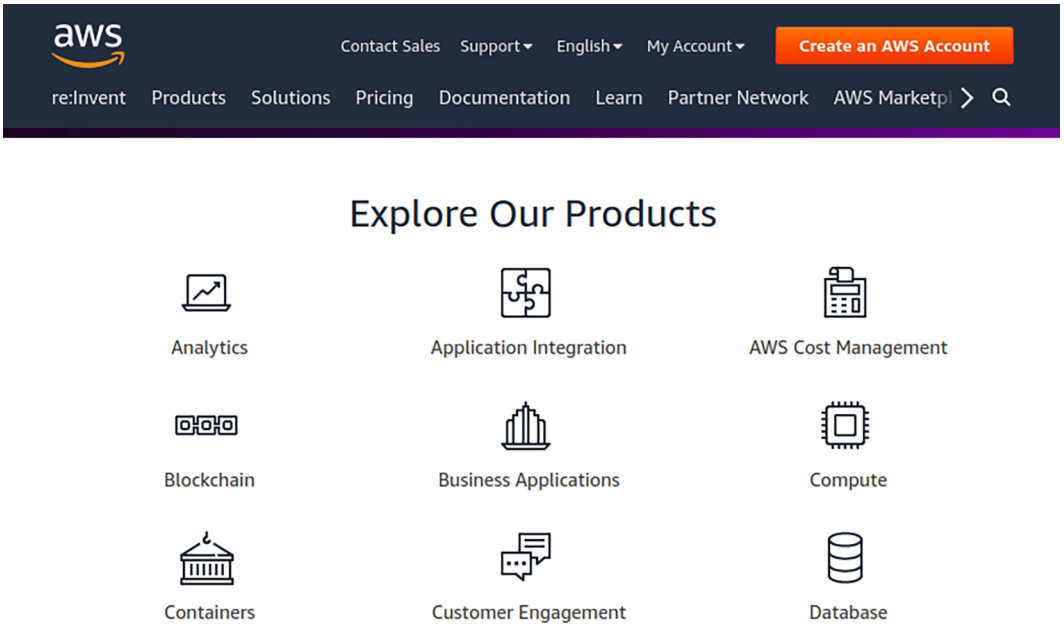


Figure A.12 To create an account, click Create an AWS Account on the main AWS page.

> **NOTE** This appendix was written in October 2019 and the screenshots were taken at that time. Please be aware that content on the AWS web site and the appearance of the management console could change.

Follow the instructions and fill in the required details. It should be a straightforward process, similar to the process of registering on any website.

> **NOTE** Please be aware that AWS will ask you to provide the details of a bank card during the registration process.

Once you've completed the registration and verified your account, you should see the main page — the AWS Management Console (figure A.13).

Congratulations! You've just created a root account. However, it's not advisable to use the root account for anything: it has very broad permissions that allow you to do

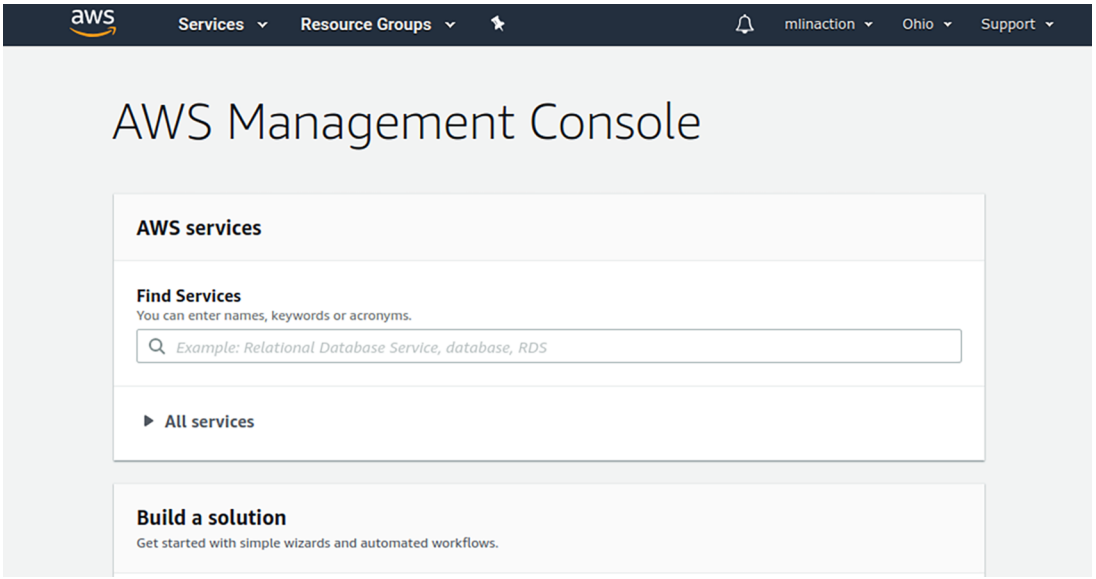


Figure A.13 The AWS Management Console is the starting page for AWS.

anything and everything on your AWS accunt. Typically, you use the root account to create less powerful accounts and then use them for your day-to-day tasks.

To create such an account, type "IAM" in the Find Services box and click on that item in the drop-down list. Select Users in the menu on the left, and click Add User (see figure A.14).

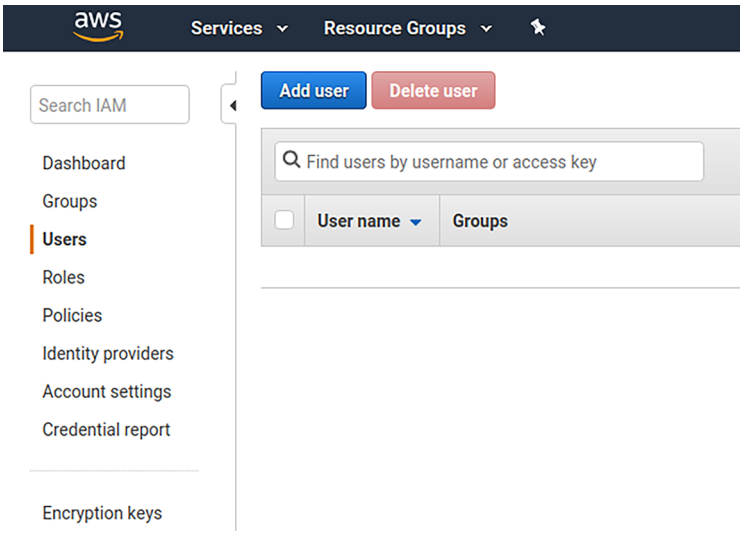


Figure A.14 Adding a user in the AWS Identity and Access Management (IAM) service

Now you just need to follow the instructions and answer the questions. At some point, it will ask about an access type: you'll need to select both Programmatic Access and AWS Management Console Access (see figure A.15). We will use both the command-line interface (CLI) and the web interface for working with AWS.

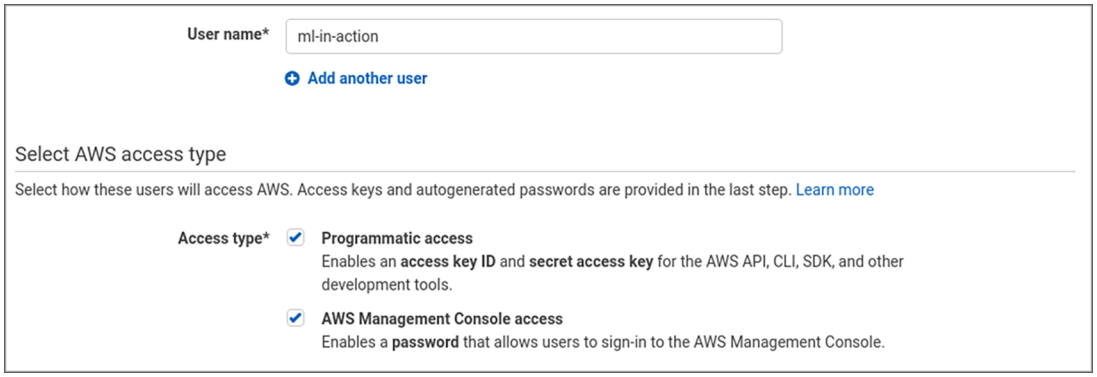


Figure A.15 We will use both the web interface and the command-line interface for working with AWS, so you need to select both access types.

In the Set Permissions step, you specify what this new user will be able to do. You want the user to have full privileges, so select Attach Existing Policies Directly at the top and choose AdministratorAccess in the list of policies (see figure A.16).

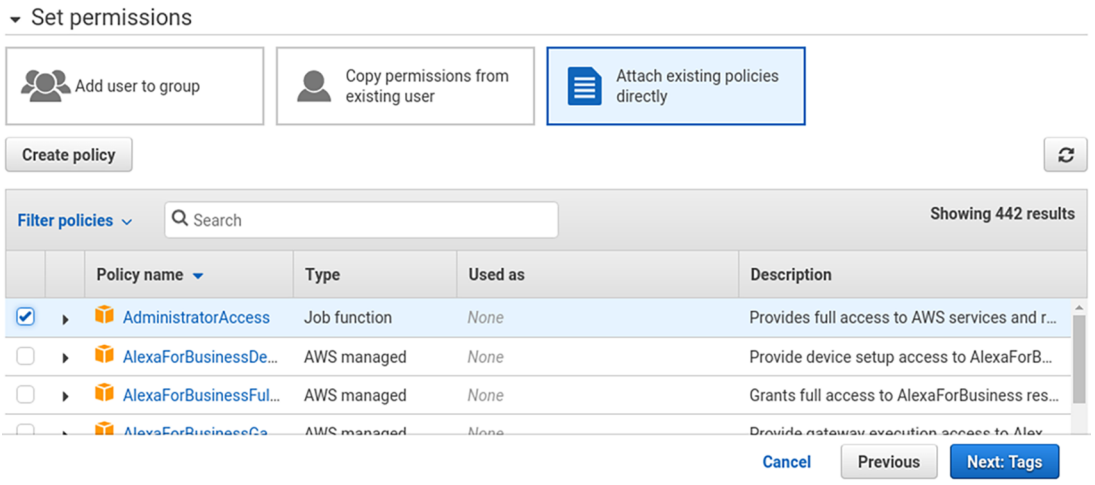


Figure A.16 Select the AdministratorAccess policy to enable the new user to access everything on AWS.

As the next step, the system will ask you about tags — you can safely ignore these for now. Tags are needed for companies where multiple people work on the same AWS account, mostly for expense-management purposes, so they shouldn't be a concern for the projects you'll do in this book.

At the end, when you've successfully created the new user, the wizard will suggest that you download the credentials (figure A.17). Download them and keep them safe; you'll need to use them later when configuring the AWS CLI.

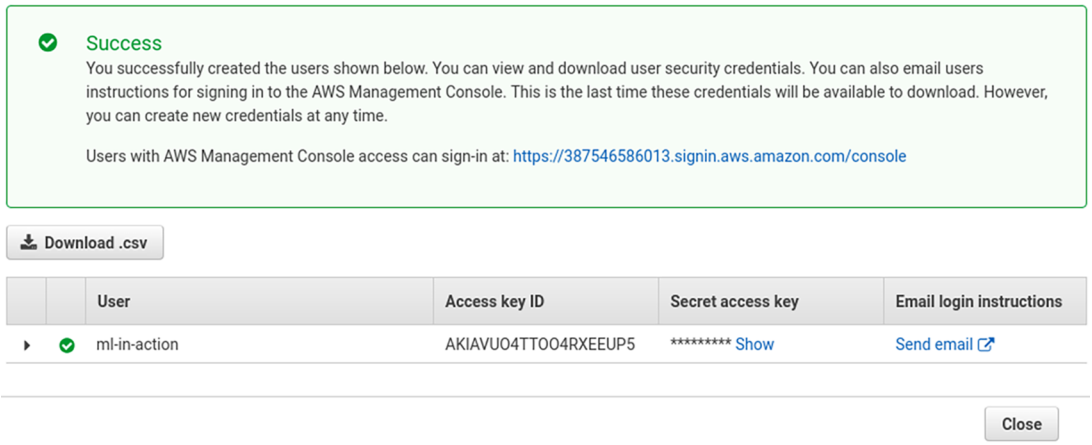


Figure A.17 The details for the newly created user. You can see the sign-in URL and download the credentials for programmatic access.

To access the management console, you can use the link AWS has generated for you. It appears in the Success box and follows this pattern:

https://<accountid>.signin.aws.amazon.com/console

It might be a good idea to bookmark this link. Once AWS has validated the account (which can take a little while), you can use it to log in: simply provide the username and password you specified when creating the user.

You can now start using the services of AWS. Most importantly, you can create an EC2 machine.

A.6.2 Accessing billing information

When using a cloud service provider, you are typically charged per second: for every second you use a particular AWS service, you pay a pre-defined rate. At the end of each month you get a bill, which is typically processed automatically. The money is withdrawn from the bank card you linked to the AWS account.

IMPORTANT Even though we use the free tier to follow most of the examples in the book, you should periodically check the billing page to make sure you're not accidentally using billable services.

To understand how much you will need to pay at the end of the month, you can access the billing page of AWS.

If you use the root account (the account you created first), simply type "Billing" on the homepage of the AWS console to navigate to the billing page (figure A.18).

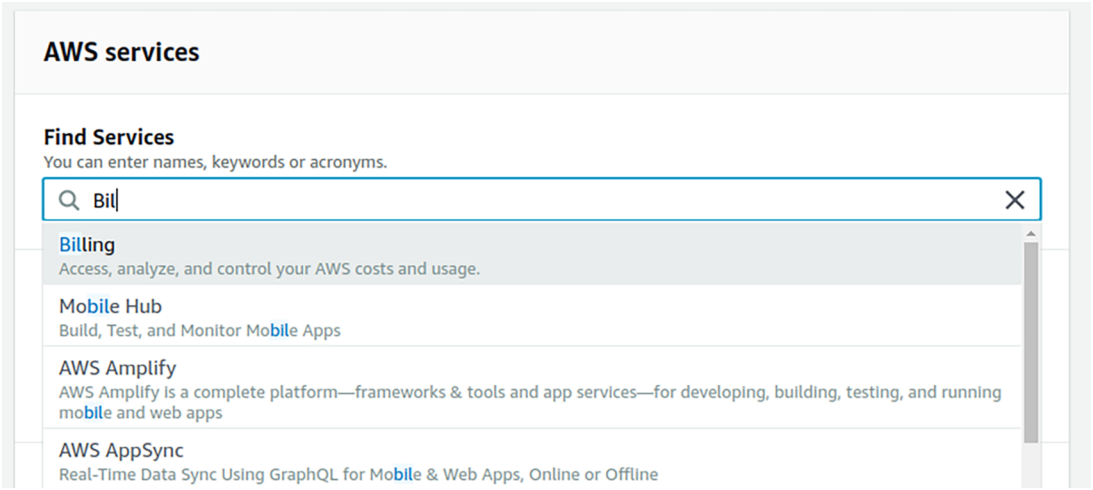
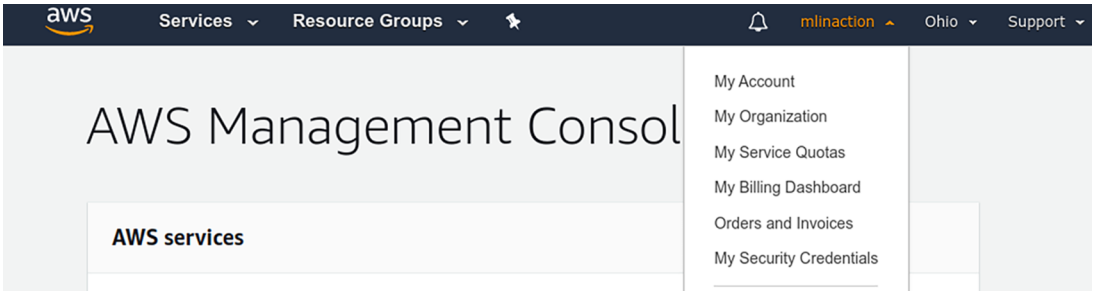


Figure A.18 To get to the billing page, type in "Billing" in the quick access search box.

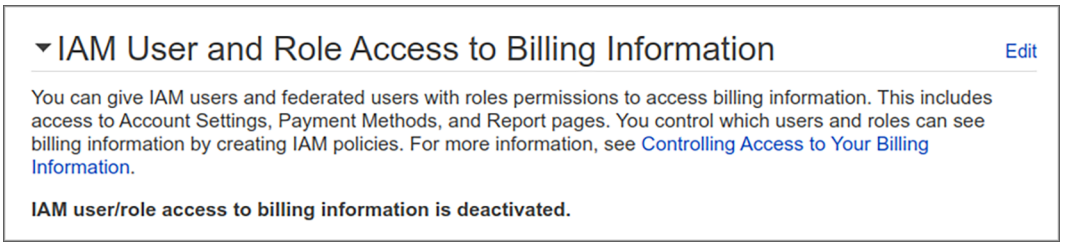
If you try to access the same page from the user account (or IAM user — the one we created after creating the root account), you will notice that this is not allowed. To fix that, you need to

- Allow the billing page to be accessed by all IAM users, and
- Give the AMI user permissions to access the billing page.

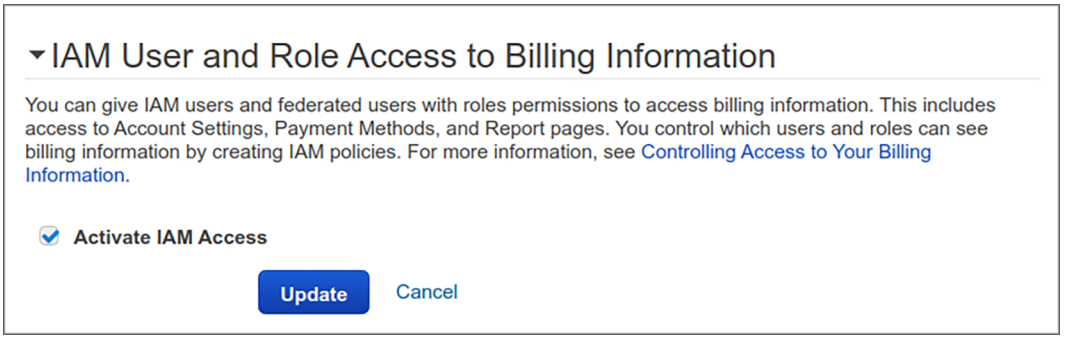
Allowing all the IAM users access the billing page is simple: go to My Account (figure A.19 A), go to the IAM User and Role Access to Billing Information section and click Edit (figure A.19 B), and then select the Activate IAM Access option and click Update (figure A.19 C).



(A) To allow AMI users to access the billing info, click on My Account.



(B) In the My Account settings, find the IAM User and Role Access to Billing Information section and click Edit.



(C) Enable the Activate IAM Access option and click Update.

Figure A.19 Enabling access to billing information to IAM users

After that, go to the IAM service, find the IAM user we previously created, and click on it. Next, click on the Add permissions button (figure A.20).

Then attach the existing Billing policy to the user (figure A.21).

After that, the IAM user should be able to access the billing information page.

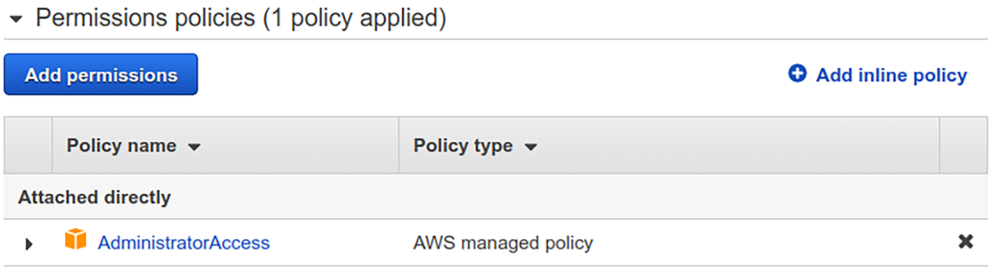


Figure A.20 To allow the IAM user access the billing information, we need to add special permissions for that. To do so, click on the Add permissions button.

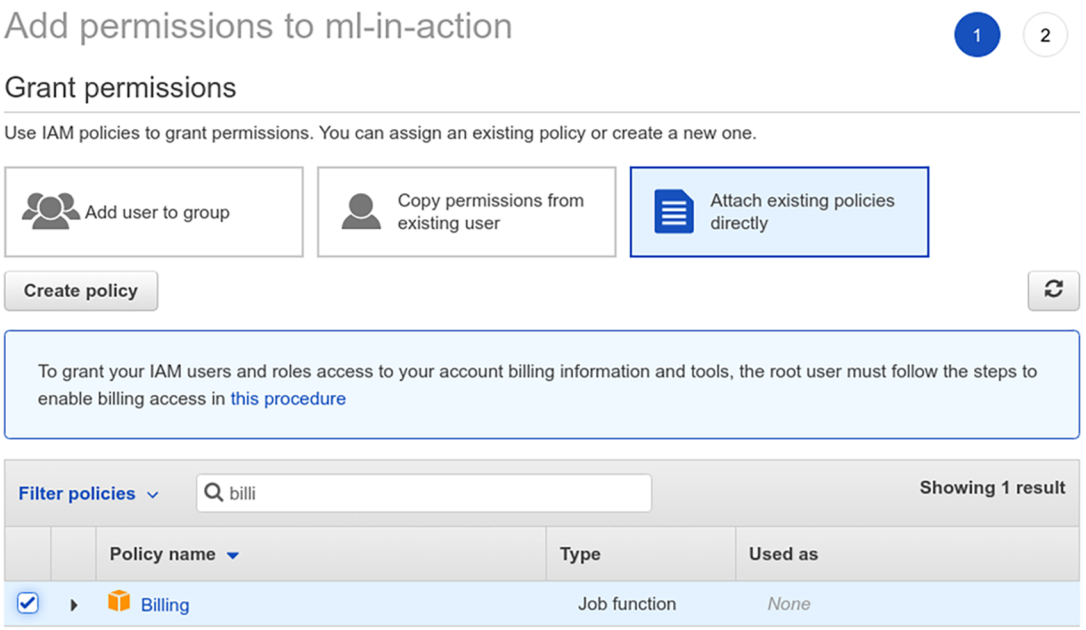


Figure A.21 After clicking on the Add permissions button, select the Attach existing policies directly option and select Billing in the list.

