

Getting Started with Machine Learning

Before we get into the ins and outs of ML, let's consider how it evolved from traditional programming. We'll start by examining what traditional programming is, then consider cases where it is limited. Then we'll see how ML evolved to handle those cases, and as a result has opened up new opportunities to implement new scenarios, unlocking many of the concepts of artificial intelligence. Traditional programming involves us writing rules, expressed in a programming language, that act on data and give us answers. This applies just about everywhere that something can be programmed with code. For example, consider a game like the popular Breakout. Code determines the movement of the ball, the score, and the various conditions for winning or losing the game.

Think about the scenario where the ball bounces off a brick

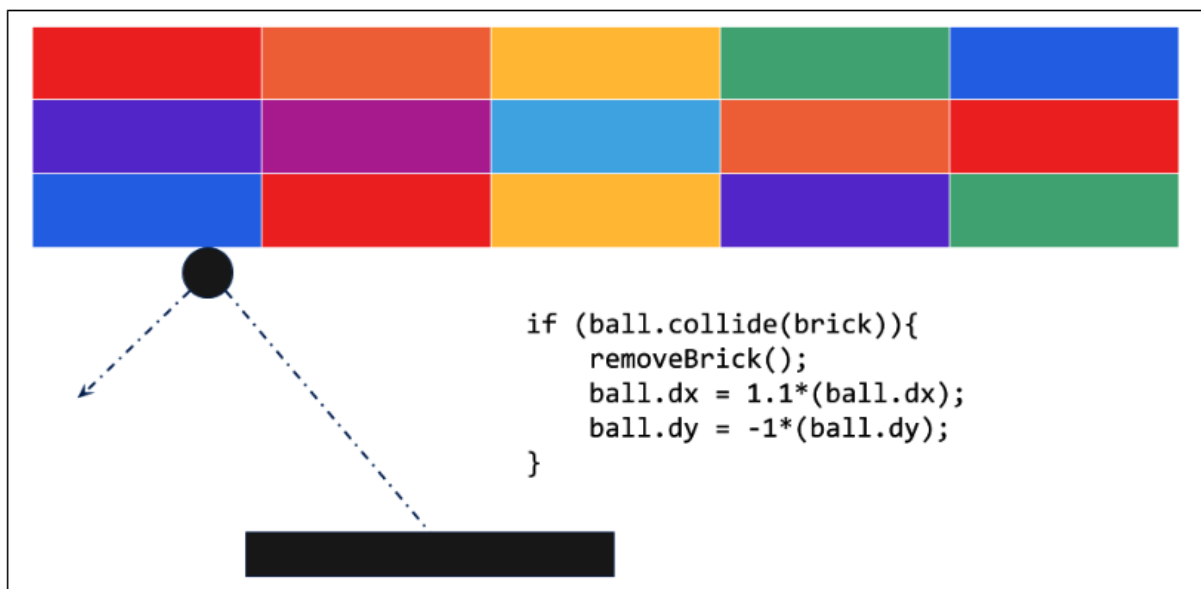


Figure 1-1. Code in a Breakout game

Here, the motion of the ball can be determined by its dx and dy properties. When it hits a brick, the brick is removed, and the velocity of the ball increases and changes direction and the game will end if we hit the lower border. The code acts on data about the game situation.

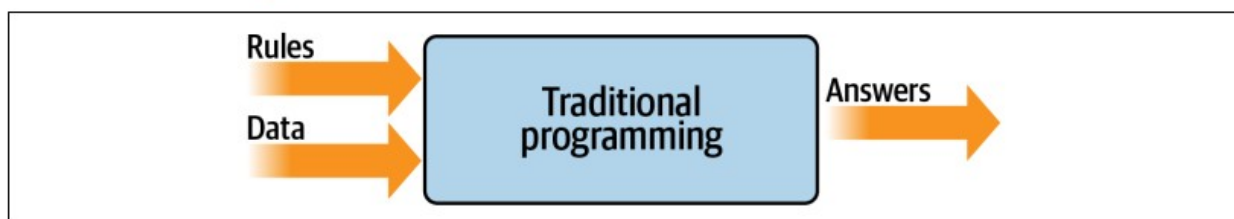


Figure 1-3. High-level view of traditional programming

As you can see, you have rules expressed in a programming language. These rules act on data, and the result is answers.

Limitations of Traditional Programming

Traditional programming has limitation: namely, that only scenarios that can be implemented are ones for which you can derive rules. What about other scenarios? Usually, they are infeasible to develop because the code is too complex. It's just not possible to write code to handle them.

Consider, for example, activity detection. Fitness monitors that can detect our activity are a recent innovation, not just because of the availability of cheap and small hardware, but also because the algorithms to handle detection weren't previously feasible. Let's explore why.

shows a naive activity detection algorithm for walking. It can consider the person's speed. If it's less than a particular value, we can determine that they are probably walking.

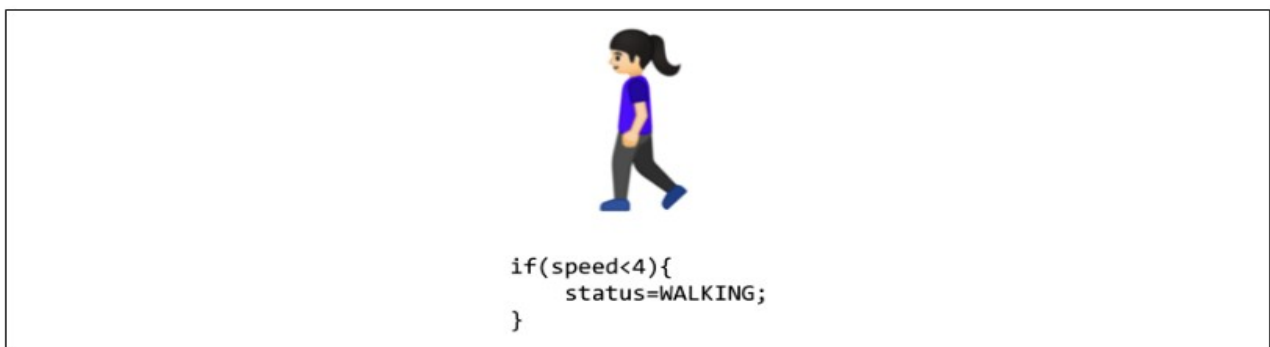


Figure 1-4. Algorithm for activity detection

Given that our data is speed, we could extend this to detect if they are running



Figure 1-5. Extending the algorithm for running

As you can see, going by the speed, we might say if it is less than a particular value (say, 4 mph) the person is walking, and otherwise they are running. It still sort of works.

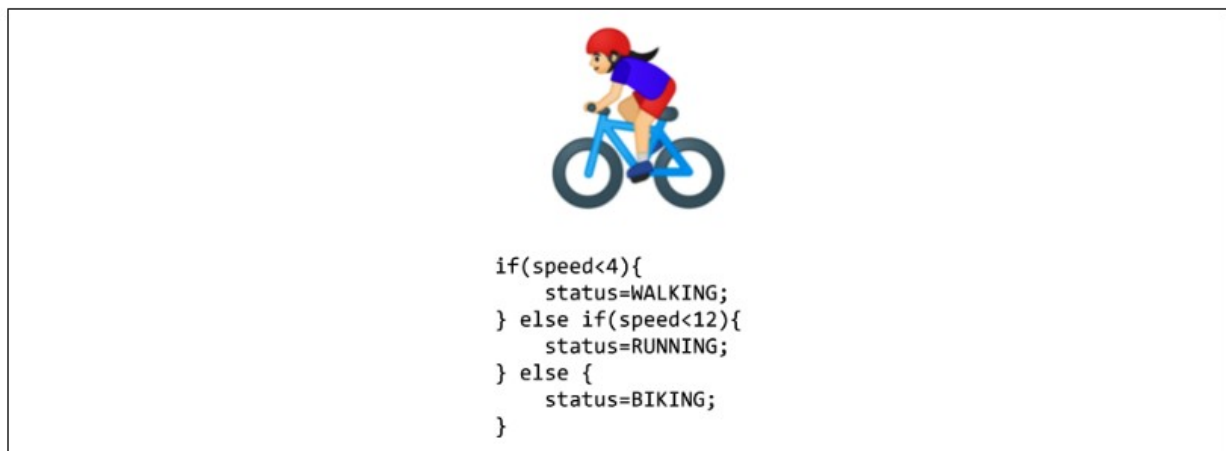


Figure 1-6. Extending the algorithm for biking

I know it's naive in that it just detects speed—some people run faster than others, and you might run downhill faster than you cycle uphill, for example. But on the whole, it still works. However, what happens if we want to implement another scenario, such as golfing ?