A.6.3 Creating an EC2 instance

EC2 is a service for renting a machine from AWS. You can use it to create a Linux machine to use for the projects in this book. To do this, first go to the EC2 page in AWS. The easiest way to do this is by typing "EC2" in the Find Services box on the home page of the AWS Management Console; then select EC2 from the drop-down list and press Enter (figure A.22).

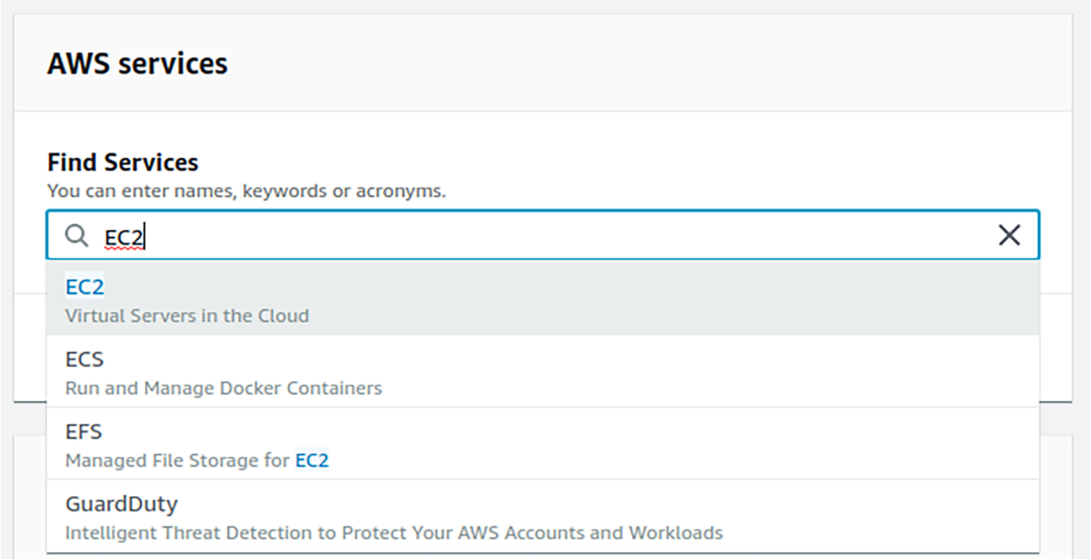


Figure A.22 To go to the EC2 service's page, type EC2 in the Find Services box on the main page of the AWS Management Console and press Enter.

On the EC2 page, choose Instances from the menu on the left and then click Launch Instance (figure A.23).

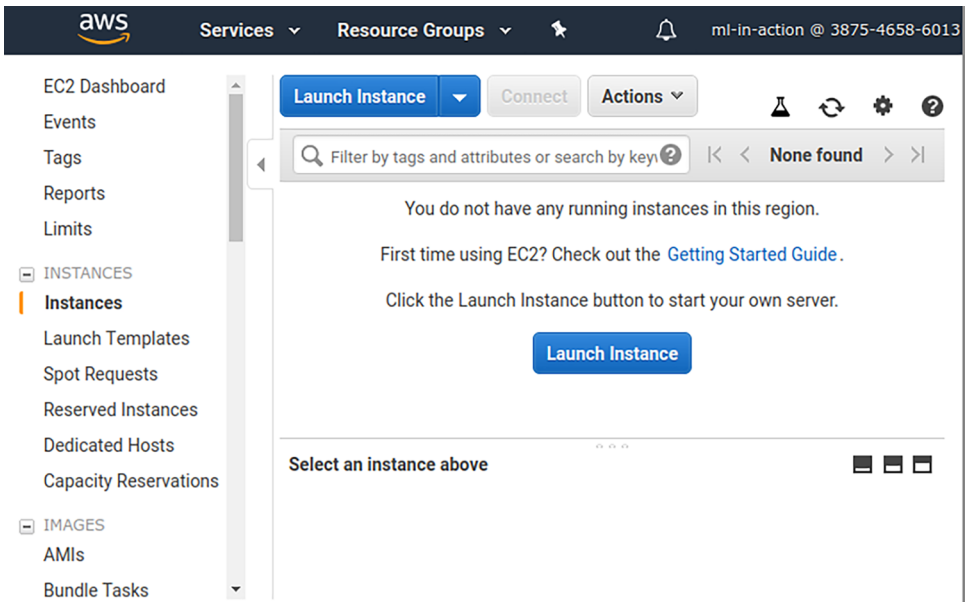


Figure A.23 To create an EC2 instance, select Instances in the menu on the left and click Launch Instance.

This brings you to a six-step form. The first step is to specify the AMI (Amazon Machine Image) you'll use for the instance. We recommend Ubuntu: it's one of the most popular Linux distributions, and we used it for all the examples in this book. Other images should also work fine, but we haven't tested them.

At the time of writing, Ubuntu Server 20.04 LTS is available (figure A.24), so use that one. Find it in the list and then click Select.

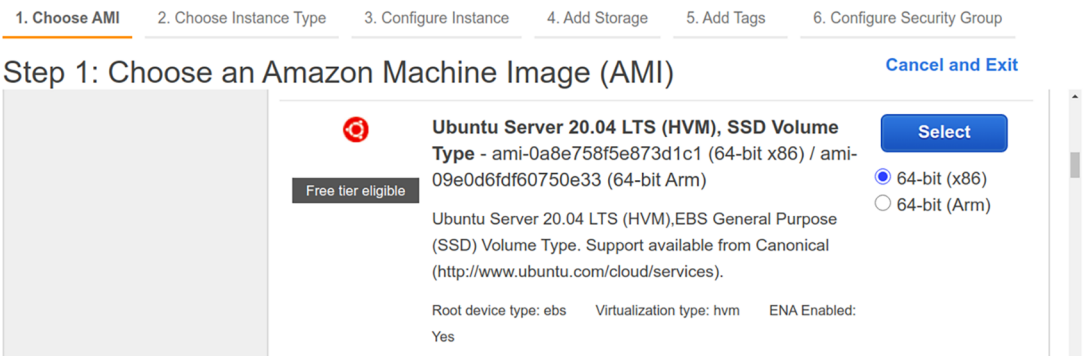


Figure A.24 Your instance will be based on Ubuntu Server 18.04 LTS.

You should take note of the AMI's ID: in this example it's ami-0a8e758f5e873d1c1, but it might be different for you, depending on your AWS region and version of Ubuntu.

> **NOTE** This AMI is free-tier eligible, which means that if you use the free tier for testing AWS, you won't be charged for using this AMI.

After that you need to select the instance type. There are many options, with different numbers of CPU cores and different amounts of RAM. If you want to stay within the free tier, select t2.micro (figure A.25). It's a rather small machine: it has only 1 CPU and 1 GB RAM. Of course, it's not the best instance in terms of computing power, but it should be enough for many projects in this book.

The next step is where you configure the instance details. You don't need to change anything here and can simply go on to the next step: adding storage (figure A.26).

Here, you specify how much space you need on the instance. The default suggestion of 8GB is not enough, so select 18GB. This should be good enough for most of the projects we'll do in this book. After changing it, click Next: Add Tags.

In the next step, you add tags to your new instance. The only tag you should add is Name, which allows you to give an instance a human-readable name. Add the key

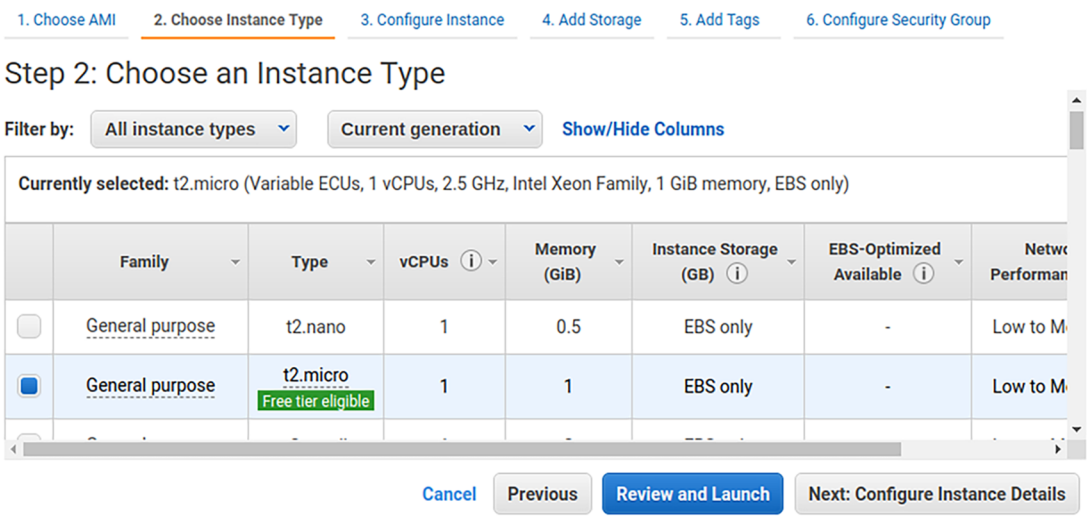


Figure A.25 The t2.micro is a rather small instance with only 1 CPU and 1 GB RAM, but it can be used for free.

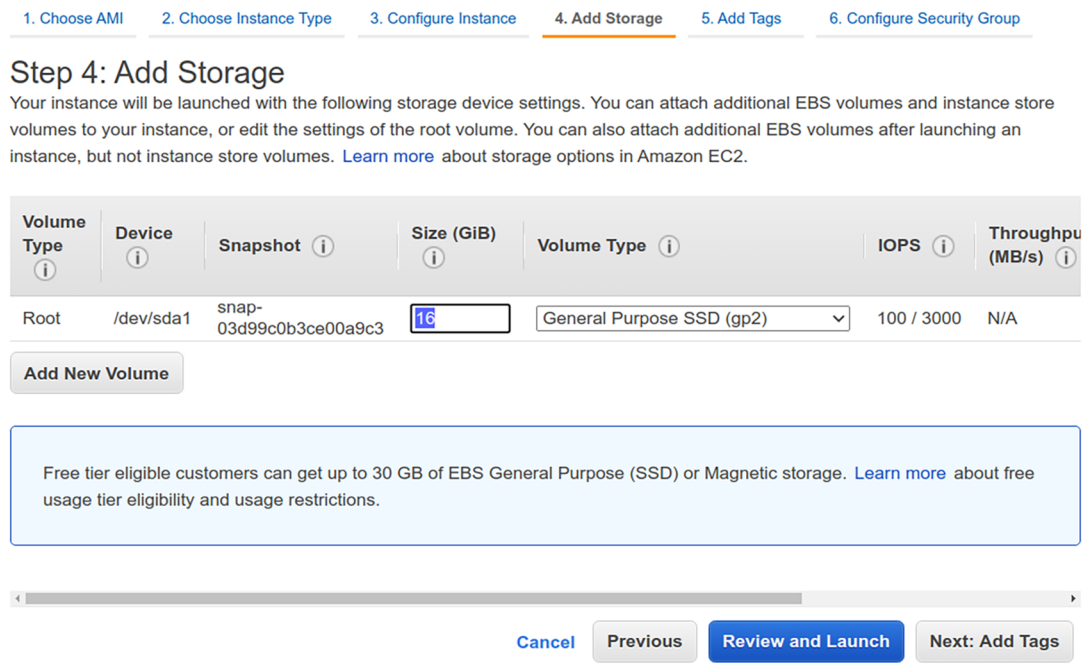


Figure A.26 The fourth step of creating an EC2 instance in AWS: adding storage. Change the size to 16GB.

Name and the value ml-bookcamp-instance (or any other name you prefer), as seen in figure A.27.

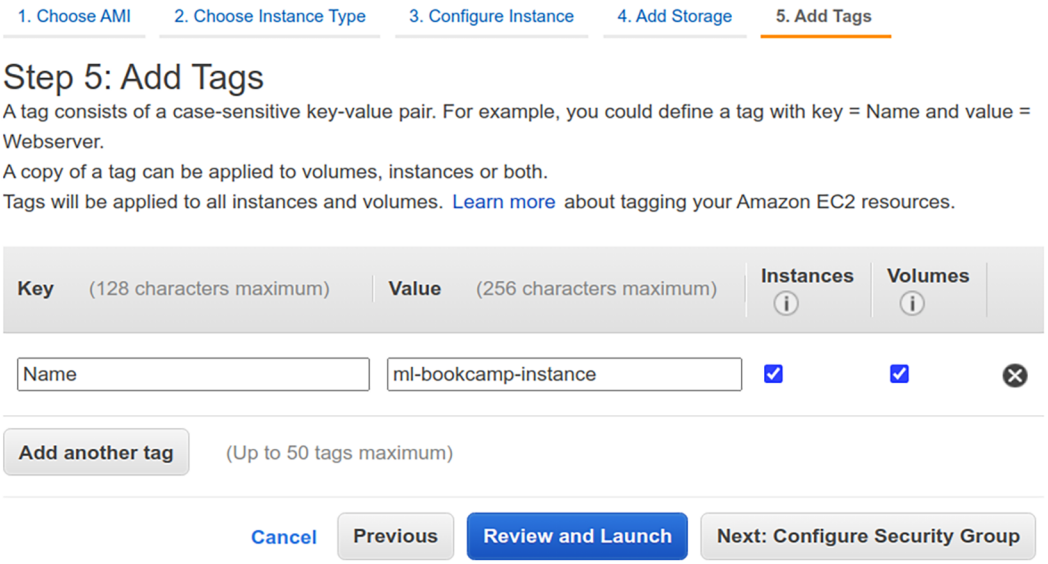


Figure A.27 The only tag you may want to specify in step 5 is "Name": it allows you to give a human-readable name to the instance.

The next step is quite an important one: choosing the security group. This allows you to configure the network firewall and specify how the instance can be accessed and which ports are open. You'll want to host Jupyter Notebook on the instance, so you need to make sure its port is open and you can log in to the remote machine.

Because you don't yet have any security groups in your AWS account, you'll need to create a new one now: choose Create a New Security Group and give it the name jupyter (figure A.28). You'll want to use SSH to connect to the instance from your computers, so you need to make sure SSH connections are allowed. To enable this, select SSH in the Type drop-down list in the first row.

Typically the Jupyter Notebook service runs on port 8888, so you need to add a custom TCP rule that port 8888 can be accessed from anywhere on the internet.

When you do this, you may see a warning telling you that this might not be safe (figure A.29). It's not a problem for us because we are not running anything critical on the instances. Implementing proper security is not trivial and is out of scope for this book.

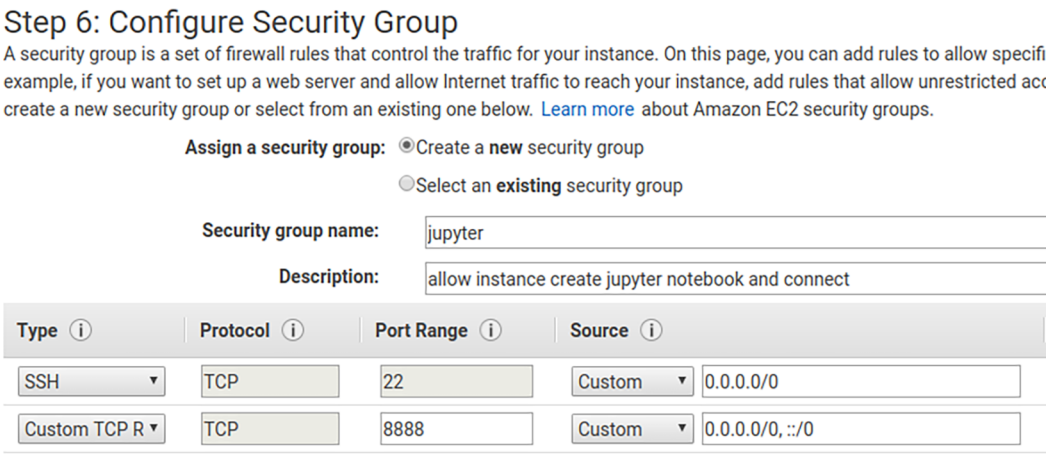


Figure A.28 Creating a security group for running Jupyter Notebook on EC2 instances

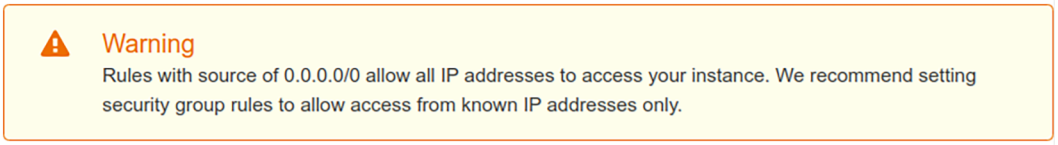


Figure A.29 AWS warns us that the rules we added are not strict. For our case it's not a problem and we can safely ignore the warning.

The next time you create an instance, you'll be able to reuse this security group instead of creating a new one. Choose Select an Existing Security Group and select it from the list (figure A.30).

Configuring the security group is the last step. Verify that everything is fine, and click Review and Launch.

AWS won't let you launch the instance yet: you still need to configure the SSH keys for logging into the instance. Because your AWS account is still fresh and doesn't have keys yet, you need to create a new key pair. Choose Create a New Key Pair from the drop-down list and give it the name jupyter (figure A.31).

Click Download Key Pair and save the file somewhere on your computer. Make sure you can access this file later; it's important for being able to connect to the instance.

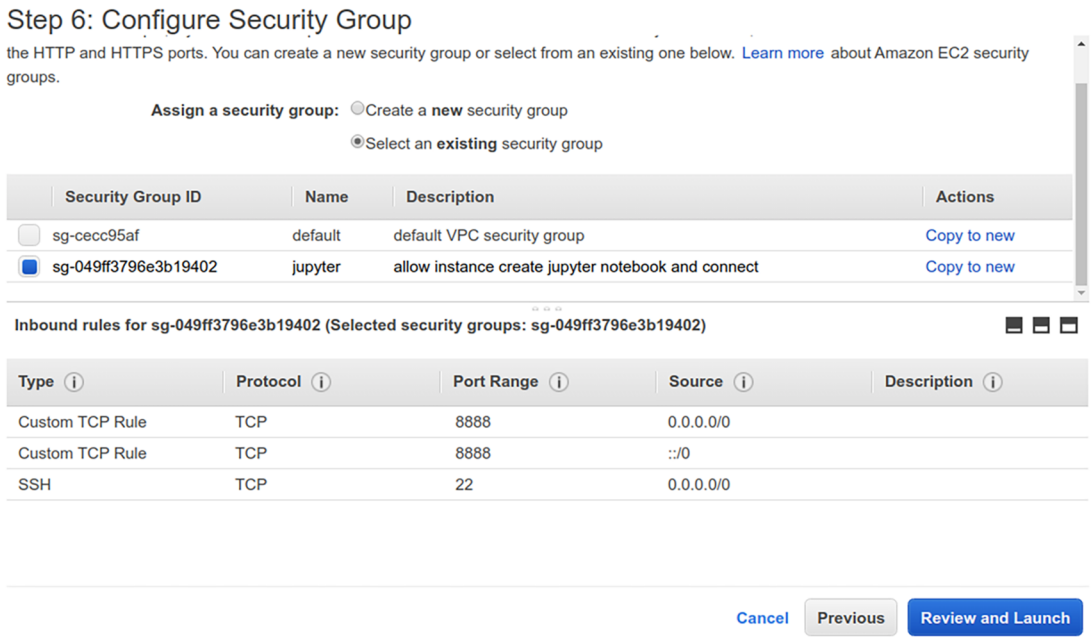


Figure A.30 When creating an instance, it's also possible to assign an existing security group to the instance.

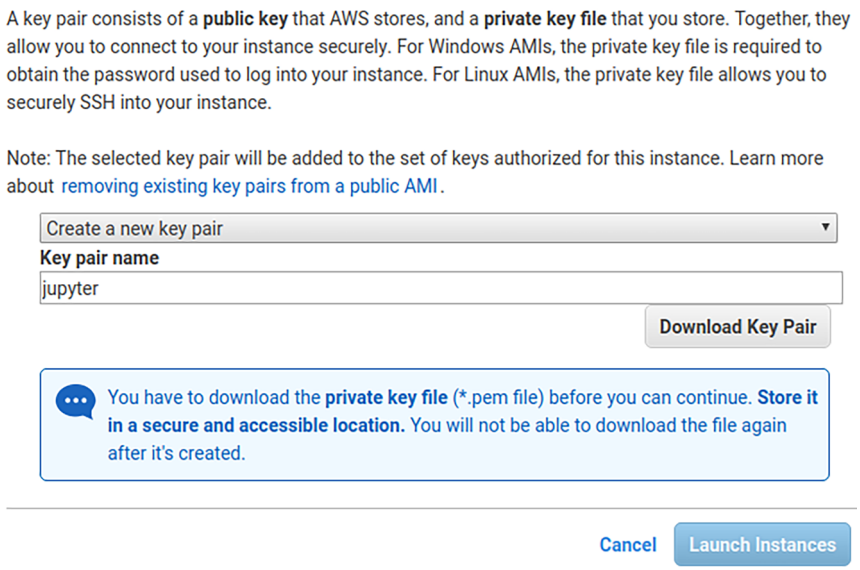


Figure A.31 To be able to use SSH to log in to the instance, you need to create a key pair.

The next time you create an instance, you can reuse this key. Select Choose an Existing Key Pair in the first drop-down list, choose the key you want to use, and click the checkbox to confirm that you still have the key (figure A.32).