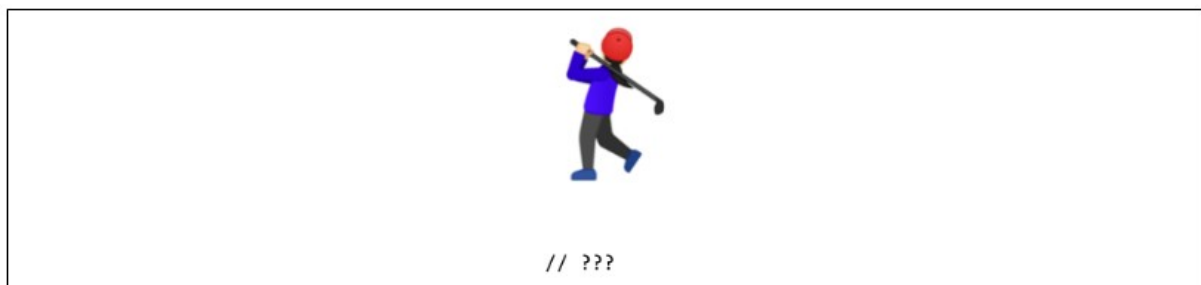


Figure 1-7. How do we write a golfing algorithm?

We're now stuck. How do we determine that someone is golfing using this methodology? The person might walk for a bit, stop, do some activity, walk for a bit more, stop, etc. But how can we tell this is golf? Our ability to detect this activity using traditional rules has hit a wall. But maybe there's a better way. which is machine learning.

From Programming to Learning

Let's look back at the diagram that we used to demonstrate what traditional programming is (Figure 1-8). Here we have rules that act on data and give us answers. In our activity detection scenario, the data was the speed at which the person was moving; from that we could write rules to detect their activity, be it walking, biking, or running. We hit a wall when it came to golfing, because we couldn't come up with rules to determine what that activity looks like. Now suppose we want to extend this to another popular fitness activity, biking.



Figure 1-9. Changing the axes to get machine learning

So what are the implications of this? Well, now instead of us trying to figure out what the rules are, we get lots of data about our scenario, we label that data, and the computer can figure out what the rules are that make one piece of data match a particular label and another piece of data match a different label. How would this work for our activity detection scenario? Well, we can look at all the sensors that give us data about this person. If they have a wearable that detects information such as heart rate, location, speed, etc.—and if we collect a lot of instances of this data while they're doing different activities—we end up with a scenario of having data that says “This is what walking looks like,” “This is what running looks like,” and so on.





			
0101001010100101010 1001010101001011101 0100101010010101001 0101001010100101010	1010100101001010101 0101010010010010001 0010011111010101111 1010100100111101011	1001010011111010101 1101010111010101110 1010101111010101011 1111110001111010101	1111111111010011101 0011111010111110101 01011101010101110 1010101010100111110
Label = WALKING	Label = RUNNING	Label = BIKING	Label = GOLFING

Figure 1-10. From coding to ML: gathering and labeling data

We will use some form representation. Now our job as programmers changes from figuring out the rules, to determining the activities,

The machine learning paradigm is one where you have data, that data is labeled, and you want to figure out the rules that match the data to the labels. The simplest possible scenario to show this in code is as follows. Consider these two sets of numbers:

```
X = -1, 0, 1, 2, 3, 4
Y = -3, -1, 1, 3, 5, 7
```

There's a relationship between the X and Y values (for example, if X is -1 then Y is -3, if X is 3 then Y is 5, and so on). Can you see it? After a few seconds you probably saw that the pattern here is $Y = 2X - 1$. How did you get that? Different people work it out in different ways, but I typically hear the observation that X increases by 1 in its sequence, and Y increases by 2; thus, $Y = 2X \pm \text{something}$. They then look at when $X = 0$ and see that $Y = -1$, so they figure that the answer could be $Y = 2X - 1$. Next they look at the other values and see that this hypothesis "fits," and the answer is $Y = 2X - 1$.

That's very similar to the machine learning process. Let's take a look at some TensorFlow code that you could write to have a neural network do this figuring out for you.

Getting Started with machine learning

```
In [17]: import tensorflow as tf
import numpy as np
from tensorflow import keras
```

```
In [18]: x_data = np.array([-1.0 , 0.0 , 1.0 , 2.0 , 3.0 , 4.0] , dtype = float)
y_data = np.array([-3.0 , -1.0 , 1.0 , 3.0 , 5.0 , 7.0] , dtype = float)
```

```
In [20]: model = keras.models.Sequential([
    keras.layers.Dense(units = 1 , input_shape = [1]),
    keras.layers.Dense(units = 10 , activation = 'relu'),
    keras.layers.Dense(units = 10 , activation = 'relu'),
    keras.layers.Dense(units = 1)
])
```

```
In [21]: model.compile(optimizer = 'sgd' , loss = 'mean_squared_error')
```

```
In [23]: history = model.fit(x_data , y_data , epochs = 500)
```

Epoch 1/500

We are using a Numpy array , because numpy helps us for data representation , specially data's that are in list.

```
In [23]: history = model.fit(x_data, y_data, epochs = 500)

Epoch 1/500
1/1 [=====] - 0s 4ms/step - loss: 13.8536
Epoch 2/500
1/1 [=====] - 0s 4ms/step - loss: 12.3690
Epoch 3/500
1/1 [=====] - 0s 3ms/step - loss: 10.7100
Epoch 4/500
1/1 [=====] - 0s 4ms/step - loss: 8.8118
Epoch 5/500
1/1 [=====] - 0s 5ms/step - loss: 6.7702
Epoch 6/500
1/1 [=====] - 0s 5ms/step - loss: 4.9435
Epoch 7/500
1/1 [=====] - 0s 8ms/step - loss: 3.7882
Epoch 8/500
1/1 [=====] - 0s 6ms/step - loss: 3.3122
Epoch 9/500
1/1 [=====] - 0s 4ms/step - loss: 3.1018
Epoch 10/500
1/1 [=====] - 0s 13ms/step - loss: 2.9349
Epoch 11/500
1/1 [=====] - 0s 13ms/step - loss: 2.7787
Epoch 12/500
1/1 [=====] - 0s 4ms/step - loss: 2.6623

In [26]: print(model.predict([10.0]))

[[18.935797]]
```

When using TensorFlow, you define your layers using Sequential . Inside the Sequential , you then specify what each layer looks like. The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs.

You then define what the layer looks like using the keras.layers API. There are lots of different layer types, but here we're using a Dense layer. "Dense" means a set of fully (or densely) connected neurons, where every neuron is connected to every neuron in the next layer. It's the most common form of layer type. Finally, when you specify the first layer in a neural network (in this case, it's our only layer), you have to tell it what the shape of the input data is. In this case our input data is our X, which is just a single value, so we specify that that's its shape.

Compile defines the loss function, the optimizer and the metrics. That's all. It has nothing to do with the weights and you can compile a model as many times as you want without causing any problem to pretrained weights. You need the "loss" for training too, so you can't train without compiling. And you can compile a model as many times as you want.

In a scenario such as this one, the computer has no idea what the relationship between X and Y is. So it will make a guess. Say for example it guesses that $Y = 10X + 10$. It then needs to measure how good or how bad that guess is. That's

the job of the loss function. It already knows the answers when X is $-1, 0, 1, 2, 3$, and 4 , so the loss function can compare these to the answers for the guessed relationship. If it guessed $Y = 10X + 10$, then when X is -1 , Y will be 0 . The correct answer there was -3 , so it's a bit off. But when X is 4 , the guessed answer is 50 , whereas the correct one is 7 . That's really far off.