Figure A.32 You can also use an existing key when creating an instance.

Now you can launch the instance by clicking Launch Instances. You should see a confirmation that everything is good and the instance is launching (figure A.33).

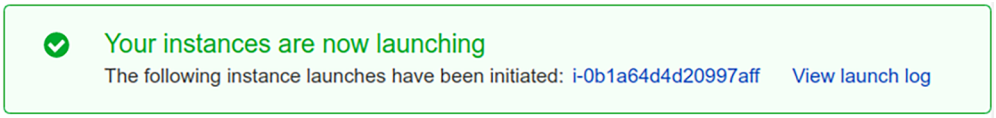


Figure A.33 AWS tells us that everything went well and now the instance is launching.

In this message, you can see the ID of the instance. In our case, it's i-0b1a64d4d20997aff. You can click on it now to see the details of the instance (figure A.34). Because you want to use SSH to connect to your instance, you need to get the public DNS name to do it. You can find this on the Description tab.

A.6.4 Connecting to the instance

In the previous section you created an instance on EC2. Now you need to log in to this instance to install all the required software. You will use SSH for this.

Connecting to the instance on Linux

You already know the public DNS name of your instance. In our example, it's ec2-18-191-156-172.us-east-2.compute.amazonaws.com. In your case the name will be different: the first part of the name (ec2-18-191-156-172) depends on the IP that the instance gets and the second (us-east-2) on the region where it's running. To use SSH to enter the instance, you will need this name.

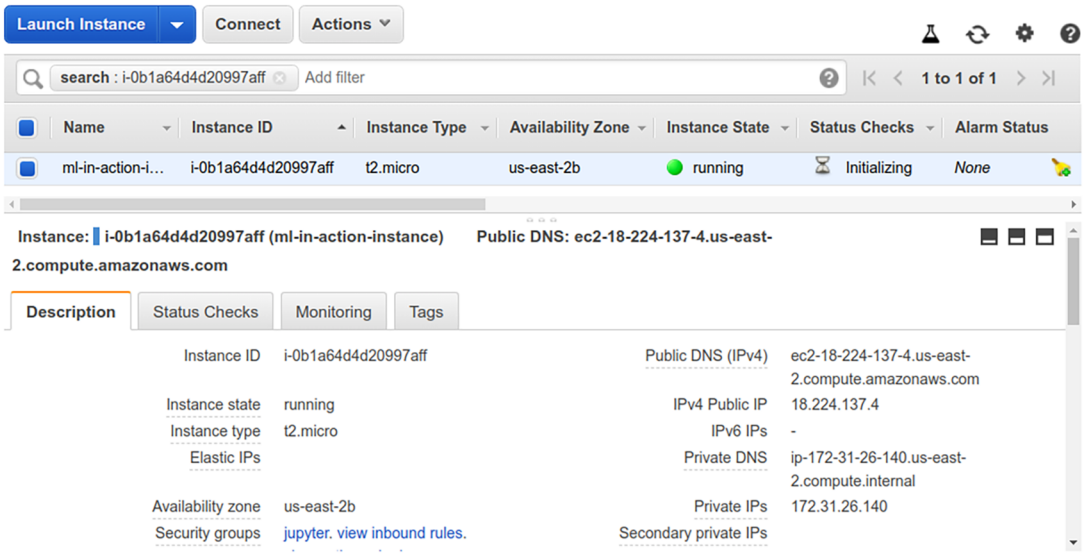


Figure A.34 The details of the newly created instance. To use SSH to connect to it, you need the public DNS name.

When using the key you downloaded from AWS for the first time, you need to make sure the permissions on the file are set correctly. Execute this command:

```
chmod 400 jupyter.pem
```

Now you can use the key to log in to the instance:

```
ssh -i "jupyter.pem" \
    ubuntu@ec2-18-191-156-172.us-east-2.compute.amazonaws.com
```

Of course, you should replace the DNS name shown here with the one you copied from the instance description.

Before allowing you to enter the machine, the SSH client will ask you to confirm that you trust the remote instance:

```
The authenticity of host 'ec2-18-191-156-172.us-east-2.compute.amazonaws.com
     (18.191.156.172)' can't be established.
ECDSA key fingerprint is SHA256:S5doTJOGwXVF3i1IFjB10RuHufaVSe+EDqKbGpIN0wI.
Are you sure you want to continue connecting (yes/no)?
```

Type "yes" to confirm.

Now you should be able to log in to the instance and see the welcome message (figure A.35).

Now it's possible to do anything you want with the machine.

```
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-1032-aws x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

System information as of Wed Jun  5 06:01:51 UTC 2019

System load:     0.02             Processes:       86
Usage of /:      13.6% of 7.69GB  Users logged in: 0
Memory usage:    14%              IP address for eth0: 172.31.46.216
Swap usage:      0%

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-46-216:~$
```

Figure A.35 After successfully logging into the EC2 instance, you should see the welcome message.

CONNECTING TO THE INSTANCE ON WINDOWS

Using the Linux subsystem on Windows is the easiest way for connecting to the EC2 instance: you can use SSH there and follow the same instructions as for Linux.

Alternatively, you can use Putty (https://www.putty.org) for connecting to EC2 instances from Windows.

CONNECTING TO THE INSTANCE ON MACOS

SSH is built in on macOS, so the steps for Linux should work on a Mac.

A.6.5 Shutting down the instance

After you've finished working with the instance, you should turn it off.

> **IMPORTANT** It's very important to turn off the instance after the work is finished. For each second you use the instance, you get billed, even if you no longer need the machine and it's idle. That doesn't apply in the first 12 months of using AWS if the requested instance is free-tier eligible, but nonetheless, it's good to develop the habit of periodically checking your account status and disabling unneeded services.

You can do this from the terminal:

```
sudo shutdown now
```

It's also possible to do it from the web interface: select the instance you want to turn off, go to Actions, and select Instance State > Stop (figure A.36).

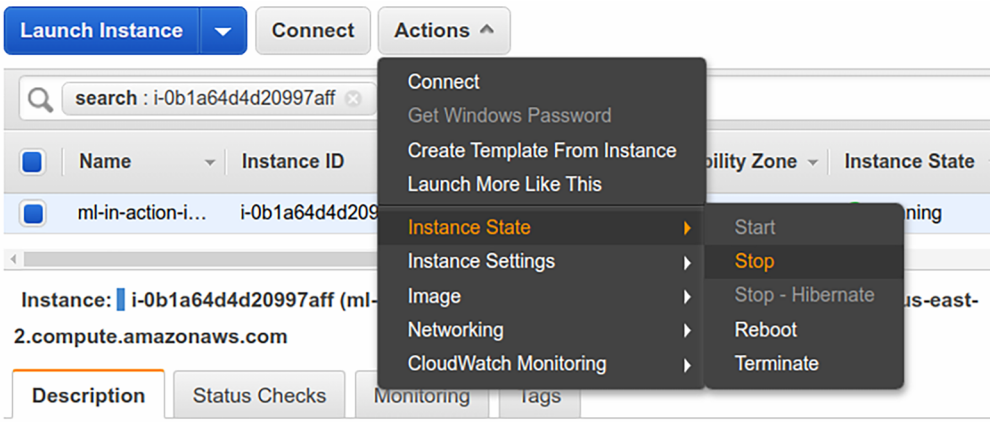


Figure A.36 Stopping the instance from the AWS console

Once the instance has been stopped, you can start it again by choosing Start from the same submenu. It's also possible to completely remove the instance: for this you need to use the Terminate option.

A.6.6 Configuring AWS CLI

AWS CLI is the command line interface for AWS. For most of the things we need, using the AWS Console is sufficient, but in some cases, we need the command line tool. For example, in chapter 5, we deploy a model to Elastic Beanstalk, and we need to configure the CLI.

To use the CLI, you need to have Python. If you use Linux or macOS, you should already have a Python distribution built in. Alternatively, you can install Anaconda using the instructions in the next section.

Just having Python is not enough; you also need to install the AWS CLI itself. You can do this by running the following command in the terminal:

```
pip install awscli
```

If you already have it, it's a good idea to update it:

```
pip install -U awscli
```

After the installation finishes you need to configure the tool, specifying the access token and secret you downloaded earlier when creating a user.

One way to do this is to use the `configure` command:

```
aws configure
```

It will ask you for the keys, which we downloaded, when creating a user:

```
$ aws configure
AWS Access Key ID [None]: <ENTER_ACCESS_KEY>
AWS Secret Access Key [None]: <ENTER_SECRET_KEY>
Default region name [None]: us-east-2
Default output format [None]:
```

The region name used here is us-east-2, which is located in Ohio.

When you're finished configuring the tool, verify that it works. You can ask the CLI to return your identity, which should match the details of your user:

```
$ aws sts get-caller-identity
{
    "UserId": "AIDAVUO4TTOO55WN6WHZ4",
    "Account": "XXXXXXXXXXXX",
    "Arn": "arn:aws:iam::XXXXXXXXXXXX:user/ml-bookcamp"
}
```

appendix B
Introduction to Python

Python is currently the most popular language for building machine learning projects, and this is why we use it for doing the projects in this book.

In case you're not familiar with Python, this appendix covers the basics: the syntax and the language features we use in the book. It's not meant to be an in-depth tutorial, but it should give you enough information to start using Python immediately after finishing the appendix. Note that it's brief, and it's aimed at people who already know how to program in any other programming language.

To get the best of this appendix, create a jupyter notebook, give it a name like appendix-b-python, and use it to execute the code from the appendix. Let's start.

B.1 Variables

Python is a dynamic language — so you don't need to declare types like in Java or C++. For example, to create a variable with integer or string, we only need to do a simple assignment:

```
a = 10
b = 'string_b'
c = "string_c"
d = 0.999
```

a is an integer.

b and c are strings.

d is a float.

To print something to standard output, we can use the `print` function:

```
print(a, b, c, d)
```

It prints

```
10 string_b string_c 0.999
```

To execute the code, you can put each code snippet in a separate jupyter notebook cell and then execute it. For executing the code in the cell, you can press the Run button, or use the Shift+Enter hotkey (figure B.1).

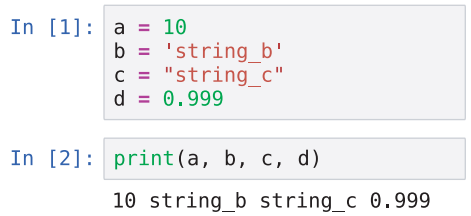


Figure B.1 Code executed in Jupyter Notebook cells. You can see the output immediately after executing the code.

When we pass multiple arguments to `print`, like in the previous example, it adds a space between the arguments when printing.

We can put multiple variables together with a special construction called *tuple*:

```
t = (a, b)
```

When we print `t`, we get the following:

```
(10, 'string_b')
```

To unwrap a tuple into multiple variables we use *tuple assignment*:

```
(c, d) = t
```

Now `c` and `d` contain the first value of the tuple, as well as the second one:

```
print(c, d)
```

It prints

```
10 string_b
```

We can drop the parentheses when using the tuple assignment:

```
c, d = t
```

This produces the same result.

Tuple assignment is quite useful and can make the code shorter. For example, we can use it to swap the content of two variables:

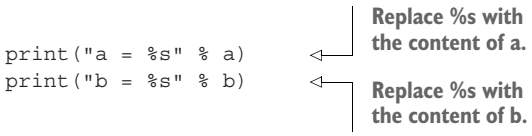
```
a = 10
b = 20
a, b = b, a
print("a =", a)
print("b =", b)
```

Replace a with b and b with a.

It will print

```
a = 20
b = 10
```

When printing, we can have nicely formatted strings using the `%` operator:



It will produce the same output:

```
a = 20
b = 10
```

Here `%s` is a placeholder: in this case, it means that we want to format the passed argument as a string. Other commonly used options are

- `%d` to format it as a number
- `%f` to format it as a floating-point number

We can pass in multiple arguments to the format operator in a tuple:

```
print("a = %s, b = %s" % (a, b))
```

The first occurrence of the placeholder `%s` will be replaced by `a`, and the second by `b`, so it will produce the following:

```
a = 20, b = 10
```

Finally, if we have a floating-point number, we can use special formatting for it:

```
n = 0.0099999999
print("n = %.2f" % n)
```

This will round the float to the second decimal point when formatting the string, so we will see 0.01 when executing the code.

There are many options for formatting strings, and also other ways of formatting. For example, there's also the so-called "new" way of formatting using the `string.format` method, which we won't cover in this appendix. You can read more about these formatting options at https://pyformat.info or in the official documentation.

B.1.1 Control flow

There are three control-flow statements in Python: `if`, `for` and `while`. Let's take a look at each of them.

CONDITIONS

A simple way to control the execution flow of a program is the `if` statement. In Python the syntax for `if` is the following:

```
a = 10

if a >= 5:
    print('the statement is true')
else:
    print('the statement is false')
```

This will print the first statement:

```
the statement is true
```

Note that in Python we use indentation for grouping the code after the `if` statement. We can chain multiple `if` statements together using `elif`, which is a shortening for `else-if`:

```
a = 3

if a >= 5:
    print('the first statement is true')
elif a >= 0:
    print('the second statement is true')
else:
    print('both statements are false')
```

This code will print the second statement:

```
the second statement is true
```

FOR LOOP

When we want to repeat the same piece of code multiple times, we use loops. The traditional `for` loop in Python looks like this:

```
for i in range(10):
    print(i)
```

This code will print numbers from 0 to 9, and 10 is not included:

```
0
1
2
3
4
5
6
7
8
9
```

When specifying the range, we can set the starting number, the end number, and the increment step:

```
for i in range(10, 100, 5):
    print(i)
```

This code will print numbers from 10 to 100 (excluded) with step 5: 10, 15, 20, …, 95.

To exit the loop earlier, we can use the `break` statement:

```
for i in range(10):
    print(i)
    if i > 5:
        break
```

This code will print numbers between 0 and 6. When `i` is 6, it will break the loop, so it will not print any numbers after 6:

```
0
1
2
3
4
5
6
```

To skip an iteration of the loop, we use the `continue` statement:

```
for i in range(10):
    if i <= 5:
        continue
    print(i)
```

This code will skip the iterations when `i` is 5 or less, so it will print only numbers starting from 6:

```
6
7
8
9
```

WHILE LOOP

The `while` loop is also available in Python. It executes while a certain condition is `True`. For example:

```
cnt = 0

while cnt <= 5:
    print(cnt)
    cnt = cnt + 1
```

In this code, we repeat the loop while the condition `cnt <= 5` is `True`. Once this condition is no longer `True`, the execution stops. This code will print numbers between 0 and 5, including 5:

```
0
1
2
3
4
5
```

We can use the `break` and `continue` statements in `while` loops as well.

B.1.2 Collections

Collections are special containers that allow keeping multiple elements in them. We will look at four types of collections: lists, tuples, sets, and dictionaries.

Lists

A *list* is an ordered collection with the possibility to access an element by index. To create a list, we can simply put elements inside squared brackets:

```
numbers = [1, 2, 3, 5, 7, 11, 13]
```

To get an element by its index, we can use the brackets notation:

```
el = numbers[1]
print(el)
```

Indexing starts at 0 in Python, so when we ask for the element at index 1, we get 2.

We can also change the values in the list:

```
numbers[1] = -2
```

To access the elements from the end, we can use negative indices. For example, `-1` will get the last element, `-2` — the one before the last and so on:

```
print(numbers[-1], numbers[-2])
```

As we expect, it prints 13 11.

To add elements to the list, use the `append` function. It will append the element to the end of the list:

```
numbers.append(17)
```

To iterate over the elements of a list, we use a `for` loop:

```
for n in numbers:
    print(n)
```

When we execute it, we see all the elements printed:

```
1
-2
3
5
7
11
13
17
```

This is also known as a `for-each` loop in other languages: we execute the body of the loop for each element of the collection. It doesn't include the indices, only the elements themselves. If we also need to have access to the index of each element, we can use `range`, as we did previously:

```
for i in range(len(numbers)):
    n = numbers[i]
    print("numbers[%d] = %d" % (i, n))
```

The function `len` returns the length of the list, so this code is roughly equivalent to the traditional way of traversing an array in C or Java and accessing each element by its index. When we execute it, the code prints the following:

```
numbers[0] = 1
numbers[1] = -2
numbers[2] = 3
numbers[3] = 5
numbers[4] = 7
numbers[5] = 11
numbers[6] = 13
numbers[7] = 17
```

A more "pythonic" (more common and idiomatic in the Python world) way of achieving the same thing is using the `enumerate` function:

```
for i, n in enumerate(numbers):
    print("numbers[%d] = %d" % (i, n))
```

In this code, the `i` variable will get the index, and the `n` variable, the respective element from the list. This code will produce the exact same output as the previous loop.

To concatenate multiple lists into one, we can use the plus operator. For example, consider two lists:

```
list1 = [1, 2, 3, 5]
list2 = [7, 11, 13, 17]
```

We can create a third list that contains all the elements from `list1` followed by the elements from `list2` by concatenating the two lists:

```
new_list = list1 + list2
```

This will produce the following list:

```
[1, 2, 3, 5, 7, 11, 13, 17]
```

Finally, it's also possible to create a list of lists: a list whose elements are lists as well. To show that, let's first create three lists with numbers:

```
list1 = [1, 2, 3, 5]
list2 = [7, 11, 13, 17]
list3 = [19, 23, 27, 29]
```

Now let's put them together in another list:

```
lists = [list1, list2, list3]
```

Now `lists` is a list of lists. When we iterate over it with a `for` loop, at each iteration we get a list:

```
for l in lists:
    print(l)
```

This will produce the following output:

```
[1, 2, 3, 5]
[7, 11, 13, 17]
[19, 23, 27, 29]
```

SLICING

Another useful concept in Python is *slicing* — it's used for getting a part of the list. For example, let's consider the list of numbers again:

```
numbers = [1, 2, 3, 5, 7]
```

If we want to select a sublist with the first three elements, we can use the colon operator (`:`) for specifying the range for selection:

```
top3 = numbers[0:3]
```

In this case, `0:3` means "select elements starting from index 0 till index 3 (exclusive)." The result contains the first three elements: `[1, 2, 3]`. Note that it selects elements at the indices 0, 1, and 2, so 3 is not included.

If we want to include the beginning of the list, we don't need to specify the first number in the range:

```
top3 = numbers[:3]
```

If we don't specify the second number in the range, we get everything until the end of the list:

```
last3 = numbers[2:]
```

The list `last3` will contain the last three elements: `[3, 5, 7]` (figure B.2).

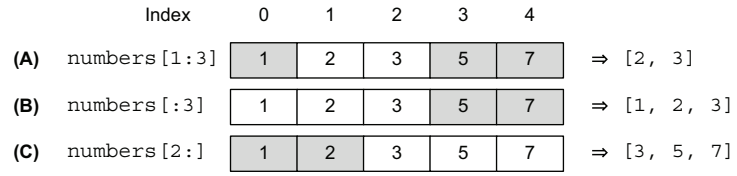


Figure B.2 Using the colon operator to select a sublist of a list

TUPLES

We already met tuples previously in the Variables section. Tuples are also collections; they are quite similar to lists. The only difference is that they are immutable: once you create a tuple, you cannot change the content of the tuple.

To create a tuple we use parentheses:

```
numbers = (1, 2, 3, 5, 7, 11, 13)
```

Like with lists, we can get the value by index:

```
el = numbers[1]
print(el)
```

However, we cannot update the values in the tuple. When we try to do it, we get an error:

```
numbers[1] = -2
```

If we try to execute this code, we get

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-9166360b9018> in <module>
----> 1 numbers[1] = -2

TypeError: 'tuple' object does not support item assignment
```

Likewise, we cannot append a new element to the tuple. However, we can use concatenation to achieve the same result:

```
numbers = numbers + (17,)
```

Here we create a new tuple that contains the old numbers, and we concatenate it with another tuple that contains only one number: 17. Note that we need to add a comma to make a tuple; otherwise Python will treat it as a simple number.

Effectively, the expression on the previous page is the same as writing

```
numbers = (1, 2, 3, 5, 7, 11, 13) + (17,)
```

After doing this, we get a new tuple that contains a new element, so when printing it, we get

```
(1, 2, 3, 5, 7, 11, 13, 17)
```

SET

Another useful collection is a *set*: it's an unordered collection that keeps only unique elements. Unlike lists, it cannot contain duplicates, and it's also not possible to access an individual element of a set by index.

To create a set, we use curly braces:

```
numbers = {1, 2, 3, 5, 7, 11, 13}
```

NOTE To create an empty set, we need to use `set`:

```
empty_set = set()
```

Simply putting empty curly braces will create a dictionary — a collection that we cover later in this appendix:

```
empty_dict = {}
```

Sets are faster than lists for checking if the collection contains an element. We use the `in` operator for checking it:

```
print(1 in numbers)
```

Since "1" is in the `numbers` set, this line of code will print `True`.

To add an element to the set, we use the `add` method:

```
numbers.add(17)
```

To iterate over all the elements of the set we again use a `for` loop:

```
for n in numbers:
    print(n)
```

When we execute it, it prints

```
1
2
3
5
7
11
13
17
```

DICTIONARIES

Dictionary is another extremely useful collection in Python: we use it to build a key-value map. To create a dictionary, we use curly braces, and to separate the keys and values we use colons (`:`):

```
words_to_numbers = {
    'one': 1,
    'two': 2,
    'three': 3,
}
```

To retrieve the value by the key, we use brackets:

```
print(words_to_numbers['one'])
```

If something is not in the dictionary, Python raises an exception:

```
print(words_to_numbers['five'])
```

When we try to execute it, we get the following error:

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-38-66a309b8feb5> in <module>
----> 1 print(words_to_numbers['five'])

KeyError: 'five'
```

To avoid it, we can first check if the key is in the dictionary before attempting to get the value. We can use the `in` statement for checking it:

```
if 'five' in words_to_numbers:
    print(words_to_numbers['five'])
else:
    print('not in the dictionary')
```

When running this code, we'll see `not in the dictionary` in the output.

Another option is to use the `get` method. It doesn't raise an exception, but returns `None` if the key is not present in the dictionary:

```
value = words_to_numbers.get('five')
print(value)
```

It will print `None`. When using `get`, we can specify the default value in case the key is not present in the dictionary:

```
value = words_to_numbers.get('five', -1)
print(value)
```

In this situation, we'll get `-1`.

To iterate over all the keys of a dictionary, we use a `for` loop over the results from the `keys` method:

```
for k in words_to_numbers.keys():
    v = words_to_numbers[k]
    print("%s: %d" % (k, v))
```

It will print

```
one: 1
two: 2
three: 3
```

Alternatively, we can directly iterate over the key-value pairs in the dictionary using the `items` method:

```
for k, v in words_to_numbers.items():
    print("%s: %d" % (k, v))
```

It produces exactly the same output as the previous code.

LIST COMPREHENSION

List comprehension is a special syntax for creating and filtering lists in Python. Let's again consider a list with numbers:

```
numbers = [1, 2, 3, 5, 7]
```

Suppose we want to create another list where all the elements of the original list are squared. For that we can use a `for` loop:

```
squared = []

for n in numbers:
    s = n * n
    squared.append(s)
```

We can concisely rewrite this code into one single line using list comprehension:

```
squared = [n * n for n in numbers]
```

It's also possible to add an `if` condition inside to process only the elements that meet the condition:

```
squared = [n * n for n in numbers if n > 3]
```

It translates to the following code:

```
squared = []

for n in numbers:
    if n > 3:
        s = n * n
        squared.append(s)
```

If all we need is to apply the filter and leave the elements as is, we can do that as well:

```
filtered = [n for n in numbers if n > 3]
```

This translates to

```
filtered = []

for n in numbers:
    if n > 3:
        filtered.append(n)
```

It's also possible to use list comprehension for creating other collections with a slightly different syntax. For example, for dictionaries we put curly braces around the expression and use a colon to separate keys with values:

```
result = {k: v * 10 for (k, v) in words_to_numbers.items() if v % 2 == 0}
```

This is a shortcut for the following code:

```
result = {}

for (k, v) in words_to_numbers.items():
    if v % 2 == 0:
        result[k] = v * 10
```

WARNING When learning about list comprehension, it might be tempting to start using it everywhere. Typically it fits best for simple cases, but for more complex situations, `for` loops should be preferred over list comprehension for better code readability. If in doubt, use `for` loops.

B.1.3 Code reusability

At some point, when we write a lot of code, we need to think about how to organize it better. We can achieve that by putting small reusable pieces of code inside functions or classes. Let's take a look at how to do it.

FUNCTIONS

To create a function, we use the `def` keyword:

```
def function_name(arg1, arg2):
    # body of the function
    return 0
```

When we want to exit the function and return some value, we use the `return` statement. If we simply put `return` without any value or don't include `return` in the body of the function, the function will return `None`.

For example, we can write a function that prints values from 0 up to a specified number:

```
def print_numbers(max):
    for i in range(max + 1):
        print(i)
```

Create a function with one argument: max.

Use the max argument inside the function.

To call this function, simply add the arguments in parentheses after the name:

```
print_numbers(10)
```

It's also possible to provide the names of the arguments when invoking the function:

```
print_numbers(max=10)
```

CLASSES

Classes provide higher-level abstraction than functions: they can have an internal state and methods that operate on this state. Let's consider a class, `NumberPrinter`, that does the same thing as the function from the previous section — it prints numbers.

```
class NumberPrinter:

    def __init__(self, max):
        self.max = max

    def print_numbers(self):
        for i in range(self.max + 1):
            print(i)
```

The class initializer

Assign the max argument to the max field.

Method of the class

Use the internal state when invoking the method.

In this code, `__init__` is the initializer. It runs whenever we want to create an instance of a class:

```
num_printer = NumberPrinter(max=10)
```

Note that inside the class, the `__init__` method has two arguments: `self` and `max`. The first argument of all the methods always has to be `self`: this way we can use `self` inside the method to access the state of the object.

However, when we invoke the method later, we don't pass anything to the `self` argument: it's hidden from us. So, when we invoke the `print_number` method on the instance of the `NumberPrinter` object, we simply put empty parentheses with no parameters:

```
num_printer.print_numbers()
```

This code produces the same output as the function from the previous section.

IMPORTING CODE

Now suppose we want to put some code to a separate file. Let's create a file called useful_code.py and place it in the same folder as the notebook.

Open this file with an editor. Inside the file, we can put the function and the class we just created. In this way, we create a module with the name `useful_code`. To access the function and the class inside the module, we import them using the `import` statement:

```
import useful_code
```

Once it's imported, we can use it:

```
num_printer = useful_code.NumberPrinter(max=10)
num_printer.print_numbers()
```

It's also possible to import a module and give it a short name. For example, if instead of writing `useful_code` we want to write `uc`, we can do the following:

```
import useful_code as uc

num_printer = uc.NumberPrinter(max=10)
num_printer.print_numbers()
```

This is a very common idiom in scientific Python. Packages like NumPy and Pandas are typically imported with shorter aliases:

```
import numpy as np
import pandas as pd
```

Finally, if we don't want to import everything from the module, we can choose what exactly to import using `from ... import` syntax:

```
from useful_code import NumberPrinter

num_printer = NumberPrinter(max=10)
num_printer.print_numbers()
```

B.1.4 Installing libraries

It's possible to put our code into packages that are available to everyone. For example, NumPy or Pandas are such packages. They are already available in the Anaconda distribution, but typically they don't come pre-installed with Python.

To install such external packages, we can use the built-in package installer called pip. To use it, open your terminal and execute the `pip install` command there:

```
pip install numpy scipy pandas
```

After the install command, we list the packages we want to install. It's also possible to specify the version of each package when installing:

```
pip install numpy==1.16.5 scipy==1.3.1 pandas==0.25.1
```

When we already have a package, but it's outdated and we want to update it, we need to run `pip install` with the `-U` flag:

```
pip install -U numpy
```

Finally, if we want to remove a package, we use `pip uninstall`:

```
pip uninstall numpy
```

B.1.5 Python programs

To execute Python code, we can simply call the Python interpreter and specify the file we want to execute. For example, to run the code inside our useful_code.py script, execute the following command in the command line:

```
python useful_code.py
```

When we execute it, nothing happens: we only declare a function and a class there and don't actually use them. To see some results, we need to add a few lines of code to the file. For example, we can add the following:

```
num_printer = NumberPrinter(max=10)
num_printer.print_numbers()
```

Now when we execute this file, we see the numbers that `NumberPrinter` prints.

However, when we import a module, internally Python executes everything inside the module. It means that the next time we do `import useful_code` in the notebook, we'll see the numbers printed there.

To avoid it, we can tell the Python interpreter that some code needs to run only when executed as a script, and not imported. To achieve that, we put our code inside the following construction:

```
if __name__ == "__main__":
    num_printer = NumberPrinter(max=10)
    num_printer.print_numbers()
```

Finally, we can also pass arguments when running python scripts:

```
import sys

# declarations of print_numbers and NumberPrinter

if __name__ == "__main__":
    max_number = int(sys.argv[1])
    num_printer = NumberPrinter(max=max_number)
    num_printer.print_numbers()
```

Parse the parameter as an integer: by default, it's a string.

Pass the parsed argument to the NumberPrinter instance.

Now we can run the script with custom parameters:

```
python useful_code.py 5
```

As a result, we'll see numbers from `0` to `5`:

```
0
1
2
3
4
5
```

appendix C
Introduction to NumPy

We don't expect any NumPy knowledge from the readers and try to put all the required information in the chapters as we go along. However, because the purpose of the book is to teach machine learning rather than NumPy, we couldn't cover everything in great detail in the chapters. That's the focus of this appendix: to give an overview of the most important concepts from NumPy in one centralized place.

In addition to introducing NumPy, the appendix also covers a bit of linear algebra useful for machine learning, including matrix and vector multiplication, matrix inverse, and the normal equation.

NumPy is a Python library, so if you're not yet familiar with Python, check out appendix B.

C.1 NumPy

NumPy is short for *Numerical Python* — it's a Python library for numerical manipulations. NumPy plays a central role in the Python machine learning ecosystem: nearly all the libraries in Python depend on it. For example, Pandas, Scikit-learn, and TensorFlow all rely on NumPy for numerical operations.

NumPy comes preinstalled in the Anaconda distribution of NumPy, so if you use it, you don't need to do anything extra. But if you don't use Anaconda, installing NumPy is quite simple with `pip`:

```
pip install numpy
```

To experiment with NumPy, let's create a new Jupyter Notebook and name it appendix-c-numpy.

To use NumPy, we need to import it. That's why in the first cell we write

```
import numpy as np
```

In the scientific Python community, it's common to use an alias when importing NumPy. That's why we add `as np` in the installation code. This allows us to write `np` in the code instead of `numpy`.

We'll start exploring NumPy from its core data structure: the NumPy array.

C.1.1 NumPy arrays

NumPy arrays are similar to Python lists, but they are better optimized for number-crunching tasks like machine learning.

To create an array of a predefined size filled with zeros, we use the `np.zeros` function:

```
zeros = np.zeros(10)
```

This creates an array with 10 zero elements (figure C.1).

```
zeros = np.zeros(10)
zeros
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Figure C.1 Creating a NumPy array of length 10 filled with zeros

Likewise, we can create an array with ones using the `np.ones` function:

```
ones = np.ones(10)
```

It works exactly the same as zeros, except the elements are ones.

Both functions are a shortcut for a more general function: `np.full`. It creates an array of a certain size filled with the specified element. For example, to create an array of size 10 filled with zeros, we do the following:

```
array = np.full(10, 0.0)
```

We can achieve the same result using the `np.repeat` function:

```
array = np.repeat(0.0, 10)
```

This code produces the same result as the earlier code (figure C.2).

```
array = np.full(10, 0.0)
array
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
array = np.repeat(0.0, 10)
array
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Figure C.2 To create an array filled with a particular number, use `np.full` or `np.repeat`.

Although in this example both functions produce the same code, `np.repeat` is actually more powerful. For example, we can use it to create an array where multiple elements are repeated one after another:

```
array = np.repeat([0.0, 1.0], 5)
```

It creates an array of size 10 where the number 0 is repeated five times, and then the number 1 is repeated five times (figure C.3):

```
array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1.])
```

```
array = np.repeat([0.0, 1.0], 5)
array
```

```
array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1.])
```

```
array = np.repeat([0.0, 1.0], [2, 3])
array
```

```
array([0., 0., 1., 1., 1.])
```

Figure C.3 The `np.repeat` function is more flexible than `np.full`: it can create arrays by repeating multiple elements.

We can be even more flexible and specify how many times each element should be repeated:

```
array = np.repeat([0.0, 1.0], [2, 3])
```

In this case, 0.0 is repeated two times and 1.0 is repeated three times:

```
array([0., 0., 1., 1., 1.])
```

Like with lists, we can access an element of an array with square brackets:

```
el = array[1]
print(el)
```

This code prints 0.0.

Unlike the usual Python lists, we can access multiple elements of the array at the same time by using a list with indices in the square brackets:

```
print(array[[4, 2, 0]])
```

The result is another array of size 3 consisting of elements of the original array indexed by 4, 2, and 0, respectively:

```
[1., 1., 0.]
```

We can also update the elements of the array using square brackets:

```
array[1] = 1
print(array)
```

Because we changed the element at index 1 from 0 to 1, it prints the following:

```
[0. 1. 1. 1. 1.]
```

If we already have a list with numbers, we can convert it to a NumPy array using `np.array`:

```
elements = [1, 2, 3, 4]
array = np.array(elements)
```

Now `array` is a NumPy array of size 4 with the same elements as the original list:

```
array([1, 2, 3, 4])
```

Another useful function for creating NumPy arrays is `np.arange`. It's the NumPy equivalent of Python's `range`:

```
np.arange(10)
```

It creates an array of length 10 with numbers from 0 to 9, and like in standard Python's `range`, 10 is not included in the array:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Often we need to create an array of a certain size filled with numbers between some number *x* and some number *y*. For example, imagine that we need to create an array with numbers from 0 to 1:

$$0.0, 0.1, 0.2, \ldots, 0.9, 1.0$$

We can use `np.linspace`:

```
thresholds = np.linspace(0, 1, 11)
```

This function takes three parameters:

- The starting number: in our case, we want to start from 0.
- The last number: we want to finish with 1.
- The length of the resulting array: in our case, we want 11 numbers in the array.

This code produces 11 numbers from 0 until 1 (figure C.4).

Python lists can usually contain elements of any type. This is not the case for NumPy arrays: all elements of an array must have the same type. These types are called *dtypes*.

```
thresholds = np.linspace(0, 1, 11)
thresholds
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Figure C.4 The function `linspace` from NumPy produces a sequence of specified length (11) that starts at 0 and ends at 1.

There are four broad categories of dtypes:

- Unsigned integers (uint): integers that are always positive (or zero)
- Signed integers (int): integers that can be positive and negative
- Floats (float): real numbers
- Booleans (bool): only True and False values

Multiple variations of each dtype exist, depending on the number of bits used for representing the value in memory.

For uint we have four types: `uint8`, `uint16`, `uint32`, and `uint64` of size 8, 16, 32, and 64 bits, respectively. Likewise, we have four types of int: `int8`, `int16`, `int32`, and `int64`. The more bits we use, the larger numbers we can store (table C.1).

Table C.1 Three common NumPy dtypes: uint, int, and float. Each dtype has multiple size variations ranging from 8 to 64 bits.

Size (bits)	uint	int	float
8	$0 .. 2^8 - 1$	$-2^7 .. 2^7 - 1$	–
16	$0 .. 2^{16} - 1$	$-2^{15} .. 2^{15} - 1$	Half precision
32	$0 .. 2^{32} - 1$	$-2^{31} .. 2^{31} - 1$	Single precision
64	$0 .. 2^{64} - 1$	$-2^{63} .. 2^{63} - 1$	Double precision

In the case of floats, we have three types: `float16`, `float32`, and `float64`. The more bits we use, the more precise the float is.

You can check the full list of different dtypes in the official documentation (https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html).

> **NOTE** In NumPy, the default float dtype is `float64`, which uses 64 bits (8 bytes) for each number. For most machine learning applications, we don't need such precision and can reduce the memory footprint two times by using `float32` instead of `float64`.

When creating an array, we can specify the dtype. For example, when using `np.zeros` and `np.ones`, the default dtype is `float64`. We can specify the dtype when creating an array (figure C.5):

```
zeros = np.zeros(10, dtype=np.uint8)
```

```
zeros = np.zeros(10, dtype=np.uint8)
zeros
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=uint8)
```

Figure C.5 We can specify the dtype when creating an array.

When we have an array with integers and assign a number outside of the range, the number is cut: only the least significant bits are kept.

For example, suppose we use the `uint8` array `zeros` we just created. Because the dtype is `uint8`, the largest number it can store is 255. Let's try to assign 300 to the first element of the array:

```
zeros[0] = 300
print(zeros[0])
```

Because 300 is greater than 255, only the least significant bits are kept, so this code prints 44.

> **WARNING** Be careful when choosing the dtype for an array. If you accidentally choose a dtype that's too narrow, NumPy won't warn you when you put in a big number. It will simply truncate them.

Iterating over all elements of an array is similar to list. We simply can use a `for` loop:

```
for i in np.arange(5):
    print(i)
```

This code prints numbers from 0 to 4:

```
0
1
2
3
4
```

C.1.2 Two-dimensional NumPy arrays

So far we have covered one-dimensional NumPy arrays. We can think of these arrays as vectors. However, for machine learning applications, having only vectors is not enough: we also often need matrices.

In plain Python, we'd use a list of lists for that. In NumPy, the equivalent is a two-dimensional array.

To create a two-dimensional array with zeros, we simply use a tuple instead of a number when invoking `np.zeros`:

```
zeros = np.zeros((5, 2), dtype=np.float32)
```

We use a tuple (5, 2), so it creates an array of zeros with five rows and two columns (figure C.6).

```
zeros = np.zeros((5, 2), dtype=np.float32)
zeros
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)
```

Figure C.6 To create a two-dimensional array, use a tuple with two elements. The first element specifies the number of rows, and the second, the number of columns.

In the same way, we can use `np.ones` or `np.fill` — instead of a single number, we put in a tuple.

The dimensionality of an array is called *shape*. This is the first parameter we pass to the `np.zeros` function: it specifies how many rows and columns the array will have. To get the shape of an array, use the `shape` property:

```
print(zeros.shape)
```

When we execute it, we see (5, 2).

It's possible to convert a list of lists to a NumPy array. As with usual lists of numbers, simply use `np.array` for that:

```
numbers = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

numbers = np.array(numbers)
```

Creates a list of lists

Converts the list to a two-dimensional array

After executing this code, `numbers` becomes a NumPy array with shape (3, 3). When we print it, we get

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

To access an element of a two-dimensional array, we need to use two numbers inside the brackets:

```
print(numbers[0, 1])
```

This code will access the row indexed by 0 and column indexed by 1. So it will print 2.

As with one-dimensional arrays, we use the assignment operator (=) to change an individual value of a two-dimensional array:

```
numbers[0, 1] = 10
```

When we execute it, the content of the array changes:

```
array([[ 1, 10,  3],
       [ 4,  5,  6],
       [ 7,  8,  9]])
```

If instead of two numbers we put only one, we get the entire row, which is a one-dimensional NumPy array:

```
numbers[0]
```

This code returns the entire row indexed by 0:

```
array([1 2 3])
```

To access a column of a two-dimensional array, we use a colon (:) instead of the first element. Like with rows, the result is also a one-dimensional NumPy array:

```
numbers[:, 1]
```

When we execute it, we see the entire column:

```
array([2 5 8])
```

It's also possible to overwrite the content of the entire row or a column using the assignment operator. For example, suppose we want to replace a row in the matrix:

```
numbers[1] = [1, 1, 1]
```

This results in the following change:

```
array([[ 1, 10,  3],
       [ 1,  1,  1],
       [ 7,  8,  9]])
```

Likewise, we can replace the content of an entire column:

```
numbers[:, 2] = [9, 9, 9]
```

As a result, the last column changes:

```
array([[ 1, 10,  9],
       [ 1,  1,  9],
       [ 7,  8,  9]])
```

C.1.3 Randomly generated arrays

Often it's useful to generate arrays filled with random numbers. To do this in NumPy, we use the `np.random` module.

For example, to generate a 5 × 2 array of random numbers uniformly distributed between 0 and 1, use `np.random.rand`:

```
arr = np.random.rand(5, 2)
```

When we run it, it generates an array that looks like this:

```
array([[0.64814431, 0.51283823],
       [0.40306102, 0.59236807],
       [0.94772704, 0.05777113],
       [0.32034757, 0.15150334],
       [0.10377917, 0.68786012]])
```

Every time we run the code, it generates a different result. Sometimes we need the results to be reproducible, which means that if we want to execute this code later, we will get the same results. To achieve that, we can set the seed of the random-number generator. Once the seed is set, the random-number generator produces the same sequence every time we run the code:

```
np.random.seed(2)
arr = np.random.rand(5, 2)
```

On Ubuntu Linux version 18.04 with NumPy version 1.17.2, it generates the following array:

```
array([[0.4359949 , 0.02592623],
       [0.54966248, 0.43532239],
       [0.4203678 , 0.33033482],
       [0.20464863, 0.61927097],
       [0.29965467, 0.26682728]])
```

No matter how many times we re-execute this cell, the results are the same.

> **WARNING** Fixing the seed of the random-number generator guarantees that the generator produces the same results when executed on the same OS with the same NumPy version. However, there's no guarantee that updating the NumPy version will not affect reproducibility: a change of version may result in changes in the random-number generator algorithm, and that may lead to different results across versions.

If instead of uniform distribution, we want to sample from the standard normal distribution, we use `np.random.randn`:

```
arr = np.random.randn(5, 2)
```

NOTE Every time we generate a random array in this appendix, we make sure we fix the seed number before generating it, even if we don't explicitly specify it in the code — we do it to ensure consistency. We use 2 as the seed. There's no particular reason for this number.

To generate uniformly distributed random integers between 0 and 100 (exclusive), we can use `np.random.randint`:

```
randint = np.random.randint(low=0, high=100, size=(5, 2))
```

When executing the code, we get a 5 × 2 NumPy array of integers:

```
array([[40, 15],
       [72, 22],
       [43, 82],
       [75,  7],
       [34, 49]])
```

Another quite useful feature is shuffling an array — rearranging the elements of an array in random order. For example, let's create an array with a range and then shuffle it:

```
idx = np.arange(5)
print('before shuffle', idx)

np.random.shuffle(idx)
print('after shuffle', idx)
```

When we run the code, we see the following:

```
before shuffle [0 1 2 3 4]
after shuffle  [2 3 0 4 1]
```

C.2 NumPy operations

NumPy comes with a wide range of operations that work with the NumPy arrays. In this section, we cover operations that we'll need throughout the book.

C.2.1 Element-wise operations

NumPy arrays support all the arithmetic operations: addition (+), subtraction (–), multiplication (*), division (/) and others.

To illustrate these operations, let's first create an array using `arange`:

```
rng = np.arange(5)
```

This array contains five elements from 0 to 4:

```
array([0, 1, 2, 3, 4])
```

To multiply every element of the array by two, we simply use the multiplication operator (*):

```
rng * 2
```

As a result, we get a new array where each element from the original array is multiplied by two:

```
array([0, 2, 4, 6, 8])
```

Note that we don't need to explicitly write any loops here to apply the multiplication operation individually to each element: NumPy does it for us. We can say that the multiplication operation is applied *element-wise* — to all elements at once. The addition (+), subtraction (–) and division (/) operations are also element-wise and require no explicit loops.

Such element-wise operations are often called *vectorized*: the `for` loop happens internally in native code (written C and Fortran), so the operations are very fast!

NOTE Whenever possible, use vectorized operations from NumPy instead of loops: they are always a magnitude faster.

In the previous code, we used only one operation. It's possible to apply multiple operations at once in one expression:

```
(rng - 1) * 3 / 2 + 1
```

This code creates a new array with the result:

```
array([-0.5, 1. , 2.5, 4. , 5.5])
```

Note that the original array contains integers, but because we used the division operation, the result is an array with float numbers.

Previously, our code involved an array and simple Python numbers. It's also possible to do element-wise operations with two arrays if they have the same shape.

For example, suppose we have two arrays, one containing numbers from 0 to 4, and another containing some random noise:

```
noise = 0.01 * np.random.rand(5)
numbers = np.arange(5)
```

We sometimes need to do that for modeling not-ideal real-life data: in reality there are always imperfections when data is collected, and we can model these imperfections by adding noise.

We build the `noise` array by first generating numbers between 0 and 1 and then multiplying them by 0.01. This effectively generates random numbers between 0 and 0.01:

```
array([0.00435995, 0.00025926, 0.00549662, 0.00435322, 0.00420368])
```

We can then add these two arrays and get a third one with the sum:

```
result = numbers + noise
```

In this array, each element of the result is the sum of the respective elements of the two other arrays:

```
array([0.00435995, 1.00025926, 2.00549662, 3.00435322, 4.00420368])
```

We can round the numbers to any precision using the `round` method:

```
result.round(4)
```

It's also an element-wise operation, so it's applied to all the elements at once and the numbers are rounded to the fourth digit:

```
array([0.0044, 1.0003, 2.0055, 3.0044, 4.0042])
```

Sometimes we need to square all the elements of an array. For that, we can simply multiply the array by itself. Let's first generate an array:

```
pred = np.random.rand(3).round(2)
```

This array contains three random numbers:

```
array([0.44, 0.03, 0.55])
```

Now we can multiply it by itself:

```
square = pred * pred
```

As a result, we get a new array where each element of the original array is squared:

```
array([0.1936, 0.0009, 0.3025])
```

Alternatively, we can use the power operator (**):

```
square = pred ** 2
```

Both approaches lead to the same results (figure C.7).

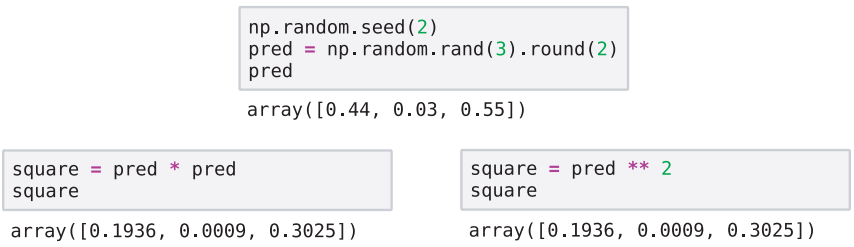


Figure C.7 There are two ways to square the elements of an array: multiply the array with itself or use the power operation (**).

Other useful element-wise operations that we might need for machine learning applications are exponent, logarithm, and square root:

```
pred_exp = np.exp(pred)
pred_log = np.log(pred)
pred_sqrt = np.sqrt(pred)
```

Computes the exponent

Computes the logarithm

Computes the square root

Boolean operations can also be applied to NumPy arrays element-wise. To illustrate them, let's again generate an array with some random numbers:

```
pred = np.random.rand(3).round(2)
```

This array contains the following numbers:

```
array([0.44, 0.03, 0.55])
```

We can see the elements that are greater than 0.5:

```
result = pred >= 0.5
```

As a result, we get an array with three Boolean values:

```
array([False, False, True])
```

We know that only the last element of the original array is greater than 0.5, so it's `True` and the rest are `False`.

As with arithmetic operations, we can apply Boolean operations on two NumPy arrays of the same shape. Let's generate two random arrays:

```
pred1 = np.random.rand(3).round(2)
pred2 = np.random.rand(3).round(2)
```

The arrays have the following values:

```
array([0.44, 0.03, 0.55])
array([0.44, 0.42, 0.33])
```

Now we can use the greater-than-or-equal-to operator (>=) to compare the values of these arrays:

```
pred1 >= pred2
```

As a result, we get an array with Booleans (figure C.8):

```
array([ True, False, True])
```

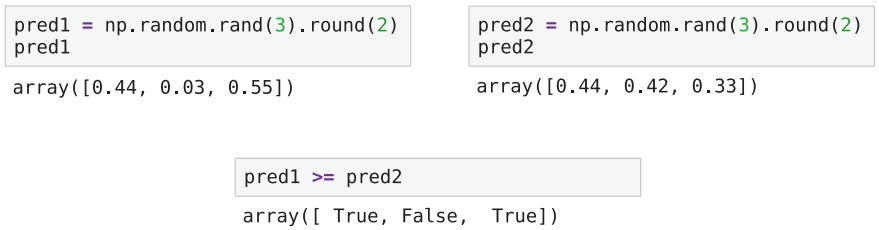


Figure C.8 Boolean operations in NumPy are element-wise and can be applied to two arrays of the same shape for comparing values.

Finally, we can apply logical operations — like logical and (&) and or (|) — to Boolean NumPy arrays. Let's again generate two random arrays:

```
pred1 = np.random.rand(5) >= 0.3
pred2 = np.random.rand(5) >= 0.4
```

The generated arrays have the following values:

```
array([ True, False, True])
array([ True, True, False])
```

Like arithmetical operations, logical operators are also element-wise. For example, to compute the element-wise and, we simply use the & operator with arrays (figure C.9):

```
res_and = pred1 & pred2
```

As a result, we get

```
array([ True, False, False])
```

The logical or works in the same way (figure C.9):

```
res_or = pred1 | pred2
```

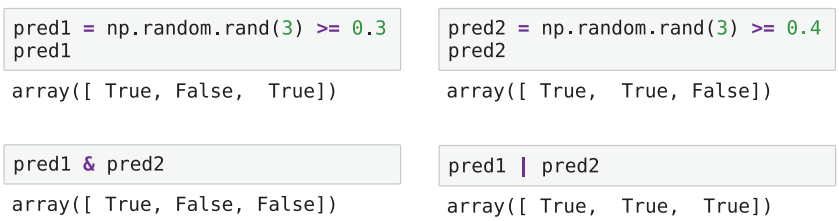


Figure C.9 Logic operations like logical and logical or can also be applied element-wise.

This creates the following array:

```
array([ True, True, True])
```

C.2.2 Summarizing operations

Whereas element-wise operations take in an array and produce an array of the same shape, the summarizing operations take in an array and produce a single number.

For example, we can generate an array and then calculate the sum of all elements:

```
pred = np.random.rand(3).round(2)
pred_sum = pred.sum()
```

In this example, `pred` is

```
array([0.44, 0.03, 0.55])
```

Then `pred_sum` is the sum of all three elements, which is 1.02:

$$0.44 + 0.03 + 0.55 = 1.02$$

Other summarizing operations include `min`, `mean`, `max` and `std`:

```
print('min = %.2f' % pred.min())
print('mean = %.2f' % pred.mean())
print('max = %.2f' % pred.max())
print('std = %.2f' % pred.std())
```

After running this code, it produces

```
min = 0.03
mean = 0.34
max = 0.55
std = 0.22
```

When we have a two-dimensional array, summarizing operations also produce a single number. However, it's also possible to apply these operations to rows or columns separately.

For example, let's generate a 4×3 array:

```
matrix = np.random.rand(4, 3).round(2)
```

This generates an array:

```
array([[0.44, 0.03, 0.55],
       [0.44, 0.42, 0.33],
       [0.2 , 0.62, 0.3 ],
       [0.27, 0.62, 0.53]])
```

When we invoke the `max` method, it returns a single number:

```
matrix.max()
```

The result is 0.62, which is the maximum number across all elements of the matrix.

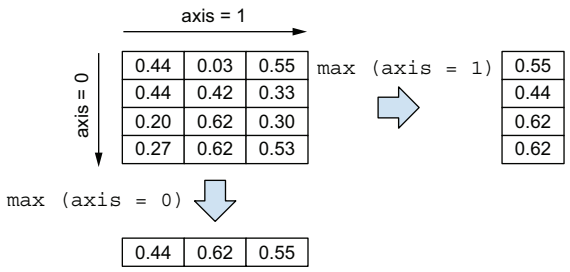


Figure C.10 We can specify the axis along which we apply the operation: `axis=1` means applying it to rows, and `axis=0` means applying it to columns.

If we now want to find the largest number in each row, we can use the `max` method specifying the axis along which we apply this operation. When we want to do it for rows, we use `axis=1` (figure C.10):

```
matrix.max(axis=1)
```

As a result, we get an array with four numbers — the largest number in each row:

```
array([0.55, 0.44, 0.62, 0.62])
```

Likewise, we can find the largest number in each column. For that, we use `axis=0`:

```
matrix.max(axis=0)
```

This time the result is three numbers — the largest numbers in each column:

```
array([0.44, 0.62, 0.55])
```

Other operations — `sum`, `min`, `mean`, `std`, and many others — can also can take `axis` as an argument. For example, we can easily calculate the sum of elements of every row:

```
matrix.sum(axis=1)
```

When executing it, we get four numbers:

```
array([1.02, 1.19, 1.12, 1.42])
```

C.2.3 Sorting

Often we need to sort elements of an array. Let's see how to do it in NumPy. First, let's generate a one-dimensional array with four elements:

```
pred = np.random.rand(4).round(2)
```

The array we generate contains the following elements:

```
array([0.44, 0.03, 0.55, 0.44])
```

To create a sorted copy of the array, use `np.sort`:

```
np.sort(pred)
```

It returns an array with all the elements sorted:

```
array([0.03, 0.44, 0.44, 0.55])
```

Because it creates a copy and sorts it, the original array `pred` remains unchanged.

If we want to sort the elements of the array in place without creating another array, we invoke the method `sort` on the array itself:

```
pred.sort()
```

Now the array `pred` is sorted.

When it comes to sorting, we have another useful thing: `argsort`. Instead of sorting an array, it returns the indices of the array in the sorted order (figure C.11):

```
idx = pred.argsort()
```

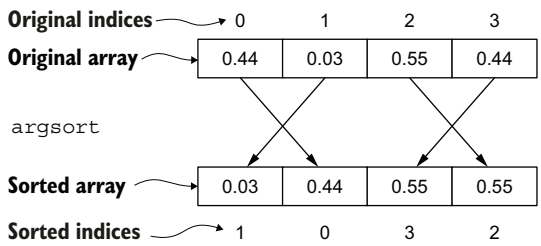


Figure C.11 The function `sort` sorts the array, whereas `argsort` produces an array of indices that sort the array.

Now the array `idx` contains indices in the sorted order:

```
array([1, 0, 3, 2])
```

Now we can use the array `idx` with indexes to get the original array in the sorted order:

```
pred[idx]
```

As we see, it's indeed sorted:

```
array([0.03, 0.44, 0.44, 0.55])
```

C.2.4 Reshaping and combining

Each NumPy array has a shape, which specifies its size. For a one-dimensional array, it's the length of the array, and for a two-dimensional array, it's the number of rows and columns. We already know that we can access the shape of an array by using the `shape` property:

```
rng = np.arange(12)
rng.shape
```

The shape of `rng` is (12), which means that it's a one-dimensional array of length 12. Because we used `np.arange` to create the array, it contains the numbers from 0 to 11 (inclusive):

```
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

It's possible to change the shape of an array from one-dimensional to two-dimensional. We use the `reshape` method for that:

```
rng.reshape(4, 3)
```

As a result, we get a matrix with four rows and three columns:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

The reshaping worked because it was possible to rearrange 12 original elements into four rows with three columns. In other words, the total number of elements didn't change. However, if we attempt to reshape it to (4, 4), it won't let us:

```
rng.reshape(4, 4)
```

When we do it, NumPy raises a `ValueError`:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-176-880fb98fa9c8> in <module>
----> 1 rng.reshape(4, 4)

ValueError: cannot reshape array of size 12 into shape (4,4)
```

Sometimes we need to create a new NumPy array by putting multiple arrays together. Let's see how to do it.

First, we create two arrays, which we'll use for illustration:

```
vec = np.arange(3)
mat = np.arange(6).reshape(3, 2)
```

The first one, `vec`, is a one-dimensional array with three elements:

```
array([0, 1, 2])
```

The second one, `mat`, is a two-dimensional one with three rows and two columns:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

The simplest way to combine two NumPy arrays is using the `np.concatenate` function:

```
np.concatenate([vec, vec])
```

It takes in a list of one-dimensional arrays and combines them into one larger one-dimensional array. In our case, we pass `vec` two times, so as a result, we have an array of length six:

```
array([0, 1, 2, 0, 1, 2])
```

We can achieve the same result using `np.hstack`, which is short for horizontal stack:

```
np.hstack([vec, vec])
```

It again takes a list of arrays and stacks them horizontally, producing a larger array:

```
array([0, 1, 2, 0, 1, 2])
```

We can also apply `np.hstack` to two-dimensional arrays:

```
np.hstack([mat, mat])
```

The result is another matrix where the original matrices are stacked horizontally by columns:

```
array([[0, 1, 0, 1],
       [2, 3, 2, 3],
       [4, 5, 4, 5]])
```

However, in case of two-dimensional arrays, `np.concatenate` works differently from `np.hstack`:

```
np.concatenate([mat, mat])
```

When we apply `np.concatenate` to matrices, it stacks them vertically, not horizontally, like one-dimensional arrays, creating a new matrix with six rows:

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 1],
```

```
       [2, 3],
       [4, 5]])
```

Another useful method for combining NumPy arrays is `np.column_stack`: it allows us to stack vectors and matrices together. For example, suppose we want to add an extra column to our matrix. For that we simply pass a list that contains the vector and then the matrix:

```
np.column_stack([vec, mat])
```

As a result, we have a new matrix, where `vec` becomes the first column, and the rest of the `mat` goes after it:

```
array([[0, 0, 1],
       [1, 2, 3],
       [2, 4, 5]])
```

We can apply `np.column_stack` to two vectors:

```
np.column_stack([vec, vec])
```

This produces a two-column matrix as a result:

```
array([[0, 0],
       [1, 1],
       [2, 2]])
```

Like with `np.hstack`, which stacks arrays horizontally, there's `np.vstack` that stacks arrays vertically:

```
np.vstack([vec, vec])
```

When we vertically stack two vectors, we get a matrix with two rows:

```
array([[0, 1, 2],
       [0, 1, 2]])
```

We can also stack two matrices vertically:

```
np.vstack([mat, mat])
```

The result is the same as `np.concatenate([mat, mat])` — we get a new matrix with six rows:

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 1],
       [2, 3],
       [4, 5]])
```

The `np.vstack` function can also stack together vectors and matrices, in effect creating a matrix with new rows:

```
np.vstack([vec, mat.T])
```

When we do it, `vec` becomes the first row in the new matrix:

```
array([[0, 1, 2],
       [0, 2, 4],
       [1, 3, 5]])
```

Note that in this code, we used the `T` property of `mat`. This is a matrix transposition operation, which changes rows of a matrix to columns:

```
mat.T
```

Originally, `mat` has the following data:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

After transposition, what was a column becomes a row:

```
array([[0, 2, 4],
       [1, 3, 5]])
```

C.2.5 Slicing and filtering

Like with Python lists, we can also use slicing for accessing a part of a NumPy array. For example, suppose we have a 5 × 3 matrix:

```
mat = np.arange(15).reshape(5, 3)
```

This matrix has five rows and three columns:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

We can access parts of this matrix by using slicing. For example, we can get the first free rows using the range operator (:):

```
mat[:3]
```

It returns rows indexed by 0, 1, and 2 (3 is not included):

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

If we need only rows 1 and 2, we specify both the beginning and the end of the range:

```
mat[1:3]
```

This gives us the rows we need:

```
array([[3, 4, 5],
       [6, 7, 8]])
```

Like with rows, we can select only some columns; for example, the first two columns:

```
mat[:, :2]
```

Here we have two ranges:

- The first one is simply a colon (:) with no start and end, which means "include all rows."
- The second one is a range that includes columns 0 and 1 (2 not included).

So as a result, we get

```
array([[ 0,  1],
       [ 3,  4],
       [ 6,  7],
       [ 9, 10],
       [12, 13]])
```

Of course, we can combine both and select any matrix part we want:

```
mat[1:3, :2]
```

This gives us rows 1 and 2 and columns 0 and 1:

```
array([[3, 4],
       [6, 7]])
```

If we don't need a range, but rather some specific rows or columns, we can simply provide a list of indices:

```
mat[[3, 0, 1]]
```

This gives us three rows indexed at 3, 0, and 1:

```
array([[ 9, 10, 11],
       [ 0,  1,  2],
       [ 3,  4,  5]])
```

Instead of individual indices, it's possible to use a binary mask to specify which rows to select. For example, suppose we want to choose rows where the first element of a row is an odd number.

To check if the first element is odd, we need to do the following:

1. Select the first column of the matrix.
2. Apply the mod 2 operation (%) to all the elements to compute the remainder of the division by 2.
3. If the remainder is 1, then the number is odd, and if 0, the number is even.

This translates to the following NumPy expression:

```
mat[:, 0] % 2 == 1
```

At the end it produces an array with Booleans:

```
array([False, True, False, True, False])
```

We see that the expression is `True` for rows 1 and 3 and `False` for rows 0, 2, and 5.

Now we can use this expression to select only rows where the expression is `True`:

```
mat[mat[:, 0] % 2 == 1]
```

This gives us a matrix with only two rows: rows 1 and 3:

```
array([[ 3,  4,  5],
       [ 9, 10, 11]])
```

C.3 Linear algebra

One of the reasons NumPy is so popular is its support of linear algebra operations. NumPy delegates all the internal computations to BLAS and LAPACK — time-proven libraries for efficient low-level computations — and this is why it's blazingly fast.

In this section, we take a short overview of the linear algebra operations we need throughout the book. We start with the most common ones: matrix and vector multiplications.

C.3.1 Multiplication

In linear algebra, we have multiple types of multiplication:

- Vector-vector multiplication: multiplying a vector by another vector
- Matrix-vector multiplication: multiplying a matrix by a vector
- Matrix-matrix multiplication: multiplying a matrix by another matrix

Let's take a closer look at each of them and see how to do them in NumPy.

VECTOR-VECTOR MULTIPLICATION

Vector-vector multiplication involves two vectors. It's typically called *dot product* or *scalar product*; it takes two vectors and produces a *scalar* — a single number.

Suppose we have two vectors, u and v, each of length n; then the dot product between u and v is

$$u^T v = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + ... + u_n v_n$$

NOTE In this appendix, the elements of a vector of length n are indexed from 0 to n–1: this way it's easier to map the concepts from the mathematical notation to NumPy.

This directly translates to Python. If we have two NumPy arrays `u` and `v`, the dot product between them is

```
dot = 0

for i in range(n):
    dot = u[i] * v[i]
```

Of course, we can take advantage of vectorized operations in NumPy and calculate it with a one-line expression:

```
(u * v).sum()
```

However, because it's quite a common operation, it's implemented inside NumPy in the `dot` method. So, to calculate the dot product, we simply invoke `dot`:

```
u.dot(v)
```

MATRIX-VECTOR MULTIPLICATION

Another type of multiplication is matrix-vector multiplication.

Suppose we have a matrix X of size m by n and a vector u of size n. If we multiply X by u, we get another vector of size m (figure C.12):

$$Xu = v$$

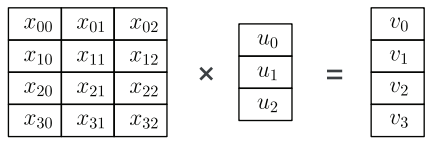


Figure C.12 When we multiply a 4 × 3 matrix by a vector of length 3, we get a vector of length 4.

We can think of the matrix X as a collection of n row-vectors x_i, each of size m (figure C.13).

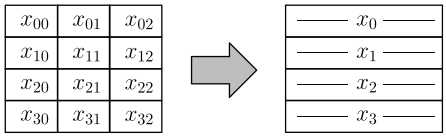


Figure C.13 We can think of the matrix X as of four row-vectors x_i, each of size 3.

Then we can represent matrix-vector multiplication Xu as m vector-vector multiplications between each row x_i and the vector u. The result is another vector — vector v (figure C.14).

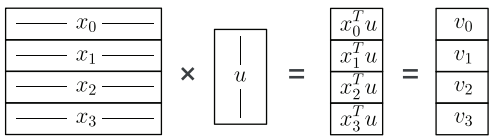


Figure C.14 The matrix-vector multiplication is a set of vector-vector multiplications: we multiply each row x_i of the matrix X by the vector u and as a result get the vector v.

Translating this idea to Python is straightforward:

```
v = np.zeros(m)
```
Creates an empty vector *v*

```
for i in range(m):
    v[i] = X[i].dot(u)
```
For each row x_i of X

Computes the ith element of *v* as a dot product x_i * *u*

Like with vector-vector multiplication, we can use the `dot` method of the matrix `X` (a two-dimensional array) to multiply it by vector `u` (a one-dimensional array):