Armed with this knowledge, the computer can then make another guess. That's the job of the optimizer. This is where the heavy calculus is used, but with TensorFlow, that can be hidden from you. You just pick the appropriate optimizer to use for different scenarios. In this case we picked one called `sgd`, which stands for stochastic gradient descent—a complex mathematical function that, when given the values, the previous guess, and the results of calculating the errors (or loss) on that guess, can then generate another one. Over time, its job is to minimize the loss, and by so doing bring the guessed formula closer and closer to the correct answer.

The learning process will then begin with the `model.fit` command. You can read this as “fit the X s to the Y s, and try it 500 times.” So, on the first try, the computer will guess the relationship (i.e., something like $Y = 10X + 10$), and measure how good or bad that guess was. It will then feed those results to the optimizer, which will generate another guess. This process will then be repeated, with the logic being that the loss (or error) will go down over time, and as a result the “guess” will get better and better.

We can see that over the first 10 epochs, the loss went from 3.2868 to 0.9682 . That is, after only 10 tries, the network was performing three times better than with its initial guess. We can now see the loss is 2.61×10^{-5} . The loss has gotten so small that the model has pretty much figured out that the relationship between the numbers is $Y = 2X - 1$ (The final hypothesis at the last epochs) The machine has learned the pattern between them.

What do you think the answer will be when we ask the model to predict Y when X is 10 ? You might instantly think 19 , but that's not correct. It will pick a value very close to 19 . There are several reasons for this. First of all, our loss wasn't 0 . It was still a very small amount, so we should expect any predicted answer to be off by a very small amount. Secondly, the neural network is trained on only a small amount of data—in this case only six pairs of (X, Y) values.

The model only has a single neuron in it, and that neuron learns a weight and a bias, so that $Y = WX + B$. This looks exactly like the relationship $Y = 2X - 1$ that we want, where we would want it to learn that $W = 2$ and $B = -1$. Given that the model was trained on only six items of data, the answer could never be

expected to be exactly these values, but something very close to them. Run the code for yourself to see what you get. I got 18.977888 when I ran it, but your answer may differ slightly because when the neural network is first initialized there's a random element: your initial guess will be slightly different from mine.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

l0 = Dense(units=1, input_shape=[1])
model = Sequential([l0])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
print("Here is what I learned: {}".format(l0.get_weights()))
```

The difference is that I created a variable called `l0` to hold the Dense layer. Then, after the network finishes learning, I can print out the values (or weights) that the layer learned.

In my case, the output was as follows:

Here is what I learned: `[array([[1.9967953]], dtype=float32), array([-0.9900647], dtype=float32)]`

Thus, the learned relationship between X and Y was $Y = 1.9967953X - 0.9900647$. This is pretty close to what we'd expect ($Y = 2X - 1$), and we could argue that it's even closer to reality, because we are assuming that the relationship will hold for other values.

Introduction to Computer Vision

The previous chapter introduced the basics of how machine learning works. You saw

how to get started with programming using neural networks to match data to labels, and from there how to infer the rules that can be used to distinguish items. A logical next step is to apply these concepts to computer vision, where we will have a model learn how to recognize content in pictures so it can “see” what’s in them. In this chapter you’ll work with a popular data-set of clothing items and build a model that can differentiate between them, thus “seeing” the difference between different types of clothing.

Recognizing Clothing Items

For our first example, let’s consider what it takes to recognize items of clothing in an image. Consider, for example, the items in the figure



Figure 2-1. Examples of clothing

There are a number of different clothing items here, and you can recognize them. You understand what is a shirt, or a coat, or a dress. But how would you explain this to somebody who has never seen clothing? How about a shoe? There are two shoes in this image, but how would you describe that to somebody? This is another area where the rules-based programming we spoke about in Chapter 1 can fall down (fall on solving complex problems). Sometimes it’s just infeasible to describe something with rules. Of course, computer vision is no exception. But consider how you learned to recognize all these items—by seeing lots of different examples, and gaining experience with how they’re used. Can we do the same with a computer? The answer is yes, but with limitations. Let’s take a look at a first example of how to teach a computer to recognize items of clothing, using a well-known dataset called Fashion MNIST.

The Data: Fashion MNIST

First, we need to load a dataset. In this chapter we will tackle Fashion MNIST, which is a drop-in replacement of MNIST (introduced in Chapter 3). **What does it mean by Drop in replacement ?** Drop-in replacement is a term used in computer science and other fields. It refers to the ability to replace one hardware (or software) component with another one without any other code or configuration changes being required and resulting in no negative impacts.

It has the exact same format as MNIST (70,000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST.

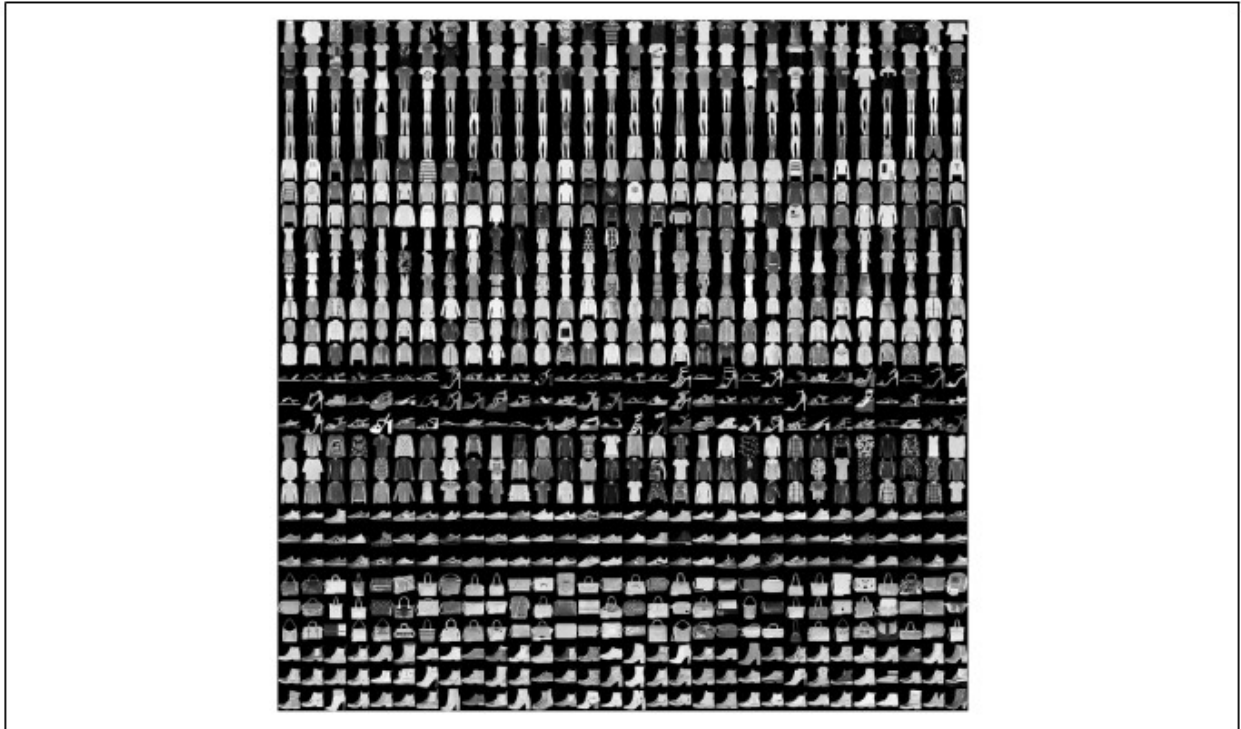


Figure 2-2. Exploring the Fashion MNIST dataset

It has a nice variety of clothing, including shirts, trousers, dresses, and lots of types of shoes. As you may notice, it's monochrome (different shades of the same color), so each picture consists of a certain number of pixels with values between 0 and 255. This makes the dataset simpler to manage.

You can see a closeup of a particular image from the dataset

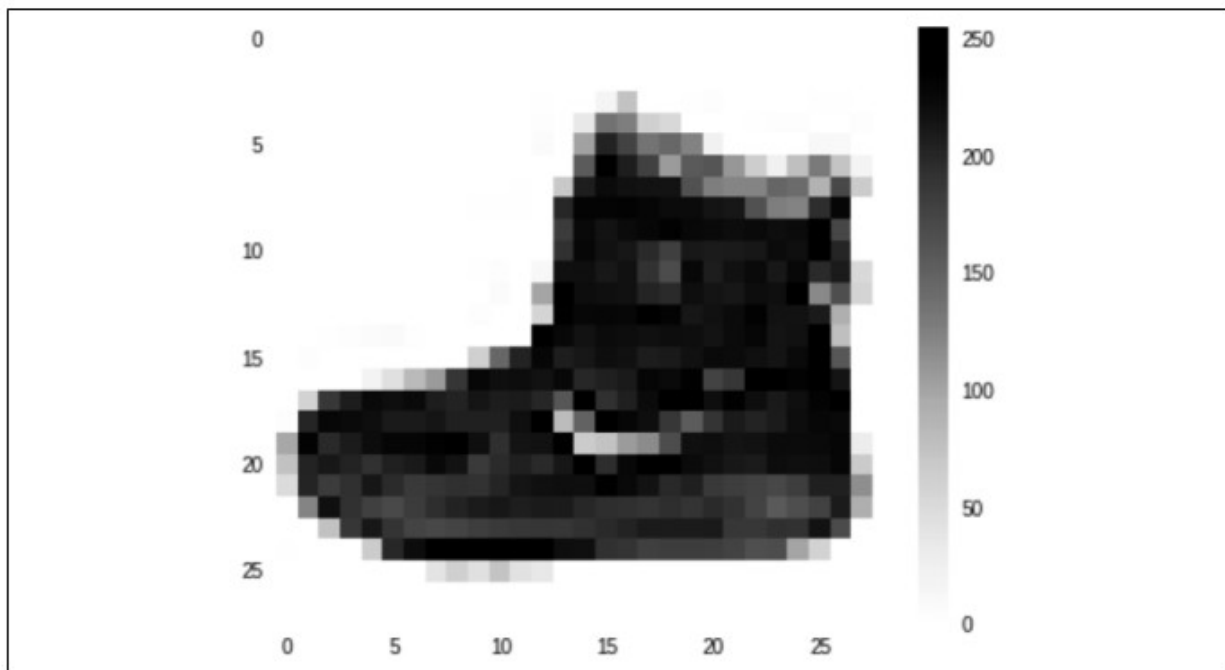


Figure 2-3. Closeup of an image in the Fashion MNIST dataset

Designing the Neural Network

Given that there are 10 labels, a random initialization should get the right answer about 10% of the time. From that, the loss function and optimizer can do their job epoch by epoch to tweak the internal parameters of each neuron to improve that 10%. And thus, over time, the computer will learn to “see” what makes a shoe a shoe or a dress a dress.

Using Keras to load the dataset Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and the California housing dataset.

Let’s load Fashion MNIST:

```
<> [35] import tensorflow as tf
import numpy as np
from tensorflow import keras

[36] data = keras.datasets.fashion_mnist
(X_train, y_train), (X_test, y_test) = data.load_data() # the Automatic split is done using Fashion MNIST API
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)

(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

Who handles the training and testing split, if we are not manually ? Keras has a number of built-in datasets that you can access with a single line of code like this. In this case you don’t have to handle downloading the 70,000 images—splitting them

into training and test sets, and so on—all it takes is one line of code. This methodology has been improved upon using an API called TensorFlow Datasets, but for the purposes of these early chapters, to reduce the number of new concepts you need to learn, we'll just use `tf.keras.datasets`.

But generally what if I want to import and load built in datasets from TensorFlow other than our current example Fashion MNIST ?

TensorFlow Dataset API :- TensorFlow API isn't the same as `tf.data`, TensorFlow datasets makes it easy to load datasets which are common, It's a high level wrapper intended to make it very easy to load commonly used datasets.

`tf.data` is helpful for custom datasets, helps us to build an input pipeline for datasets that perhaps we gathered on our own or scraped from the internet. Whereas TensorFlow datasets are a collection of datasets which are ready to use with TensorFlow.

First if we didn't install the tensorflow Dataset API, we need to do "pip install tensorflow_datasets".

There are so many Built in datasets from TensorFlow like

- 1) Boston Housing
- 2) Cifar 10
- 3) Fashion_MNIST
- 4) imdb :- IMDB sentiment classification dataset
- 5) mnist :- MNIST handwritten digit dataset

What if I want to import and load external files that are not in the built in datasets of Tensorflow ?

Fashion MNIST is designed to have 60,000 training images and 10,000 test images. So, the return from `data.load_data` will give you an array of 60,000 28×28 -pixel arrays called `training_images`, and an array of 60,000 values (0–9) called `training_labels`. Similarly, the `test_images` array will contain 10,000 28×28 -pixel arrays, and the `test_labels` array will contain 10,000 values between 0 and 9. **How can I maintain the proportion of the training, development and test set for other projects that have non built in datasets ?**

So how does the neural network learn, Our job will be to fit the training images to the training labels in a similar manner to how we fit Y to X but the best advantage of **neural networks is they can learn non linearity.**

We'll hold back the test images and test labels so that the network does not see them while training. These can be used to test the efficacy of the network with hitherto unseen data.

These is the basic difference When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28×28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0).

Note that the dataset is already split into a training set and a test set, but there is no validation set, so we'll create one now. Additionally, since we are going to train the neural network using Gradient Descent, we must scale the input features. For simplicity, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0 (this also converts them to floats). [How to normalize data generally , for both two dimensional data or one dimensional or for any dimension , which in turn helps us for fast convergence to the optimal position or to the one with very much less cost ?](#)

One common pre-processing step in machine learning is to center and standardize your dataset, meaning that you subtract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

```
✓ [37] # since we will use Gradient Descent we need to normalize the data
0s X_train = X_train/255
    y_train = y_train/255

✓ [38] # Example of a python syntax
0s a = 4
    b = 5
    c , d = a , b
    print("The value of c is = {}".format(c))
    print("The value of d is = {}".format(d))

The value of c is = 4
The value of d is = 5

✓ [39] # we need to create a validation dataset for the purpose of Hyper Tuning
0s X_valid , X_train = X_train[0:5000] , X_train[5000:]
    y_valid , y_train = y_train[0:5000] , y_train[5000:]
    print(X_valid.shape , X_train.shape)
    print(y_valid.shape , y_train.shape)

(5000, 28, 28) (55000, 28, 28)
(5000,) (55000,)
```

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents a coat:

```
>>> class_names[y_train[0]]
'Coat'
```

Figure 10-11 shows some samples from the Fashion MNIST dataset.



Figure 10-11. Samples from Fashion MNIST

Creating the model using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

```
[21] # now we can create the model
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape = [28 , 28]),
    keras.layers.Dense(units = 300 , activation = 'relu'),
    keras.layers.Dense(units = 300 , activation = 'relu'),
    keras.layers.Dense(units = 10 , activation = 'softmax')
])
```

```
[22] # another way of creating a model
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape = [28 , 28]))
model.add(keras.layers.Dense(units = 300 , activation = 'relu'))
model.add(keras.layers.Dense(units = 300 , activation = 'relu'))
model.add(keras.layers.Dense(units = 10 , activation = 'softmax'))
```

```
[23] # The another another way of creating a model
import tensorflow as tf
import numpy as np
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
model = Sequential([
    Flatten(input_shape = [28 , 28]),
    Dense(units = 300 , activation = keras.activations.relu),
    Dense(units = 300 , activation = keras.activations.relu),
    Dense(units = 10 , activation = keras.activations.softmax)
])
```


The first line creates a Sequential model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially. This is called the Sequential API. Next, we build the first layer and add it to the model. It is a Flatten layer whose role is to convert each input image into a 1D array: This layer does not have any parameters; it is just there to do some simple pre-processing. Since it is the first layer in the model, you should specify the `input_shape`, which doesn't include the batch size (training examples), only the shape of the instances, suppose when you have an input of 5 units, you got an input shape of `(None,5)`. But you actually say only `(5,)` to your model, because the `None` part is the batch size, which will only appear when training. Alternatively, you could add a `keras.layers.InputLayer` as the first layer, setting `input_shape=[28,28]`. Next we add a Dense hidden layer with 300 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). Then we add a second Dense hidden layer with 300 neurons, also using the ReLU activation function. Finally, we add a Dense output layer with 10 neurons (one per class), using the softmax activation function (because the classes are mutually exclusive).