```
v = X.dot(u)
```

The result is the vector `v` — a one-dimensional NumPy array.

MATRIX-MATRIX MULTIPLICATION

Finally, we have a matrix-matrix multiplication. Suppose we have two matrices, X of size m by n and U of size n by k. Then the result is another matrix V of size m by k (figure C.15):

$$XU = V$$

The easiest way to understand matrix-matrix multiplication is to consider U as a set of columns: u_0, u_1, …, u_{k-1} (figure C.16).

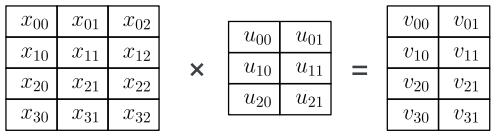


Figure C.15 When we multiply a 4 × 3 matrix *X* by a 3 × 2 matrix *U*, we get a 4 × 2 matrix *V*.

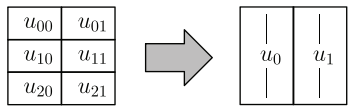


Figure C.16 We can think of *U* as a collection of column vectors. In this case, we have two columns: u_0 and u_1.

Then matrix-matrix multiplication XU is a set of matrix-vector multiplications Xu_i. The result of each multiplication is a vector v_i, which is the ith column of the resulting matrix V (figure C.17):

$$v_i = Xu_i$$

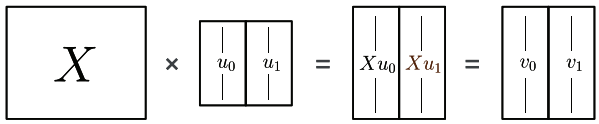


Figure C.17 We can think of matrix-matrix multiplication *XU* as a set of matrix-vector multiplications $v_i = Xu_i$, where u_is are the columns of *U*. The result is a matrix *V* with all the v_i's stacked together.

To implement it in NumPy, we can simply do this:

```
V = np.zeros((m, k))

for i in range(k):
    vi = X.dot(U[:, i])
    V[:, i] = vi
```

Creates an empty matrix *V*

For each column u_i of *U*

Computes v_i as matrix-vector multiplication *X* * u_i

Passing v_i as the *i*th column of *V*

Recall that `U[:, i]` means getting the ith column. Then we multiply `X` by that column and get `vi`. With `V[:, i]`, and because we have assignment (=), we overwrite the ith column of `V` with `vi`.

Of course, in NumPy there's a shortcut for that — it's again the `dot` method:

```
V = X.dot(U)
```

C.3.2 Matrix inverse

The inverse of a square matrix X is the matrix X^{-1} such that $X^{-1}X = I$, where I is the identity matrix. The identity matrix I doesn't change a vector when we perform matrix-vector multiplication:

$$Iv = v$$

Why do we need it? Suppose we have a system:

$$Ax = b$$

We know the matrix A and the resulting vector b, but don't know the vector x — we want to find it. In other words, we want to *solve* this system.

One of the possible ways of doing it is

- Compute A^{-1}, which is the inverse of A, and then
- Multiply both sides of the equation by the inverse A^{-1}

When doing so, we get

$$A^{-1}Ax = A^{-1}b$$

Because $A^{-1}A = I$, we have

$$Ix = A^{-1}b$$

Or

$$x = A^{-1}b$$

In NumPy, to compute the inverse, we use `np.linalg.inv`:

```
A = np.array([
    [0, 1, 2],
    [1, 2, 3],
    [2, 3, 3]
])

Ainv = np.linalg.inv(A)
```

For this particular square matrix `A`, it's possible to compute its inverse, so `Ainv` has the following values:

```
array([[-3.,  3., -1.],
       [ 3., -4.,  2.],
       [-1.,  2., -1.]])
```

We can verify that if we multiply the matrix with its inverse, we get the identity matrix:

```
A.dot(Ainv)
```

The result is indeed the identity matrix:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

> **NOTE** If all you want is to solve the equation $Ax = b$, then you don't really have to compute the inverse. From a computational point of view, calculating the inverse is an expensive operation. Instead, we should use `np.linalg.solve`, which is a magnitude faster:
>
> ```
> b = np.array([1, 2, 3])
> x = np.linalg.solve(A, b)
> ```
>
> In this book, when computing the weights for linear regression, we use the inverse for simplicity: it makes the code easier to understand.

There are matrices for which there's no inverse. First of all, it's not possible to invert nonsquare matrices. Also, not all square matrices can be inverted: there are *singular* matrices — matrices for which there exists no inverse matrix.

When we try to invert a singular matrix in NumPy, we get an error:

```
B = np.array([
    [0, 1, 1],
    [1, 2, 3],
    [2, 3, 5]
])

np.linalg.inv(B)
```

This code raises `LinAlgError`:

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-286-14528a9f848e> in <module>
      5 ])
      6
----> 7 np.linalg.inv(B)
```

```
<__array_function__ internals> in inv(*args, **kwargs)

<...>

LinAlgError: Singular matrix
```

C.3.3 Normal equation

In chapter 2, we used the normal equation to compute the weights vector for linear regression. In this section, we briefly outline how to arrive at the formula but without going into details. For more information, please refer to any linear algebra textbook.

This section may look math-heavy, but feel free to skip it: it will not affect your understanding of the book. If you studied the normal equation and linear regression in college, but already forgot most of it, this section should help you refresh your memory.

Suppose we have a matrix X with observations and a vector y with results. We want to find such vector w that

$$Xw = y$$

However, because X is not a square matrix, we cannot simply invert it, and the exact solution to this system doesn't exist. We can try to find an inexact solution and do the following trick. We multiply both sides by the transpose of X:

$$X^T Xw = X^T y$$

Now $X^T X$ is a square matrix, which should be possible to invert. Let's call this matrix C:

$$C = X^T X$$

The equation becomes

$$Cw = X^T y$$

In this equation, $X^T y$ is also a vector: when we multiply a matrix by a vector, we get a vector. Let's call it z. So now we have

$$Cw = z$$

This system now has an exact solution, which is the best approximation solution to the system we originally wanted to solve. Proving this is out of the scope of the book, so please refer to a textbook for more details.

To solve the system, we can invert C and multiply both sides by it:

$$C^{-1}Cw = C^{-1}z$$

Or

$$w = C^{-1}z$$

Now we have the solution for w. Let's rewrite it in terms of the original X and y:

$$w = (X^T X)^{-1} X^T y$$

This is the normal equation, which finds the best approximate solution w to the original system $Xw = y$.

It's quite simple to translate to NumPy:

```
C = X.T.dot(X)
Cinv = np.linalg.inv(C)
w = Cinv.dot(X.T).dot(y)
```

Now the array `w` contains the best approximate solution to the system.

appendix D
Introduction to Pandas

We don't expect any Pandas knowledge from the readers of this book. However, we use it extensively throughout the book. When we do, we try to explain the code, but it's not always possible to cover everything in detail.

In this appendix, we give a more in-depth introduction to Pandas, covering all the features we use in the chapters.

D.1 Pandas

Pandas is a Python library for working with tabular data. It's a popular and convenient tool for data manipulation. It's especially useful when preparing data for training machine learning models.

If you use Anaconda, it already has Pandas preinstalled. If not, install it with `pip`:

```
pip install pandas
```

To experiment with Pandas, let's create a notebook called appendix-d-pandas and use it for running the code from this appendix.

First, we need to import it:

```
import pandas as pd
```

Like with NumPy, we follow a convention and use an alias, `pd`, instead of the full name.

We start by exploring Pandas from its core data structures: DataFrames and Series.

D.1.1 DataFrame

In Pandas, a *DataFrame* is simply a table: a data structure with rows and columns (figure D.1).

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Figure D.1 A DataFrame in Pandas: a table with five rows and eight columns

To create a DataFrame, we first need to create some data that we'll put in the table. It can be a list of lists with some values:

```
data = [
    ['Nissan', 'Stanza', 1991, 138, 4, 'MANUAL', 'sedan', 2000],
    ['Hyundai', 'Sonata', 2017, None, 4, 'AUTOMATIC', 'Sedan', 27150],
    ['Lotus', 'Elise', 2010, 218, 4, 'MANUAL', 'convertible', 54990],
    ['GMC', 'Acadia',  2017, 194, 4, 'AUTOMATIC', '4dr SUV', 34450],
    ['Nissan', 'Frontier', 2017, 261, 6, 'MANUAL', 'Pickup', 32340],
]
```

This data is taken from the price-prediction dataset we use in chapter 2: we have some car characteristics like model, make, year of manufacture, and transmission type.

When creating a DataFrame, we need to know what each of the columns contains, so let's create a list with column names:

```
columns = [
    'Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
    'Transmission Type', 'Vehicle_Style', 'MSRP'
]
```

Now we're ready to create a DataFrame from it. For that, we use `pd.DataFrame`:

```
df = pd.DataFrame(data, columns=columns)
```

It creates a DataFrame with five rows and eight columns (figure D.1).

The first thing we can do with a DataFrame is look at the first few rows in the data to get an idea of what's inside. For that, we use the `head` method:

```
df.head(n=2)
```

It shows the first two rows of the DataFrame. The number of rows to display is controlled by the `n` parameter (figure D.2).

```
df.head(n=2)
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

Figure D.2 Previewing the content of a DataFrame with `head`

Alternatively, we can use a list of dictionaries to create a DataFrame:

```
data = [
    {
        "Make": "Nissan",
        "Model": "Stanza",
        "Year": 1991,
        "Engine HP": 138.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "sedan",
        "MSRP": 2000
    },
    ... # more rows
]

df = pd.DataFrame(data)
```

In this case, we don't need to specify column names: Pandas automatically takes them from the fields of the dictionaries.

D.1.2 Series

Each column in a DataFrame is a *Series* — a special data structure for containing values of one type. In a way, it's quite similar to one-dimensional NumPy arrays.

We can access the values of a column in two ways. First, we can use the dot notation (figure D.3, A):

```
df.Make
```

The other way is to use brackets notation (figure D.3, B):

```
df['Make']
```

The result is exactly the same: a Pandas Series with the values from the Make column.

If a column name contains spaces or other special characters, then we can use only the brackets notation. For example, to access the Engine HP column, we can use only brackets:

```
df['Engine HP']
```

```
df.Make
```

```
0     Nissan
1    Hyundai
2      Lotus
3        GMC
4     Nissan
Name: Make, dtype: object
```

(A) The dot notation

```
df['Make']
```

```
0     Nissan
1    Hyundai
2      Lotus
3        GMC
4     Nissan
Name: Make, dtype: object
```

(B) The brackets notation

Figure D.3 Two ways of accessing a column of a DataFrame: (A) the dot notation and (B) the brackets notation

The bracket notation is also more flexible. We can keep the name of a column in a variable and use it to access its content:

```
col_name = 'Engine HP'
df[col_name]
```

If we need to select a subset of columns, we again use brackets but with a list of names instead of a single string:

```
df[['Make', 'Model', 'MSRP']]
```

This returns a DataFrame with only three columns (figure D.4).

```
df[['Make', 'Model', 'MSRP']]
```

	Make	Model	MSRP
0	Nissan	Stanza	2000
1	Hyundai	Sonata	27150
2	Lotus	Elise	54990
3	GMC	Acadia	34450
4	Nissan	Frontier	32340

Figure D.4 To select a subset of columns of a DataFrame, use brackets with a list of names.

To add a column to a DataFrame, we also use the brackets notation:

```
df['id'] = ['nis1', 'hyu1', 'lot2', 'gmc1', 'nis2']
```

We have five rows in the DataFrame, so the list with values should also have five values. As a result, we have another column, id (figure D.5).

```
df['id'] = ['nis1', 'hyu1', 'lot2', 'gmc1', 'nis2']
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP	id
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000	nis1
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150	hyu1
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990	lot2
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450	gmc1
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340	nis2

Figure D.5 To add a new column, use the brackets notation.

In this case, id didn't exist, so we appended a new column to the end of the DataFrame. If id exists, then this code overwrites the existing values:

```
df['id'] = [1, 2, 3, 4, 5]
```

Now the content of the id column changes (figure D.6).

```
df['id'] = [1, 2, 3, 4, 5]
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP	id
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000	1
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150	2
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990	3
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450	4
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340	5

Figure D.6 To change the content of a column, use the brackets notation as well.

To delete a column, use the `del` operator:

```
del df['id']
```

After running it, this column disappears from the DataFrame.

D.1.3 Index

Both DataFrame (figure D.7, A) and Series (figure D.7, B) have numbers on the left; these numbers are called an *index*. The index describes how we can access rows from a DataFrame (or a Series).

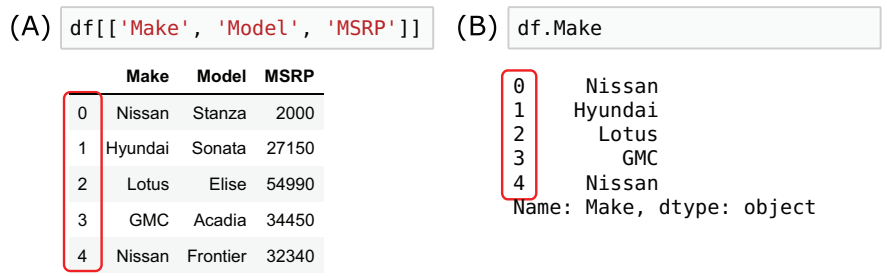


Figure D.7 Both DataFrame and Series have an index — the numbers on the left.

We can get the index of a DataFrame using the `index` property:

```
df.index
```

Because we didn't specify the index when creating a DataFrame, it uses the default one, a series of autoincrementing numbers starting from 0:

```
RangeIndex(start=0, stop=5, step=1)
```

The index behaves in the same way as a Series object, so everything that works for Series also works for the index.

Although a Series has only one index, a DataFrame has two: one for accessing rows, and the other for accessing columns. We already used Index for columns, when selecting individual columns from the DataFrame:

```
df['Make']
```

Uses the column index to get the Make column

To get the column names, we use the `columns` property (figure D.8):

```
df.columns
```

```
df.columns
```

```
Index(['Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
       'Transmission Type', 'Vehicle_Style', 'MSRP'],
      dtype='object')
```

Figure D.8 The `columns` property contains the column names.

D.1.4 Accessing rows

We can access rows in two ways: using `iloc` and `loc`.

First, let's start with `iloc`. We use it to access the rows of a DataFrame using their positional numbers. For example, to access the first row of the DataFrame, use the index 0:

```
df.iloc[0]
```

This returns the content of the first row:

```
Make                Nissan
Model               Stanza
Year                  1991
Engine HP              138
Engine Cylinders         4
Transmission Type    MANUAL
Vehicle_Style         sedan
MSRP                   2000
Name: 0, dtype: object
```

To get a subset of rows, pass a list with integers — row numbers:

```
df.iloc[[2, 3, 0]]
```

The result is another DataFrame containing only the rows we need (figure D.9).

```
df.iloc[[2, 3, 0]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.9 Using `iloc` to access rows of a DataFrame

We can use `iloc` for shuffling the content of a DataFrame. In our DataFrame, we have five rows. So, we can create a list of integers from 0 to 4 and shuffle it. Then we can use the shuffled list in `iloc`; this way, we'll get a DataFrame with all the rows shuffled.

Let's implement it. First, we create a range of size 5 using NumPy:

```
import numpy as np

idx = np.arange(5)
```

It creates an array with integers from 0 to 4:

```
array([0, 1, 2, 3, 4])
```

Now we can shuffle this array:

```
np.random.seed(2)
np.random.shuffle(idx)
```

As a result, we get

```
array([2, 4, 1, 3, 0])
```

Finally, we use this array with `iloc` to get the rows in shuffled order:

```
df.iloc[idx]
```

In the result, the rows are reordered according to the numbers in `idx` (figure D.10).

```
df.iloc[idx]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.10 Using `iloc` to shuffle the rows of a DataFrame

This doesn't change the DataFrame that we have in `df`. But we can reassign the `df` variable to the new DataFrame:

```
df = df.iloc[idx]
```

As a result, `df` now contains a shuffled DataFrame.

In this shuffled DataFrame, we can still use `iloc` to get rows by using their positional number. For example, if we pass `[0, 1, 2]` to `iloc`, we'll get the first three rows (figure D.11).

```
df.iloc[[0, 1, 2]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

Figure D.11 When using `iloc`, we get rows by their position.

However, you have probably noticed that the numbers on the left are not sequential anymore: when shuffling the DataFrame, we shuffled the index as well (figure D.12).

```
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.12 When shuffling the rows of a DataFrame, we also change the index: it's no longer sequential.

Let's check the index:

```
df.index
```

It's different now:

```
Int64Index([2, 4, 1, 3, 0], dtype='int64')
```

To use this index to access rows, we need `loc` instead of `iloc`. For example:

```
df.loc[[0, 1]]
```

As a result, we get a DataFrame with rows indexed by 0 and 1 — the last row and the row in the middle (Figure D.13).

```
df.loc[[0, 1]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

Figure D.13 When using `loc`, we get rows using the index, not the position.

It's quite different from `iloc`: `iloc` doesn't use the index. Let's compare them:

```
df.iloc[[0, 1]]
```

In this case, we also get a DataFrame with two rows, but these are the first two rows, indexed by 2 and 4 (figure D.14).

```
df.iloc[[0, 1]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Figure D.14 Unlike `loc`, `iloc` gets rows by the position, not index. In this case, we get rows at positions 0 and 1 (indexed by 2 and 4, respectively).

So, `iloc` doesn't look at the index at all; it uses only the actual position.

It's possible to replace the index and set it back to the default one. For that, we can use the `reset_index` method:

```
df.reset_index(drop=True)
```

It creates a new DataFrame with a sequential index (figure D.15).

```
df
```

	Make	Model	Year	Engine HP
2	Lotus	Elise	2010	218.0
4	Nissan	Frontier	2017	261.0
1	Hyundai	Sonata	2017	NaN
3	GMC	Acadia	2017	194.0
0	Nissan	Stanza	1991	138.0

`reset_index`

```
df.reset_index(drop=True)
```

	Make	Model	Year	Engine HP
0	Lotus	Elise	2010	218.0
1	Nissan	Frontier	2017	261.0
2	Hyundai	Sonata	2017	NaN
3	GMC	Acadia	2017	194.0
4	Nissan	Stanza	1991	138.0

Figure D.15 We can reset the index to sequential numbering by using `reset_index`.

D.1.5 Splitting a DataFrame

We can also use `iloc` to select subsets of a DataFrame. Suppose we want to split a DataFrame into three parts: train, validation, and test. We'll use 60% of data for training (three rows), 20% for validation (one row), and 20% for testing (one row):

```
n_train = 3
n_val = 1
n_test = 1
```

For selecting a range of rows, we use the slicing operator (:). It works for DataFrames in the same way it works for lists.

Thus, for splitting the DataFrame, we do the following:

```
df_train = df.iloc[:n_train]
df_val = df.iloc[n_train:n_train+n_val]
df_test = df.iloc[n_train+n_val:]
```

❶ Selects rows for train data
❷ Selects rows for validation data
❸ Selects rows for test data

In ❶, we get the train set: `iloc[:n_train]` selects rows from the start of the DataFrame until the row before `n_train`. For `n_train=3`, it selects rows 0, 1, and 2. Row 3 is not included.

In ❷, we get the validation set: `iloc[n_train:n_train+n_val]` selects rows from 3 to 3 + 1 = 4. It's not inclusive, so it takes only row 3.

In ❸, we get the test set: `iloc[n_train+n_val:]` selects rows from 3 + 1 = 4 until the end of the DataFrame. In our case, it's only row 4.

As a result, we have three DataFrames (figure D.16).

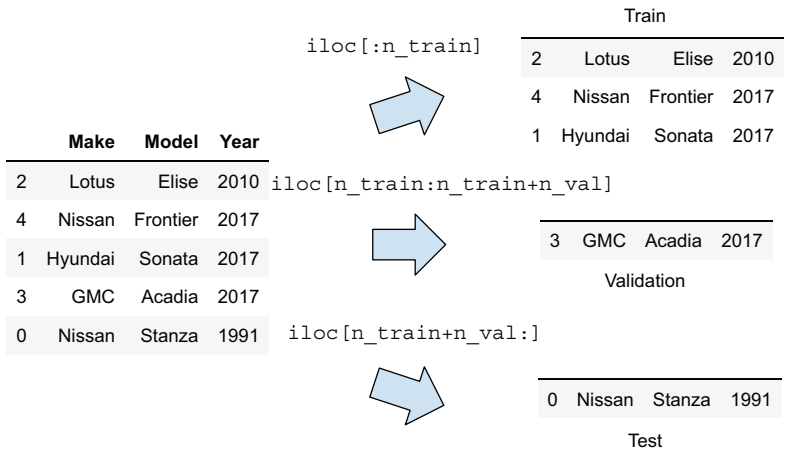


Figure D.16 Using `iloc` with the colon operator to split the DataFrame into train, validation, and test DataFrames

For more information about slicing in Python, refer to appendix B.

We've covered the basic Pandas data structures, so now let's see what we can do with them.

D.2 Operations

Pandas is a great tool for data manipulation, and it supports a wide variety of operations. We can group these operations into element-wise operations, summarizing operations, filtering, sorting, grouping, and more. In this section, we cover these operations.

D.2.1 Element-wise operations

In Pandas, Series support *element-wise* operations. Just as in NumPy, element-wise operations are applied to each element in a Series, and we get another Series as a result.

All basic arithmetic operations are element-wise: addition (+), subtraction (–), multiplication (*), and division (/). For element-wise operations, we don't need to write any loops: Pandas does it for us.

For example, we can multiply each element of a Series by 2:

```
df['Engine HP'] * 2
```

The result is another Series with each element multiplied by 2 (figure D.17).

```
df['Engine HP']
```

```
0    218.0
1    261.0
2      NaN
3    194.0
4    138.0
Name: Engine HP, dtype: float64
```

(A) Original Series

```
df['Engine HP'] * 2
```

```
0    436.0
1    522.0
2      NaN
3    388.0
4    276.0
Name: Engine HP, dtype: float64
```

(B) Result of multiplication

Figure D.17 As with NumPy arrays, all basic arithmetic operations for Series are element-wise.

As with arithmetic, logical operations are also element-wise:

```
df['Year'] > 2000
```

This expression returns a Boolean Series, with `True` for elements higher than 2000 (figure D.18).