We're asking for 300 neurons to have their internal parameters randomly initialized. Often the question I'll get asked at this point is "Why 300?" This is entirely arbitrary—there's no fixed rule for the number of neurons to use. As you design the layers you want to pick the appropriate number of values to enable your model to actually learn. More neurons means it will run more slowly, as it has to learn more parameters. More neurons could also lead to a network that is great at recognizing the training data, but not so good at recognizing data that it hasn't previously seen (this is known as overfitting and we'll discuss it later in this chapter). On the other hand, fewer neurons means that the model might not have sufficient parameters to learn.

Specifying `activation = "relu"` is equivalent to specifying `activation = keras.activations.relu`

Using Code Examples from keras.io

Code examples documented on keras.io will work fine with tf.keras, but you need to change the imports. For example, consider this keras.io code:

```
from keras.layers import Dense
output_layer = Dense(10)
```

You must change the imports like this:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Or simply use full paths, if you prefer:

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

This approach is more verbose, but I use it in this book so you can easily see which packages to use, and to avoid confusion between standard classes and custom classes. In production code, I prefer the previous approach. Many people also use `from tensorflow.keras import layers` followed by `layers.Dense(10)`.

The model's `summary()` method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters.

```
✓ [29] model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
dense_7 (Dense)	(None, 300)	235500
dense_8 (Dense)	(None, 300)	90300
dense_9 (Dense)	(None, 10)	3010

```
=====
Total params: 328,810
Trainable params: 328,810
Non-trainable params: 0
=====
```

Note that Dense layers often have a lot of parameters. For example, the first hidden layer has 784×300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of over fitting, especially when you do not have a lot of training data. You can easily get a model's list of layers, to

fetch a layer by its index, or you can fetch it by name , All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods. For a Dense layer, this includes both the connection weights and the bias terms:

```
[31] # we can access the layers,the weights in the layer using thier index or using their names
model.layers

[<keras.layers.core.flatten.Flatten at 0x7f22bee981d0>,
 <keras.layers.core.dense.Dense at 0x7f22b8191450>,
 <keras.layers.core.dense.Dense at 0x7f22b803ec10>,
 <keras.layers.core.dense.Dense at 0x7f22b80e0e90>]

[32] input_layer = model.layers[0]
input_layer.name

'flatten_2'

[38] model.get_layer('flatten_2') is input_layer

True

[33] hidden1 = model.layers[1]
print(hidden1)

<keras.layers.core.dense.Dense object at 0x7f22b8191450>

# we can also access the weights and the biases of the layer one
# but it will displayed the last values after they are back propagated
weights , biases = hidden1.get_weights()
print(weights)
print(biases)
```

Compiling The Model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can specify a list of extra metrics to compute during training and evaluation. Using `loss = "sparse_categorical_crossentropy"` is equivalent to using `loss=keras.losses.sparse_categorical_crossentropy`. Similarly, specifying `optimizer="sgd"` is equivalent to specifying `optimizer=keras.optimizers.SGD()` , and `metrics=["accuracy"]` is equivalent to `metrics=[keras.metrics.sparse_categorical_accuracy]` (when using this loss).

```
# compiling the model
model.compile(optimizer = keras.optimizers.SGD() , loss = keras.losses.sparse_categorical_crossentropy , metrics = [keras.metrics.sparse_categorical_accuracy])

[26] # another way of compiling the model
model.compile(optimizer= 'sgd' , loss = 'sparse_categorical_crossentropy' , metrics = ["accuracy"])
```

The loss function in this case is called sparse categorical cross entropy, and it's one of

the arsenal of loss functions that are built into TensorFlow. Again, choosing which loss function to use is an art in itself.

How to Choose Loss Functions When Training Deep Learning Neural Networks

Regarding the optimizer, "sgd" means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the back-propagation algorithm described earlier.

When using the SGD optimizer, it is important to tune the learning rate. So, you will generally want to use `optimizer=keras.optimizers.SGD(lr=???)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to `lr=0.01`.

You might notice that a new line specifying the metrics we want to report is also present in this code. Here, we want to report back on the accuracy of the network as we're training. The simple example in Chapter 1 just reported on the loss, and we interpreted that the network was learning by looking at how the loss was reduced. In this case, it's more useful to us to see how the network is learning by looking at the accuracy—where it will return how often it correctly matched the input pixels to the output label.

Training and evaluating the model

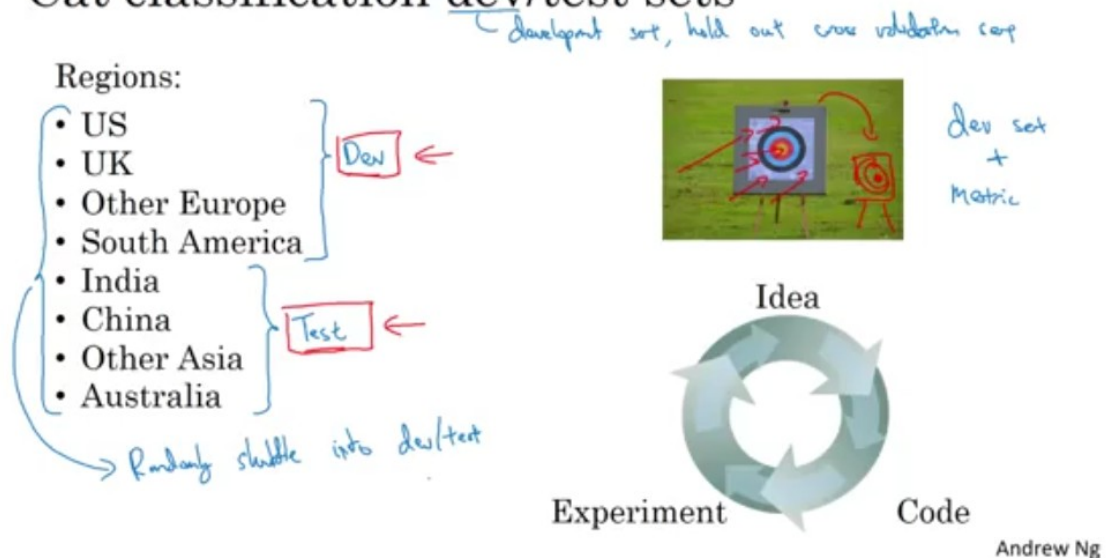
```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218      - accuracy: 0.7660
                                   - val_loss: 0.4973 - val_accuracy: 0.8366

Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840      - accuracy: 0.8327
                                   - val_loss: 0.4456 - val_accuracy: 0.8480

[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252      - accuracy: 0.9192
                                   - val_loss: 0.2999 - val_accuracy: 0.8926
```

We pass it the input features (X_{train}) and the target classes (y_{train}), as well as the number of epochs to train (or else it would default to just one epochs which is the random initialization , which will be accurate by 10 %), which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional). Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs. If the performance on the training set is much better than on the validation set, your model is probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set). what does it mean by a **data mismatch problem** ? It's a problem arises when there is training and validation data comes from different distribution , they must be from the same distribution , this will happen by randomly shuffle the data.

Cat classification dev/test sets



Suppose let's see the above image , we are doing a cat classifier, but the data for validation set (which used to tune our hyper parameters) and the data for the test set (which used to generalization for unseen data's) but we were collecting from different distribution , the dev set data from the Eastern part of the world and the test set data from the western part of the world , so our algorithm will learn the geography effect , these Data mis match problem will be solved by random shuffling.

And that's it! The neural network is trained. 15 At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached 89.26% after 30 epochs. That's not too far from the training accuracy, so there does not seem to be much overfitting going on.

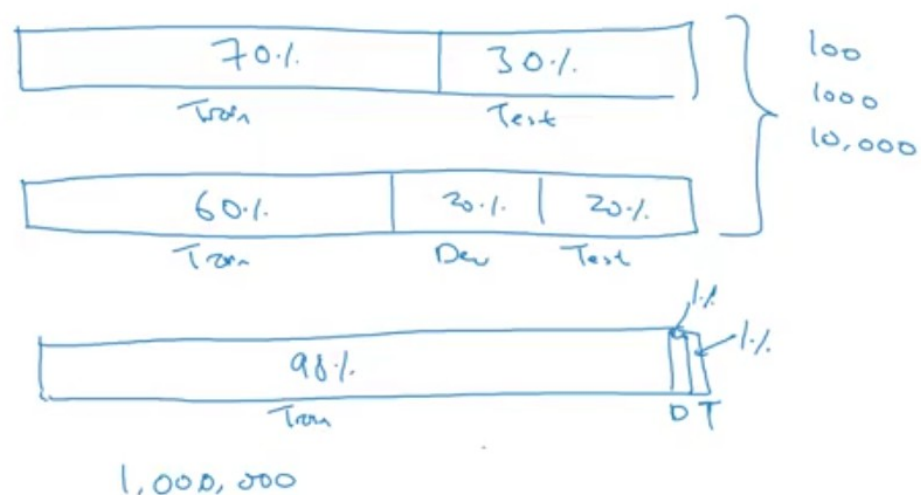
Instead of passing a validation set using the `validation_data` argument, you could set `validation_split` to the ratio of the training set that you want Keras to use for validation. For example, `validation_split=0.1` tells Keras to use the last 10% of the data (before shuffling) for validation. **What to do to shuffle the data before splitting the data set in to training and validation ?**

```
-> model.fit(X_train , y_train , epochs = 50 , validation_split = 0.1 , shuffle = True)
```

But how to decide the proportion of the validation set from the training set ?

Old ways of splitting were 70 %(training) by 30 %(testing) or 60 % , 20 % , 20 % which is training , validation , testing but these was before the big data era , we are now in the big data generation, if we have 1 million datasets giving 20% for validation of testing set will be very much a problem so if we are working on a big datasets suppose if our datasets are 1 million give 98 % for training , 1% for validation and 1% for a test set.

Old way of splitting data



The `fit()` method returns a History object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). **What is history.params ?**

Why is the test set accuracy lower than the training set ? You're probably wondering why the accuracy is lower for the test data than it is for the training data. This is very commonly seen, and when you think about it, it makes sense: the neural network only really knows how to match the inputs it has been trained on with the outputs for those values. Our hope is that, given enough data, it will be able to generalize from the examples it has seen, "learning" what a shoe or a dress looks like. But there will always be examples of items that it hasn't seen that are sufficiently different from

what it has to confuse it. For example, if you grew up only ever seeing sneakers, and that's what a shoe looks like to you, when you first see a high heel you might be a little confused. From your experience, it's probably a shoe, but you don't know for sure. This is a similar concept.

Learning Curves In Neural Networks

- What are learning curves
- Why are we studying learning curves
- How to use learn and understand learning curves
- How to plot learning curves
- Should we need to compare the training and test accuracies to identify over-fitting or should we need to compare the training and validation accuracies to identify over-fitting problem ?
- What is the need for the validation set
- Is the back-propagation also useful for the validation set
- Why are we iterating the validation set at every epochs
- Basically our validation set is used for selecting the good hypothesis , so in neural networks in each of the epochs we are creating a hypothesis which decrease the loss by updating the weights and biases , so our validation set will check the evolution of the training accuracy to select the good hypothesis. **Question :- why are we need another split-ted dataset for choosing the hypothesis , shouldn't we simply peak the one with the lowest cost or loss ?** No , because we need our estimate to be judged by examples that haven't seen before , because just because our loss is very small doesn't mean it's a good hypothesis. **Question :- How can we select the good hypothesis using the validation set ?**
- How does the validation set enables us to tune the hyper-parameter's ?
- How to recover if our validation accuracy is low , which is basically we will train it for longer time (by increasing the number of epochs)? , which makes the training data more accurate so it will generalize better for unseen data's(and so we will select a good model) , so if we can't satisfy good hypothesis by increasing the number of epochs (training it for longer time) we will try tuning so many different hyper-parameters and fed them to the training data , train them with the new hyper-parameters and now see the evolution of the training accuracy at each epochs then select the one with the lesser validation loss , because having higher accuracy doesn't give us a good validation loss because it overfit's the data , so in order to solve these problem we can use TensorFlow callbacks for the early stopping purpose.

What is a Validation Dataset by the Experts?

Importantly, Russell and Norvig comment that the training dataset used to fit the model can be further split into a training set and a validation set, and that it is this subset of the training dataset, called the validation set, that can be used to get an early estimate of the skill of the model.

If the test set is locked away, but you still want to measure performance on unseen data as a way of selecting a good hypothesis, then divide the available data (without the test set) into a training set and a validation set.

- Training set:- A set of examples used for learning, that is to fit the parameters of the classifier.
- Validation set:- A set of examples used to tune the parameters (hyper parameters) of a classifier, for example to choose the number of hidden units in a neural network also they are used for model selection.
- Test set:- A set of examples used only to assess the performance of a fully-specified classifier.

- Is the back-propagation also useful for the validation set

Back-propagation only works during training the model on a data-set. The parameters of the model are *learned* through Back-prop. During validation, you assess which *hyper-parameters* work best for your model *while using the parameters learned during Back-prop*, i.e., learning rate, momentum (if you are training neural nets), filter size (for CNNs) etc. You save the model with the hyper-parameters which give you best accuracy/least error on the validation set. You run your model with the learned parameters (from Back-propagation) and best hyper-parameters (from validation) **once** on the Test set and report the accuracy. You **never** learn anything, be it parameters or hyper-parameters on the Test set.

Note:- Back-propagation is only used during Training.

Why do we iterate over the validation set?

Do we calculate error in validation to get weight updates? No. We do all our updating only using the training set. Validation is only there to see how our trained model generalizes outside of training data. Think of it as a test before the test you run with test set.

In general, for static/finite data-set settings, it is very typical for one to check model validation performance after each epoch (or training iteration, depending on the choice of terminology). An epoch, in terms of neural-based training, is one full pass through the data-set. This allows you to observe the evolution of the model's generalization ability at "reasonable" checkpoints without spending too much computation on calculating validation error (or loss), since you will want to dedicate most of the computation towards parameter estimation/updating.

How does the validation set enables us to select from different models , in the case of neural networks the validation set helps us to select the “good” hypothesis by checking the training evolutions , so how does the validation set even helps us to select the good hypothesis even from the different iteration (epochs) of the neural network ?

$\Rightarrow d = \text{degree of polynomial}$

Model selection

$d=1$ 1. $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x \rightarrow \Theta^{(1)} \rightarrow J_{\text{test}}(\Theta^{(1)})$

$d=2$ 2. $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \rightarrow \Theta^{(2)} \rightarrow J_{\text{test}}(\Theta^{(2)})$

$d=3$ 3. $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_3 x^3 \rightarrow \Theta^{(3)} \rightarrow J_{\text{test}}(\Theta^{(3)})$

\vdots

$d=10$ 10. $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x^{10} \rightarrow \Theta^{(10)} \rightarrow J_{\text{test}}(\Theta^{(10)})$

Choose $\theta_0 + \dots + \theta_5 x^5 \leftarrow$

How well does the model generalize? Report test set error $J_{\text{test}}(\theta^{(5)})$.

Problem: $J_{\text{test}}(\theta^{(5)})$ is likely to be an optimistic estimate of generalization error. I.e. our extra parameter ($d = \text{degree of polynomial}$) is fit to test set.

Andrew Ng

On the above image , if we are using the same test set data for both selecting the model and for evaluating the model , then our final result will be an optimistic estimate of the generalization error.

Evaluating your hypothesis

Dataset:

Size	Price	
2104	400	60% Training set
1600	330	
2400	369	
1416	232	
3000	540	
1985	300	
1534	315	20% Cross validation set (CV)
1427	199	
1380	212	20% test set
1494	243	

$(x^{(1)}, y^{(1)})$
 $(x^{(2)}, y^{(2)})$
 \vdots
 $(x^{(m)}, y^{(m)})$

$(x_{cv}^{(1)}, y_{cv}^{(1)})$
 $(x_{cv}^{(2)}, y_{cv}^{(2)})$
 \vdots
 $(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$

$(x_{test}^{(1)}, y_{test}^{(1)})$
 $(x_{test}^{(2)}, y_{test}^{(2)})$
 \vdots
 $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

$m_{cv} = \text{no. of cv examples}$
 $(x_{cv}^{(i)}, y_{cv}^{(i)})$
 m_{test}

Andrew Ng

So we need to split our entire data-set in to three parts which are training , Cross validation or hold out validation and the test set (the proportion is dependent to the scale of the data)

Train/validation/test error

Training error:

$$\rightarrow J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad J(\theta)$$

Cross Validation error:

$$\rightarrow J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_{\theta}(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_{\theta}(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Model selection

$$\begin{array}{llll}
 d=1 & 1. & h_{\theta}(x) = \theta_0 + \theta_1 x & \xrightarrow{\min_{\theta} J(\theta)} \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)}) \\
 d=2 & 2. & h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 & \xrightarrow{\quad} \theta^{(2)} \rightarrow J_{cv}(\theta^{(2)}) \\
 d=3 & 3. & h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_3 x^3 & \xrightarrow{\quad} \theta^{(3)} \rightarrow J_{cv}(\theta^{(3)}) \\
 \vdots & \vdots & \vdots & \vdots \\
 d=10 & 10. & h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x^{10} & \xrightarrow{\quad} \theta^{(10)} \rightarrow J_{cv}(\theta^{(10)})
 \end{array}$$

$\underline{d=4} \xrightarrow{\quad} \uparrow$

Pick $\theta_0 + \theta_1 x_1 + \dots + \theta_4 x^4 \leftarrow$

Estimate generalization error for test set $\underline{J_{test}(\theta^{(4)})} \leftarrow$

So we will train (with back-propagation) the data using our training data-set and using that weight and biases that we get at the end of that epoch (reasonable checkpoint) we will try to calculate the validation error , and then we will choose a certain model (good hypothesis) that really have a least validation error. If the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the fit() method again, since Keras just continues training where it left off .

If you are not satisfied with the performance of your model after you trained it for a longer time, you should go back and tune the hyper-parameters. The first one to check is the learning rate. If that doesn't help, try another optimizer (and always retune the learning rate after changing any hyper-parameter). If the performance is still not great, then try tuning model hyper-parameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer. We will get back to hyper-parameter tuning at the end of this chapter. Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method.

Should we need to compare the training and test accuracies to identify over-fitting or should we need to compare the training and validation accuracies to identify over-fitting problem ?

You should compare the training and test accuracies to identify over-fitting. A training accuracy that is subjectively far higher than test accuracy indicates over-fitting.

More on validation set

Validation set shows up in two general cases: (1) building a model, and (2) selecting between multiple models,

1. Two examples for building a model: we (a) stop training a neural network, or (b) stop pruning a decision tree when accuracy of model on validation set starts to decrease. Then, we test the final model on a held-out set, to get the test accuracy.

2. Two examples for selecting between multiple models:

We do K-fold CV on one neural network with 3 layers, and one with 5 layers (to get K models for each), then we select the Neural networks with the highest validation accuracy averaged over K models; suppose the 5 layer Neural networks. Finally, we train the 5 layer Neural networks on a 80% train, 20% validation split of combined K folds, and then test it on a held out set to get the test accuracy.

How does the validation set enables us to tune the hyper-parameter's ?

Exploring the Model Output

Now that the model has been trained, and we have a good gage of its accuracy using the test set