```
df['Year']
```

```
0    2010
1    2017
2    2017
3    2017
4    1991
Name: Year, dtype: int64
```

(A) Original Series

```
df['Year'] > 2000
```

```
0     True
1     True
2     True
3     True
4    False
Name: Year, dtype: bool
```

(B) Result

Figure D.18 Logical operations are applied element-wise: in the results, we have `True` for all the elements that satisfy the condition.

We can combine multiple logical operations with logical and (&) or logical or (|):

```
(df['Year'] > 2000) & (df['Make'] == 'Nissan')
```

The result is also a Series. Logical operations are useful for filtering, which we cover next.

D.2.2 Filtering

Often, we need to select a subset of rows according to some criteria. For that, we use Boolean operations together with the bracket notation.

For example, to select all Nissan cars, put the condition inside the brackets:

```
df[df['Make'] == 'Nissan']
```

As a result, we have another DataFrame that contains only Nissans (figure D.19).

```
df[df['Make'] == 'Nissan']
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.19 To filter rows, put the condition for filtering inside brackets.

If we need a more complex selection condition, we combine multiple conditions with logical operators like and (&) and or (|).

For example, to select cars made after 2000 with automatic transmission, we use and (figure D.20):

```
df[(df['Year'] > 2010) & (df['Transmission Type'] == 'AUTOMATIC')]
```

```
df[(df['Year'] > 2010) & (df['Transmission Type'] == 'AUTOMATIC')]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450

Figure D.20 To use multiple selection criteria, combine them with logical and (&).

D.2.3 String operations

Although for NumPy arrays it's possible to do only arithmetic and logical element-wise operations, Pandas supports string operations: lowercasing, replacing substrings, and all the other operations that we can do on string objects.

Let's take a look at `Vehicle_Style`, which is one of the columns in the DataFrame. We see some inconsistencies in the data: sometimes names start with lowercase letters and sometimes with uppercase ones (figure D.21).

```
df['Vehicle_Style']
```

```
0    convertible
1         Pickup
2          Sedan
3        4dr SUV
4          sedan
Name: Vehicle_Style, dtype: object
```

Figure D.21 The Vehicle_Style column with some inconsistencies in the data

To resolve this, we can make everything lowercase. For usual Python strings, we'd use the `lower` function and apply it to all the elements of the series. In Pandas, instead of writing a loop, we use the special `str` accessor — it makes string operations element-wise and lets us avoid writing a `for` loop explicitly:

```
df['Vehicle_Style'].str.lower()
```

The result is a new Series with all the strings styled in lowercase (figure D.22).

```
df['Vehicle_Style'].str.lower()
```

```
0    convertible
1         pickup
2          sedan
3        4dr suv
4          sedan
Name: Vehicle_Style, dtype: object
```

Figure D.22 To lowercase all strings of a Series, use `lower`.

It's also possible to chain several string operations by using the `str` accessor multiple times (figure D.23):

```
df['Vehicle_Style'].str.lower().str.replace(' ', '_')
```

```
df['Vehicle_Style'].str.lower().str.replace(' ', '_')
```

```
0    convertible
1         pickup
2          sedan
3        4dr_suv
4          sedan
Name: Vehicle_Style, dtype: object
```

Figure D.23 To replace characters in all strings of a Series, use the `replace` method. It's possible to chain multiple methods together in one line.

Here, we make everything lowercase and replace spaces with underscores, all at once.

The column names of our DataFrame are also not consistent: sometimes there are spaces, and sometimes there are underscores (figure D.24).

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.24 The DataFrame: column names are not consistent.

We can also use string operations to normalize the column names:

```
df.columns.str.lower().str.replace(' ', '_')
As a result, we have:
Index(['make', 'model', 'year', 'engine_hp', 'engine_cylinders',
       'transmission_type', 'vehicle_style', 'msrp'],
      dtype='object')
```

This line of code returns new names, but it doesn't change the column names of the DataFrame. To modify them, we need to assign the results back to `df.columns`:

```
df.columns = df.columns.str.lower().str.replace(' ', '_')
```

When we do so, the column names change (figure D.25).

We can solve such inconsistency problems in all the columns of our DataFrame. For that, we need to select all the columns with strings and normalize them.

To select all strings, we can use the `dtype` property of a DataFrame (figure D.26).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

Figure D.25 The DataFrame after we normalized the column names

```
df.dtypes
```

```
make                object
model               object
year                 int64
engine_hp          float64
engine_cylinders     int64
transmission_type   object
vehicle_style       object
msrp                 int64
dtype: object
```

Figure D.26 The `dtypes` property returns the types of each column of a DataFrame.

All the strings columns have their dtype set to `object`. So, if we want to select them, we use filtering:

```
df.dtypes[df.dtypes == 'object']
```

That gives us a Series with `object` dtype columns only (figure D.27).

```
df.dtypes[df.dtypes == 'object']
```

```
make                object
model               object
transmission_type   object
vehicle_style       object
dtype: object
```

Figure D.27 To get only columns with strings, select the `object` dtype.

The actual names are stored in the index, so we need to get them:

```
df.dtypes[df.dtypes == 'object'].index
```

This gives us the following column names:

```
Index(['make', 'model', 'transmission_type', 'vehicle_style'], dtype='object')
```

Now we can use this list to iterate over string columns and apply the normalization for each column separately:

```
string_columns = df.dtypes[df.dtypes == 'object'].index

for col in string_columns:
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

This is what we have after running it (figure D.28).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.0	4	manual	convertible	54990
1	nissan	frontier	2017	261.0	6	manual	pickup	32340
2	hyundai	sonata	2017	NaN	4	automatic	sedan	27150
3	gmc	acadia	2017	194.0	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.0	4	manual	sedan	2000

Figure D.28 Both column names and values are normalized: names are lowercase, and spaces are replaced with underscores.

Next, we cover another type of operation: summarizing operations.

D.2.4 Summarizing operations

Just as we do in NumPy, in Pandas we have element-wise operations that produce another Series, as well as summarizing operations that produce a summary — one or multiple numbers.

Summarizing operations are quite useful for doing exploratory data analysis. For numerical fields, the operations are similar to what we have in NumPy. For example, to compute the average of all values in a column, we use the `mean` method:

```
df.msrp.mean()
```

Other methods that we can use include

- `sum`: Computes the sum of all values
- `min`: Gets the smallest number in the Series

- `max`: Gets the largest number in the Series
- `std`: Computes the standard deviation

Instead of checking these things separately, we can use `describe` to get all these values at once:

```
df.msrp.describe()
```

It creates a summary with the number of rows, mean, min, and max, as well as standard deviation and other characteristics:

```
count        5.000000
mean     30186.000000
std      18985.044904
min       2000.000000
25%      27150.000000
50%      32340.000000
75%      34450.000000
max      54990.000000
Name: msrp, dtype: float64
```

When we invoke `mean` on the entire DataFrame, it computes the mean value for all the numerical columns:

```
df.mean()
```

In our case, we have four numerical columns, so we get the average for each:

```
year                  2010.40
engine_hp              202.75
engine_cylinders         4.40
msrp                 30186.00
dtype: float64
```

Likewise, we can use `describe` on a DataFrame:

```
df.describe()
```

Because `describe` already returns a Series, when we invoke it on a DataFrame, we get a DataFrame as well (figure D.29).

```
df.describe().round(2)
```

	year	engine_hp	engine_cylinders	msrp
count	5.00	4.00	5.00	5.00
mean	2010.40	202.75	4.40	30186.00
std	11.26	51.30	0.89	18985.04
min	1991.00	138.00	4.00	2000.00
25%	2010.00	180.00	4.00	27150.00
50%	2017.00	206.00	4.00	32340.00
75%	2017.00	228.75	4.00	34450.00
max	2017.00	261.00	6.00	54990.00

Figure D.29 To get the summary statistics of all numerical features, use the `describe` method.

D.2.5 Missing values

We didn't focus on it previously, but we have a missing value in our data: we don't know the value of engine_hp for row 2 (figure D.30).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.0	4	manual	convertible	54990
1	nissan	frontier	2017	261.0	6	manual	pickup	32340
2	hyundai	sonata	2017	NaN	4	automatic	sedan	27150
3	gmc	acadia	2017	194.0	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.0	4	manual	sedan	2000

Figure D.30 There's one missing value in our DataFrame.

We can see which values are missing using the `isnull` method:

```
df.isnull()
```

This method returns a new DataFrame where a cell is True if the corresponding value is missing in the original DataFrame (figure D.31).

However, when we have large DataFrames, looking at all the values is impractical. We can easily summarize them by running the `sum` method on the results:

```
df.isnull().sum()
```

```
df.isnull()
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False
2	False	False	False	True	False	False	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False

Figure D.31 To find missing values, use the `isnull` method.

It returns a Series with the number of missing values per column. In our case, only engine_hp has missing values; others don't (figure D.32).

```
df.isnull().sum()
```

```
make                0
model               0
year                0
engine_hp           1
engine_cylinders    0
transmission_type   0
vehicle_style       0
msrp                0
dtype: int64
```

Figure D.32 To find columns with missing values, use `isnull` followed by `sum`.

To replace the missing values with some actual values, we use the `fillna` method. For example, we can fill the missing values with zero:

```
df.engine_hp.fillna(0)
```

As a result, we get a new Series where NaNs are replaced by 0:

```
0    218.0
1    261.0
2      0.0
3    194.0
4    138.0
Name: engine_hp, dtype: float64
```

Alternatively, we can replace it by getting the mean:

```
df.engine_hp.fillna(df.engine_hp.mean())
```

In this case, the NaNs are replaced by the average:

```
0    218.00
1    261.00
2    202.75
3    194.00
4    138.00
Name: engine_hp, dtype: float64
```

The `fillna` method returns a new Series. Thus, if we need to remove the missing values from our DataFrame, we need to write the results back:

```
df.engine_hp = df.engine_hp.fillna(df.engine_hp.mean())
```

Now we get a DataFrame without missing values (figure D.33).

```
df.engine_hp = df.engine_hp.fillna(df.engine_hp.mean())
df
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.00	4	manual	convertible	54990
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.00	4	manual	sedan	2000

Figure D.33 The DataFrame without missing values

D.2.6 Sorting

The operations we covered previously were mostly used for Series. We also can perform operations on DataFrames.

Sorting is one of these operations: it rearranges the rows in a DataFrame such that they are sorted by the values of some column (or multiple columns).

For example, let's sort the DataFrame by MSRP. For that, we use the `sort_values` method:

```
df.sort_values(by='msrp')
```

The result is a new DataFrame where rows are sorted from the smallest MSRP (2000) to the largest (54990) (figure D.34).

```
df.sort_values(by='msrp')
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
4	nissan	stanza	1991	138.00	4	manual	sedan	2000
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
0	lotus	elise	2010	218.00	4	manual	convertible	54990

Figure D.34 To sort the rows of a DataFrame, use `sort_values`.

If we want the largest values to appear first, we set the `ascending` parameter to `False`:

```
df.sort_values(by='msrp', ascending=False)
```

Now we have the MSRP of 54990 in the first row and 2000 in the last (figure D.35).

```
df.sort_values(by='msrp', ascending=False)
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.00	4	manual	convertible	54990
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
4	nissan	stanza	1991	138.00	4	manual	sedan	2000

Figure D.35 To sort the rows of a DataFrame in descending order, use `ascending=False`.

D.2.7 Grouping

Pandas offers quite a few summarizing operations: sum, mean, and many others. We previously have seen how to apply them to calculate a summary over the entire DataFrame. Sometimes, however, we'd like to do it per group — for example, calculate the average price per transmission type.

In SQL, we'd write something like this:

```
SELECT
    tranmission_type,
    AVG(msrp)
FROM
    cars
```

```
GROUP BY
    transmission_type;
```

In Pandas, we use the `groupby` method:

```
df.groupby('transmission_type').msrp.mean()
```

The result is the average price per transmission type:

```
transmission_type
automatic    30800.000000
manual       29776.666667
Name: msrp, dtype: float64
```

If we'd like to also compute the number of records per each type along with the average price, in SQL, we'd add another statement in the `SELECT` clause:

```
SELECT
    tranmission_type,
    AVG(msrp),
    COUNT(msrp)
FROM
    cars
GROUP BY
    transmission_type
```

In Pandas, we use `groupby` followed by `agg` (short for "aggregate"):

```
df.groupby('transmission_type').msrp.agg(['mean', 'count'])
```

As a result, we get a DataFrame (figure D.36).

```
df.groupby('transmission_type').msrp.agg(['mean', 'count'])
```

transmission_type	mean	count
automatic	30800.000000	2
manual	29776.666667	3

Figure D.36 When grouping, we can apply multiple aggregate functions using the `agg` method.

Pandas is quite a powerful tool for data manipulation, and it's often used to prepare data before training a machine learning model. With the information from this appendix, it should be easier for you to understand the code in this book.

appendix E
AWS SageMaker

AWS SageMaker is a set of services from AWS related to machine learning. SageMaker makes it easy to create a server on AWS with Jupyter installed on it. The notebooks are already configured: they have most of the libraries we need, including NumPy, Pandas, Scikit-learn, and TensorFlow, so we can just use them for our projects!

E.1 AWS SageMaker Notebooks

SageMaker's notebooks are especially interesting for training neural networks for two reasons:

- We don't need to worry about setting up TensorFlow and all the libraries.
- It's possible to rent a computer with a GPU, which enables us to train neural networks a lot faster.

To use a GPU, we need to adjust the default quotas. In the next section, we tell you how to do this.

E.1.1 Increasing the GPU quota limits

Each account on AWS has quota limits. For example, if our quota limit on the number of instances with GPUs is 10, we cannot request an eleventh instance with a GPU.

By default, the quota limit is zero, which means that it's not possible to rent a GPU machine without changing the quota limits.

To request an increase, open the support center in AWS Console: click Support in the top-right corner and select Support Center (figure E.1).

Next, click the Create Case button (figure E.2).

Now select the Service Limit Increase option. In the Case Details section, select SageMaker from the Limit Type dropdown list (figure E.3).

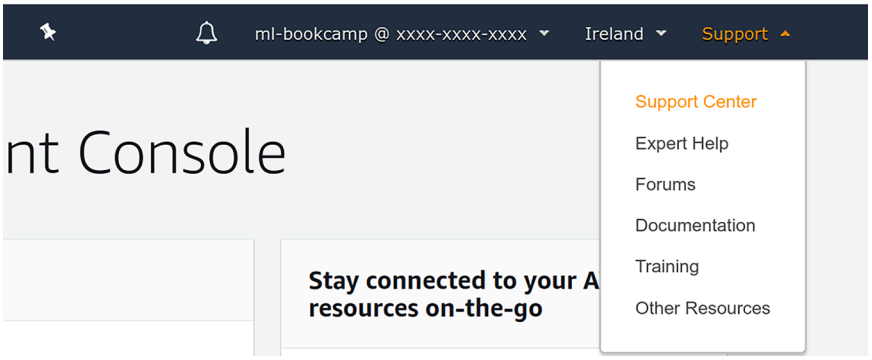


Figure E.1 To open the support center, click Support > Support Center.

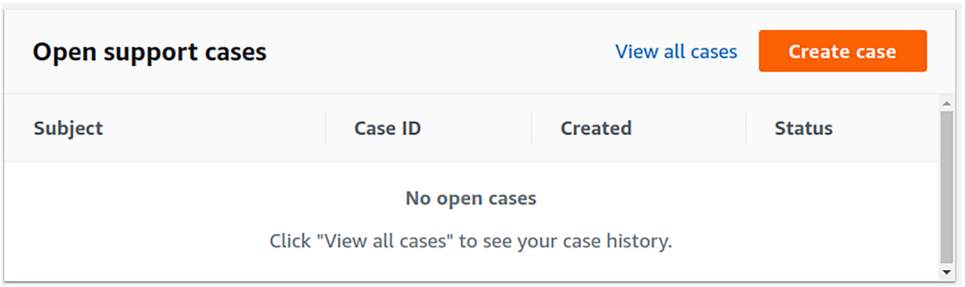


Figure E.2 In the support center, click the Create Case button.

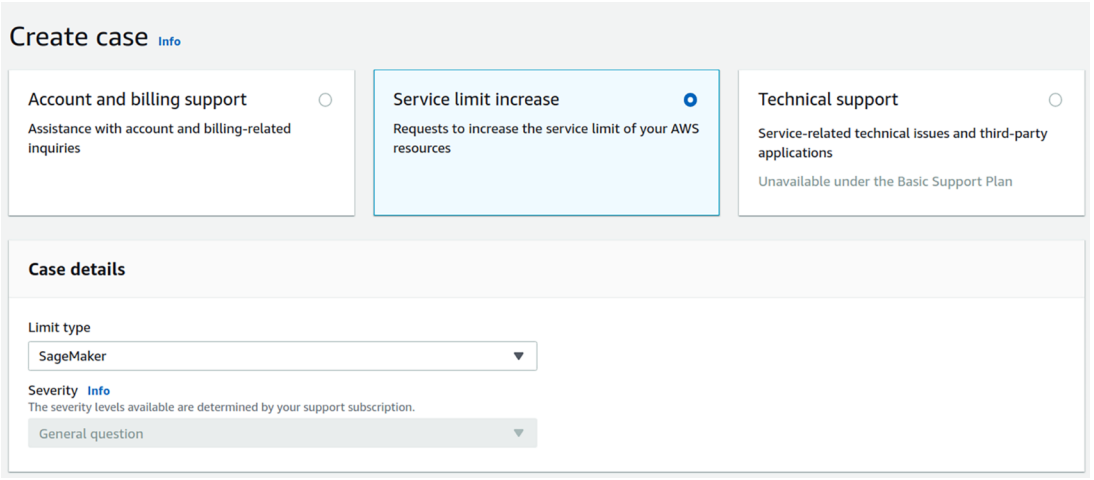


Figure E.3 When creating a new case, select Service Limit Increase > SageMaker.

After that, fill in the quota increase form (figure E.4):

- Region: select the closest to you or the cheapest. You can see the prices here: https://aws.amazon.com/sagemaker/pricing/. Resource type: SageMaker Notebooks.
- Limit: ml.p2.xlarge instances for a machine with one GPU.
- New limit value: 1.

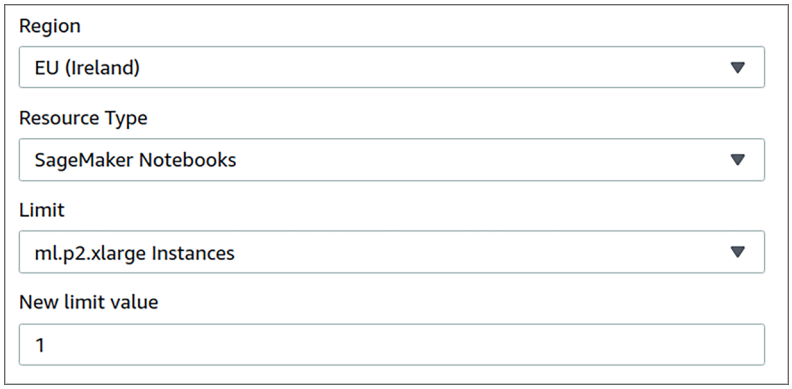


Figure E.4 Increase the limit for ml.p2.xlarge to one instance.

Finally, describe why you need an increase in quota limits. For example, you can type "I'd like to train a neural network using a GPU machine" (figure E.5).

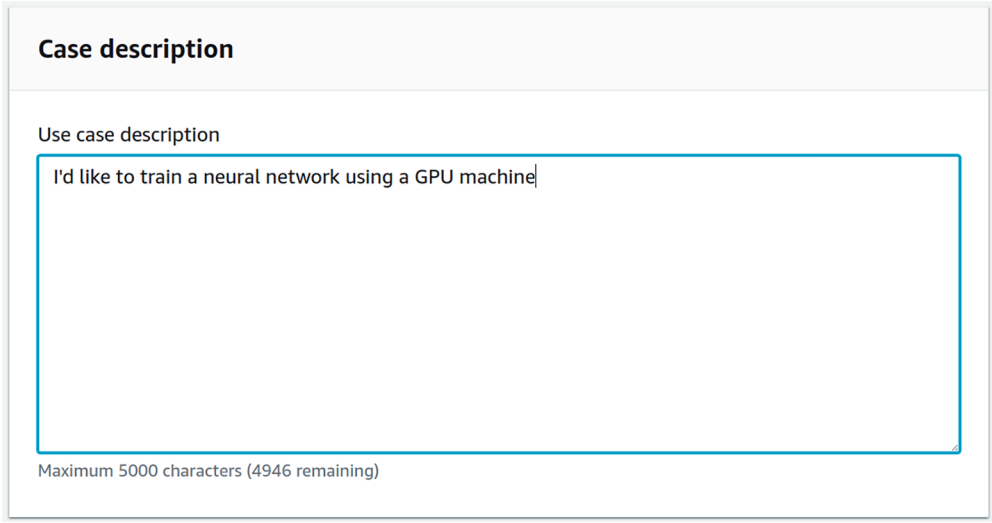


Figure E.5 We need to explain why we want to increase the limit.

We're ready; now click Submit.

After that, we see some details of the request. Back in the Support Center, we see the new case in the list of open cases (figure E.6).

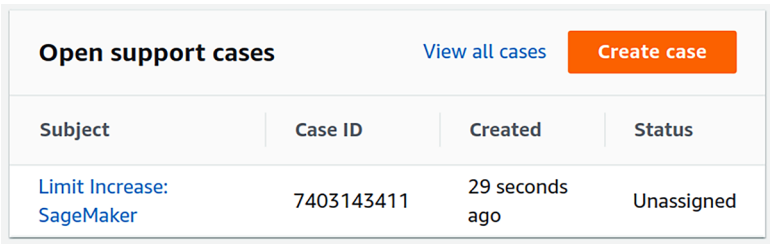


Figure E.6 The list of open support cases

It typically takes one to two days to process the request and increase the limits.

Once the limit is increased, we can create a Jupyter Notebook instance with a GPU.

E.1.2 Creating a notebook instance

To create a Jupyter Notebook in SageMaker, first find SageMaker in the list of services (figure E.7).