```
[ ] # We can Evaluate the model
    model.evaluate(X_test , y_test) # the model is overfitted

313/313 [=====] - 1s 3ms/step - loss: 62.8934 - accuracy: 0.8590
[62.893436431884766, 0.859000027179718]

[ ] # Exploring the model's output
    y_pred = model.predict(X_test[:2])
    print("The prediction value is = {}".format(y_pred))

The prediction value is = [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]

[ ] # checking the predicted one
    real_value = y_test[:2]
    print("The real value is = {}".format(real_value))

The real value is = [9 2]
```

It is common to get slightly lower performance on the test set than on the validation set, because the hyper-parameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyper-parameter tuning (means we are just do good on the default ones , putting the learning rate as 0.01 and the optimizer as sgd), so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyper-parameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the model to make predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first three instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 96%, the probability of class 5 (sandal) is 3%, the probability of class 7 (sneaker) is 1%, and the probabilities of the other classes are negligible. In other words, it “believes” the first image is footwear, most likely ankle boots but possibly sandals or sneakers. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `predict_classes()` method instead:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Here, the classifier actually classified all three images correctly (these images are shown in [Figure 10-13](#)):

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

Building a Regression Multi Layer Perceptron Using the Sequential API

Building Complex Model Using Functional API

- 1) What is functional API and what does the name functional refers to ?
- 2) What is the problem with the Sequential API
- 3) What does it mean when it says a complex model
- 4) How does the functional API helps us to build these complex models
- 5) implementing functional API

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that have multiple inputs or outputs. [why do we need to have multiple inputs and outputs ?](#)

The functional API in Keras is an alternate way of creating models that offers a lot more flexibility, including creating more complex models. [we can also create the normal Sequential API using the functional API.](#)

After completing this tutorial, we will know:

- The difference between the Sequential and Functional API's.
- How to define simple Multilayer Perceptron, Convolutional Neural Network using the functional API.
- How to define more complex models with multiple inputs and outputs

Keras Sequential Models

The Sequential model API is a way of creating deep learning models where an instance of the Sequential class is created and model layers are created and added to it. The Sequential model API is great for developing deep learning models in most situations, but it also has some limitations. For example, it is not straightforward to define models that may have multiple different input sources, produce multiple output destinations or models that re-use layers.

```
✓ [21] # now we can create the model
Os
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape = [28 , 28]),
    keras.layers.Dense(units = 300 , activation = 'relu'),
    keras.layers.Dense(units = 300 , activation = 'relu'),
    keras.layers.Dense(units = 10 , activation = 'softmax')
])
```

```
✓ [22] # another way of creating a model
Os
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape = [28 , 28]))
model.add(keras.layers.Dense(units = 300 , activation = 'relu'))
model.add(keras.layers.Dense(units = 300 , activation = 'relu'))
model.add(keras.layers.Dense(units = 10 , activation = 'softmax'))
```

```
✓ [23] # The another another way of creating a model
Os
import tensorflow as tf
import numpy as np
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
model = Sequential([
    Flatten(input_shape = [28 , 28]),
    Dense(units = 300 , activation = keras.activations.relu),
    Dense(units = 300 , activation = keras.activations.relu),
    Dense(units = 10 , activation = keras.activations.softmax)
])
```

Keras Functional Models

The Keras functional API provides a more flexible (flexible means it must not be sequential) way for defining models. It specifically allows you to define multiple input or output models. Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model.

One example of a non sequential neural network is a Wide & Deep neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Chen et al. [16]. It connects all or part of the inputs directly to the output layer (End to End deep learning, it's a process of mapping the input x directly to y , but we will need a lot of data to do this, suppose I am doing a machine translator which translates English words to French, in this case I may not need to learn the deep patterns by passing through the deep path because there are so many labeled French data's for a corresponding English word, so we don't need to learn any deep patterns). This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). Contrast, a regular MLP forces all the data to flow through the full stack of layers, thus, **simple patterns in the data may end up being distorted by this sequence of transformations.**

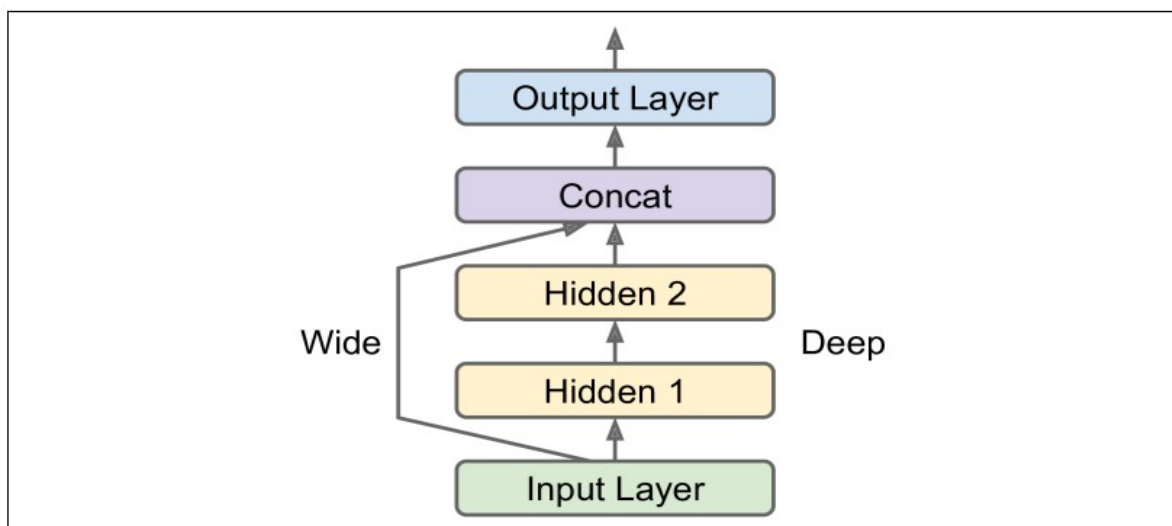


Figure 10-14. Wide & Deep neural network

Now, let's build a neural network using Functional API :-