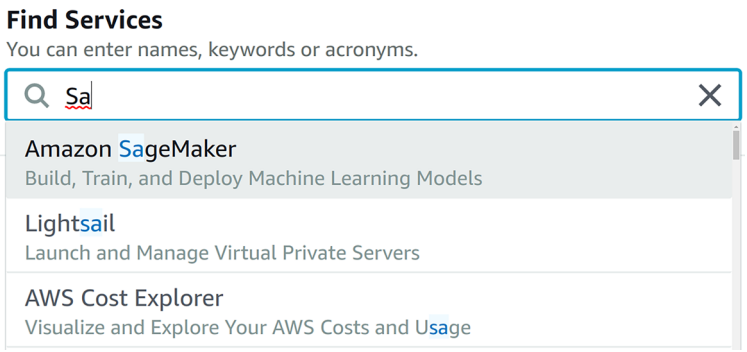


Figure E.7 To find SageMaker, type SageMaker in the search box.

NOTE SageMaker notebooks are not covered by the free tier, so it costs money to rent a Jupyter Notebook.

For an instance with one GPU (ml.p2.xlarge), the cost of one hour at the moment of writing is

- Frankfurt: $1.856
- Ireland: $1.361
- Northern Virginia: $1.26

The project from chapter 7 requires one to two hours to complete.

NOTE Make sure you are in the same region where you requested the quota limits increase.

In SageMaker, select Notebook Instances, and then click the Create Notebook Instance button (figure E.8).

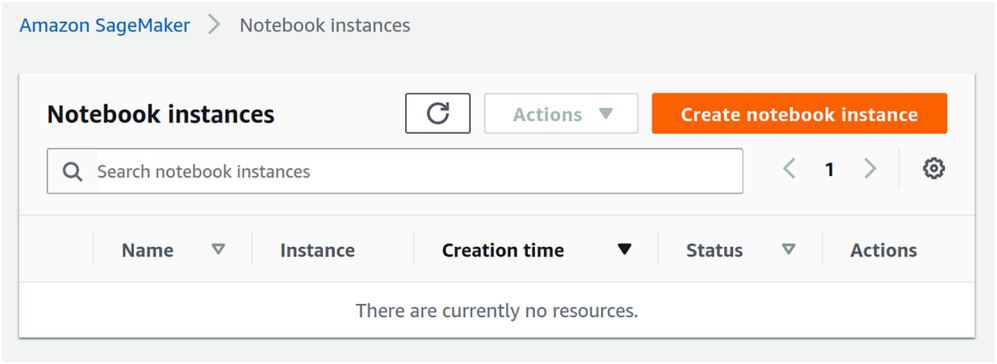


Figure E.8 To create a Jupyter Notebook, click Create Notebook Instance.

Next, we need to configure the instance. First, enter the name of the instance as well as the instance type. Because we're interested in a GPU instance, select ml.p2.xlarge in the Accelerated Computing section (figure E.9).

In Additional Configuration, write 5 GB in the Volume Size field. This way, we should have enough space to store the dataset as well as save our models.

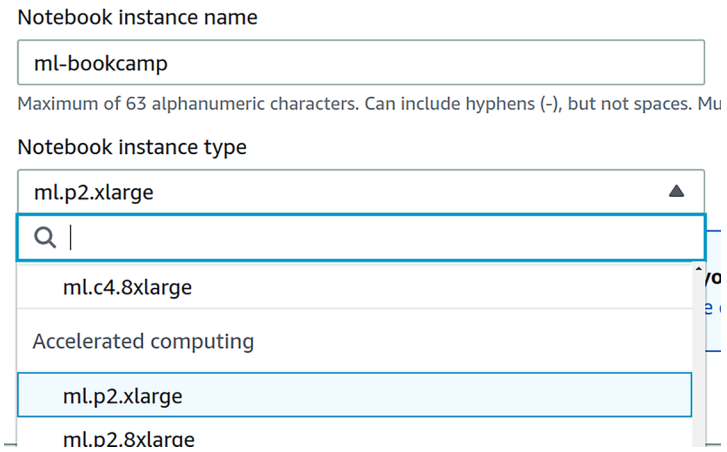


Figure E.9 The Accelerated Computing section contains instances with GPUs.

If you previously used SageMaker and already have an IAM role for it, select it in the IAM Role section.

But if you're doing it for the first time, select Create a New Role (figure E.10).

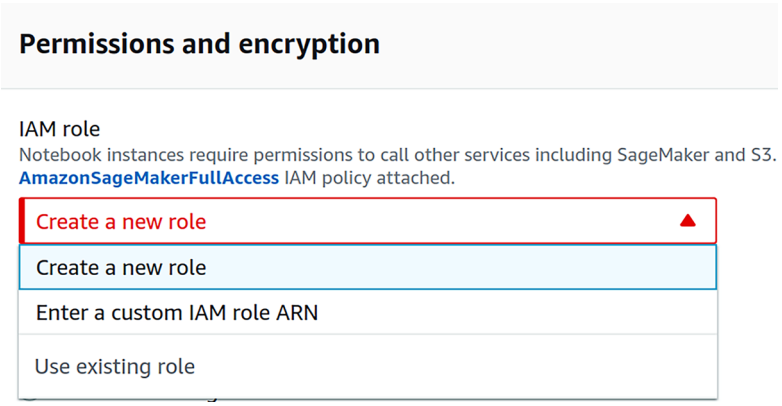


Figure E.10 To use a SageMaker notebook, we need to create an IAM role for it.

When creating the role, keep the default values, and click the Create Role button (figure E.11).

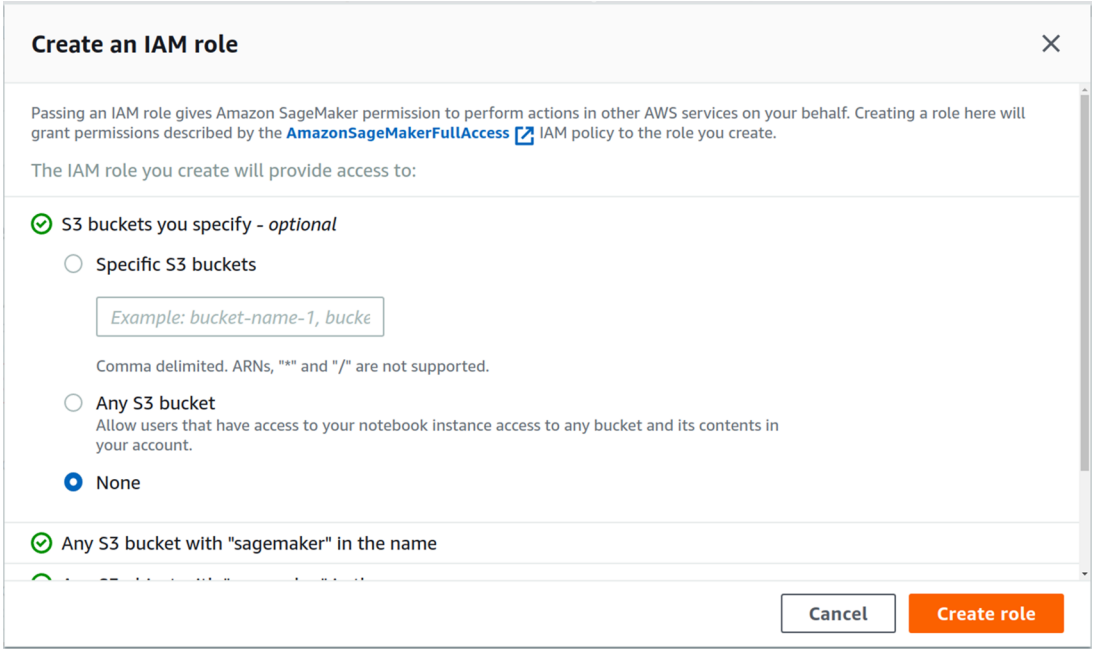


Figure E.11 The default values for the new IAM role are sufficient.

Keep the rest of the options unchanged:

- Root access: Enable
- Encryption key: No custom encryption
- Network: No VPC
- Git repositories: None

Finally, click the Create Notebook Instance to launch it.

If for some reasons you see a ResourceLimitExceeded error message (figure E.12), make sure that

- You have requested an increase in quota limits for the ml.p2.xlarge instance type.
- The request was processed.
- You're trying to create a notebook in the same region where you requested the increase.

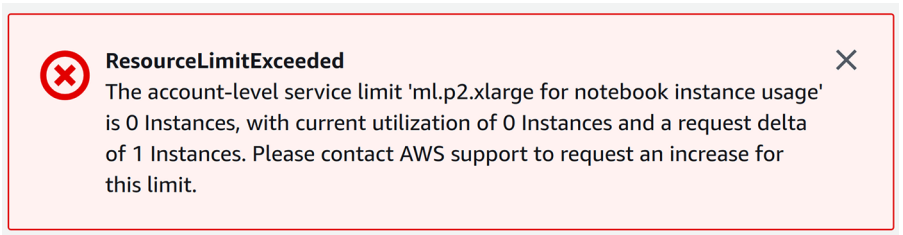


Figure E.12 If you see the ResourceLimitExceeded error message, you need to increase the quota limits.

After creating an instance, the notebook appears in the list of notebook instances (figure E.13).

Now we need to wait until the notebook changes status from Pending to InService; this may take one to two minutes.

Once it's in the InService state, it's ready to be used (figure E.14). Click Open Jupyter to access it.

Next, we see how to use it with TensorFlow.

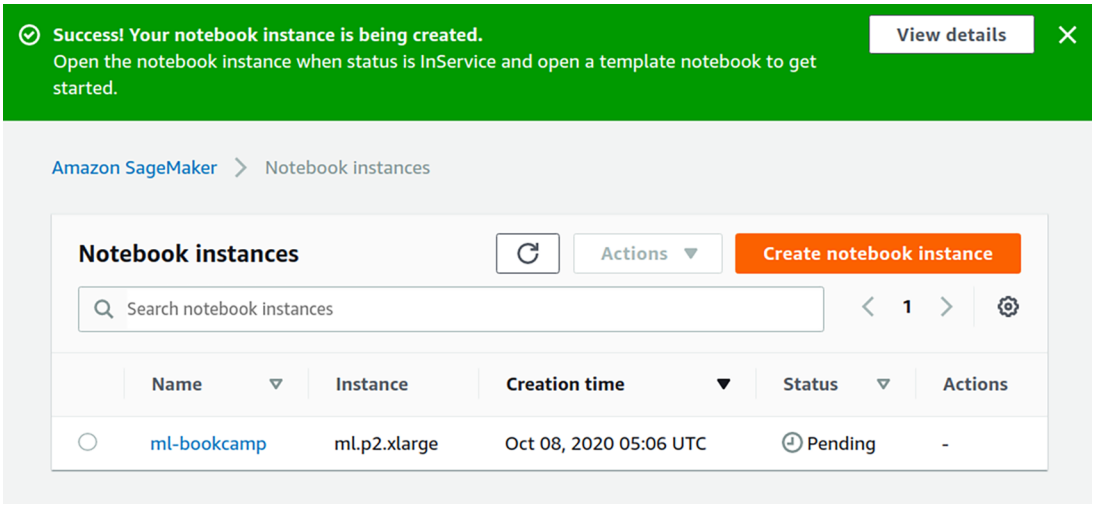


Figure E.13 Success! The notebook instance has been created.

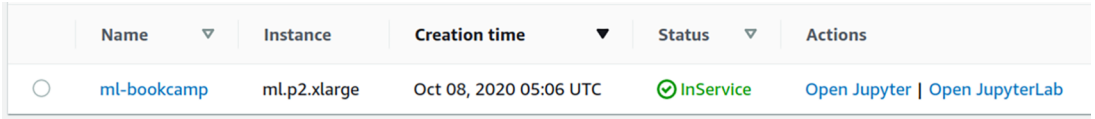


Figure E.14 The new notebook instance is in service and ready to be used.

E.1.3 Training a model

After clicking Open Jupyter, we see the familiar interface forJupyter Notebook.

To create a new notebook, click New, and select conda_tensorflow2_p36 (figure E.15).

This notebook has Python version 3.6 and TensorFlow version 2.1.0. At the time of writing, this is the newest version of TensorFlow available in SageMaker.

Now, import TensorFlow and check its version:

```
import tensorflow as tf
tf.__version__
```

The version should be 2.1.0 or higher (figure E.16).

Now go to chapter 7 and train a neural network! After training is finished, we need to turn off the notebook.

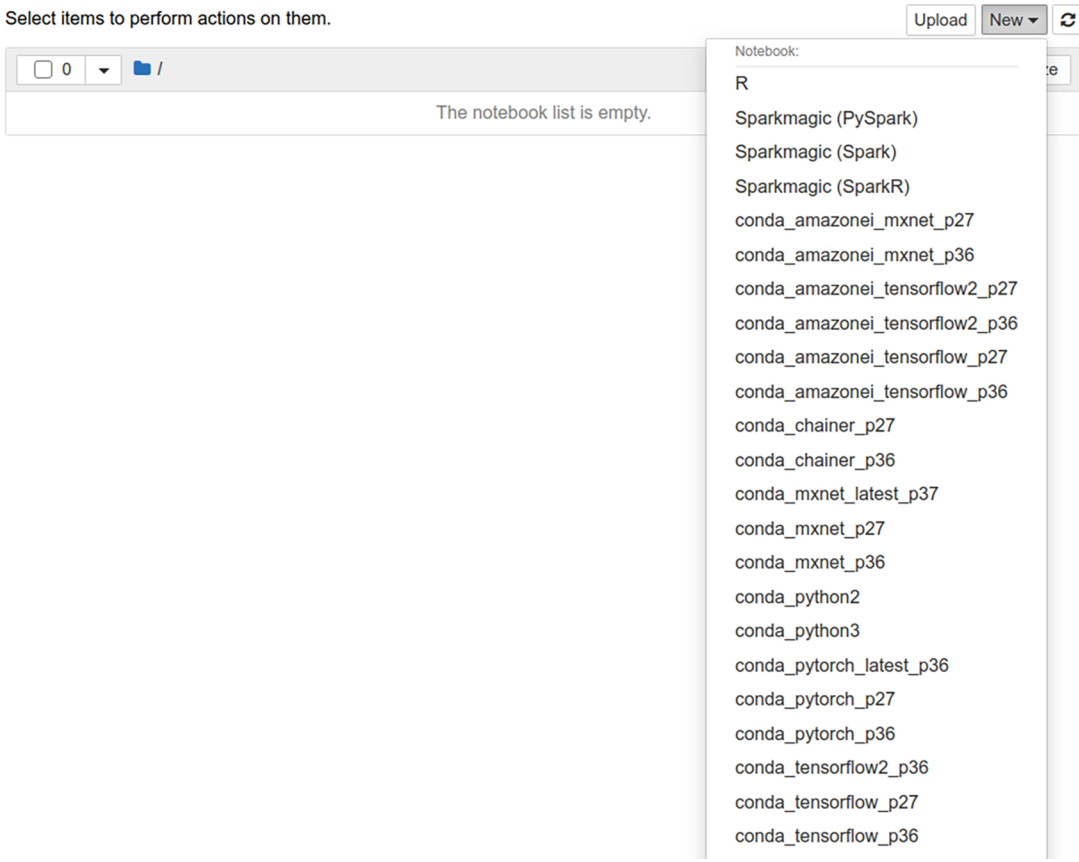


Figure E.15 To create a new notebook with TensorFlow, select conda_tensorflow2_p36.

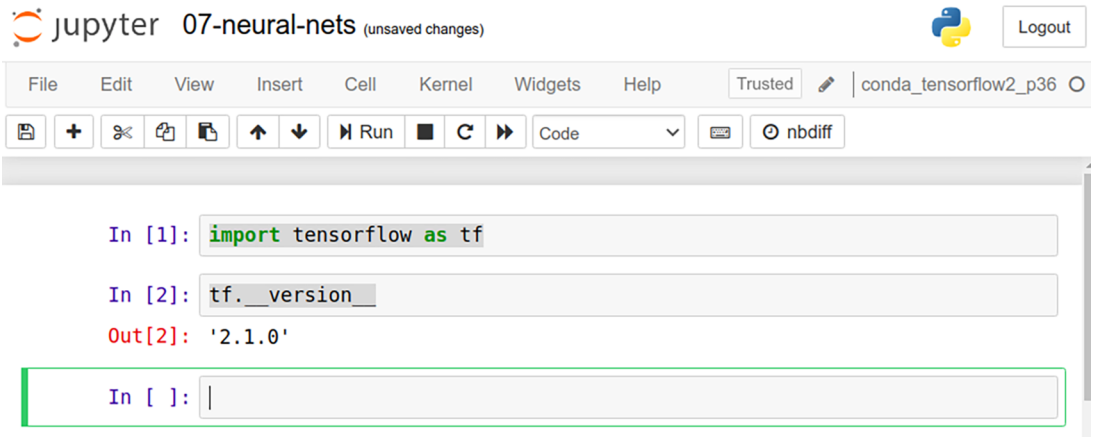


Figure E.16 For our examples, we need at least TensorFlow version 2.1.0.

E.1.4 Turning off the notebook

To stop a notebook, first select the instance you want to stop, and then select Stop in the Actions dropdown list (figure E.17).

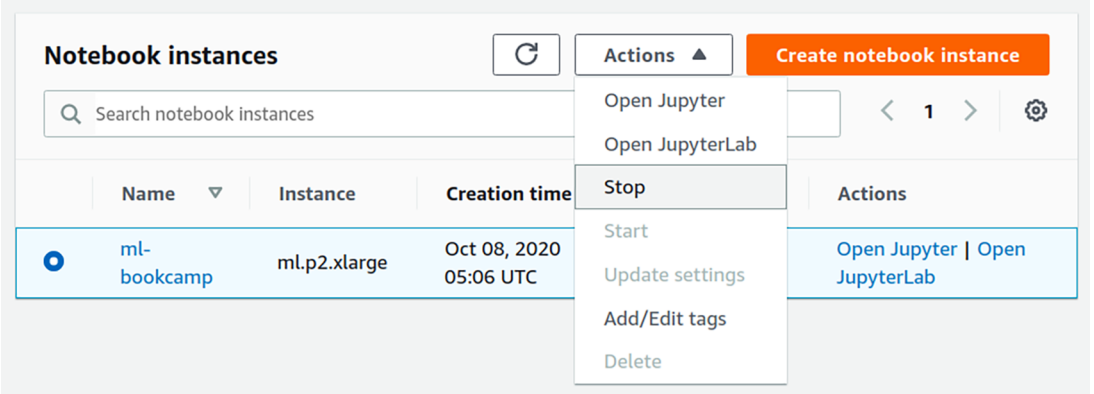


Figure E.17 To turn off a notebook, select the Stop action.

After doing this, the status of the notebook will change from InService to Stopping. It may take a few minutes before it fully stops and changes status from Stopping to Stopped.

> **NOTE** When we stop a notebook, all our code and data are saved. The next time we start it, we can continue where we left off.

> **IMPORTANT** The notebook instances are expensive, so make sure you don't accidentally leave it running. SageMaker is not covered by the free tier, so if you forget to stop it, you'll receive a huge bill at the end of a month. There's a way to set a budget in AWS to avoid huge bills. See the documentation about managing costs at AWS: https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/budgets-managing-costs.html. Be careful, and turn off your notebook when you no longer need it.

Once you finish working on a project, you can delete the notebook. Select a notebook, and then choose Delete from the dropdown list (figure E.18). The notebook must be in the Stopped state to delete it.

It will first change status from Stopped to Deleting, and after 30 seconds, it will disappear from the list of notebooks.

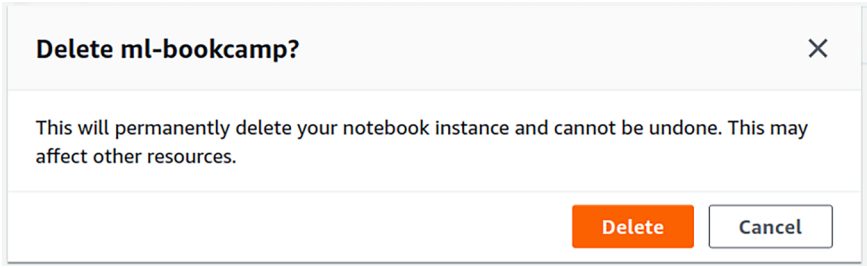


Figure E.18 After you finish chapter 7, you can delete the notebook.

index

C

D

E

F

M

N

O

P

R

S

T

U

V

W

X

Y

Z

DATA SCIENCE/PYTHON

Machine Learning Bookcamp

Alexey Grigorev

Master key machine learning concepts as you build actual projects! Machine learning is what you need for analyzing customer behavior, predicting price trends, evaluating risk, and much more. To master ML, you need great examples, clear explanations, and lots of practice. This book delivers all three!

Machine Learning Bookcamp presents realistic, practical machine learning scenarios, along with crystal-clear coverage of key concepts. In it, you'll complete engaging projects, such as creating a car price predictor using linear regression and deploying a churn prediction service. You'll go beyond the algorithms and explore important techniques like deploying ML applications on serverless systems and serving models with Kubernetes and Kubeflow. Dig in, get your hands dirty, and have fun building your ML skills!

What's Inside

- Collect and clean data for training models
- Use popular Python tools, including NumPy, Scikit-Learn, and TensorFlow
- Deploy ML models to a production-ready environment

Python programming skills assumed. No previous machine learning knowledge is required.

Alexey Grigorev is a principal data scientist at OLX Group. He runs DataTalks.Club, a community of people who love data.

"The best hands-on guide to begin your machine learning journey."

—Gustavo Filipe Ramos Gomes Troido

"Great theory, real-life examples, and just the right amount of code. An absolute feast for a beginner who wants to enter the realm of machine learning."

—Krishna Chaitanya Anipindi Hexagon

"A practical guide for anyone aspiring to be a data scientist."

—Amaresh Rajasekharan IBM Corporation

"Exactly the practice I needed to be comfortable with machine learning."

—Nathan D'Elboux, Isparex

ISBN: 978-1-61729-681-9

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]