```
input_ = keras.layers.Input(shape=[2,]) # one dimensional but 2 refers the number of your dimen:
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Defining Input

Unlike the Sequential model, you must create and define a standalone Input layer that specifies the shape of input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality(the features) of the input data. When input data is one-dimensional, such as for a multilayer Perceptron, the shape must explicitly leave room for the shape of the mini-batch size used when splitting the data when training the network. Therefore, the shape tuple is always defined with a hanging last dimension when the input is one-dimensional (2,).

Connecting Layers

The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket notation is used, such that after the layer is created, the layer from which the input to the current layer comes from is specified. Let's make this clear with a short example. It is this way of connecting layers piece by piece that gives the functional API its flexibility. and This is why this is called the Functional API , because we are calling the order using functions , which in turn creates flexibility.

Concatenating the layers

Next, we create a Concatenate layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer. You may prefer the `keras.layers.concatenate()` function, which creates a Concatenate layer and immediately calls it with the given inputs. Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.

Creating the Model

After creating all of your model layers and connecting them together and concatenating it , you must define the model. As with the Sequential API, the model is the thing you can summarize, fit, evaluate, and use to make predictions. Keras provides a model class that you can use to create a model from your created layers. It requires that you only specify the input and output layers.

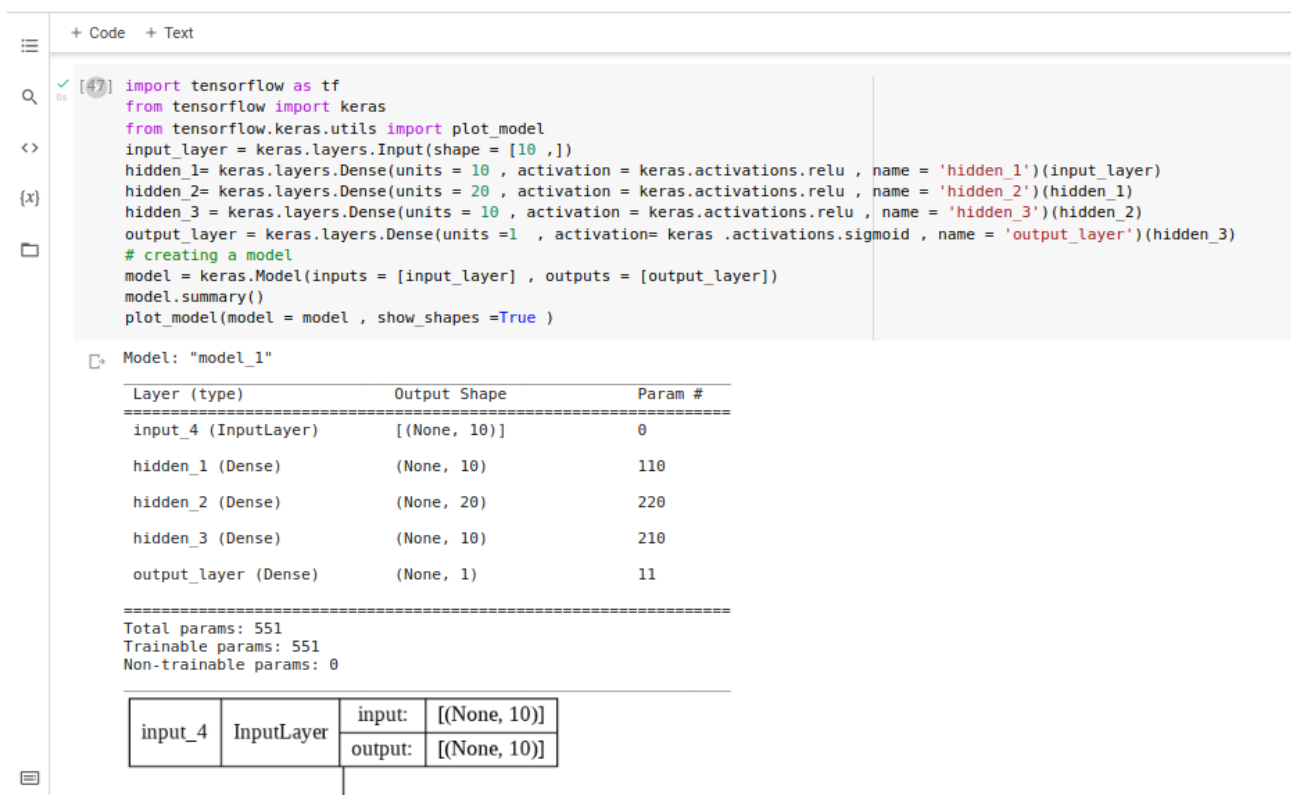
Now that we know all of the key pieces of the Keras functional API, let's work through defining a suite of different models and build up some practice with it.

Standard Network Models

When getting started with the functional API, it is a good idea to see how some standard neural network models are defined. In this section, we will look at defining a simple multilayer Perceptron, convolutional neural network.

Multilayer Perceptron

We define a multilayer Perceptron model for binary classification. The model has 10 inputs, 3 hidden layers with 10, 20, and 10 neurons, and an output layer with 1 output. Rectified linear activation functions are used in each hidden layer and a sigmoid activation function is used in the output layer, for binary classification.



```
[47] import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model
input_layer = keras.layers.Input(shape = [10 ,])
hidden_1= keras.layers.Dense(units = 10 , activation = keras.activations.relu , name = 'hidden_1')(input_layer)
hidden_2= keras.layers.Dense(units = 20 , activation = keras.activations.relu , name = 'hidden_2')(hidden_1)
hidden_3 = keras.layers.Dense(units = 10 , activation = keras.activations.relu , name = 'hidden_3')(hidden_2)
output_layer = keras.layers.Dense(units = 1 , activation= keras .activations.sigmoid , name = 'output_layer')(hidden_3)
# creating a model
model = keras.Model(inputs = [input_layer] , outputs = [output_layer])
model.summary()
plot_model(model = model , show_shapes =True )
```

Model: "model_1"

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 10)]	0
hidden_1 (Dense)	(None, 10)	110
hidden_2 (Dense)	(None, 20)	220
hidden_3 (Dense)	(None, 10)	210
output_layer (Dense)	(None, 1)	11

=====
Total params: 551
Trainable params: 551
Non-trainable params: 0

input_4	InputLayer	input:	[(None, 10)]
		output:	[(None, 10)]

Convolutional Neural Network

we will define a convolutional neural network for image classification. The model receives black and white 64×64 images as input, then has a sequence of two convolutional and [pooling layers](#) as feature extractors, followed by a fully connected layer to interpret the features and an output layer with a sigmoid activation for two-class predictions.

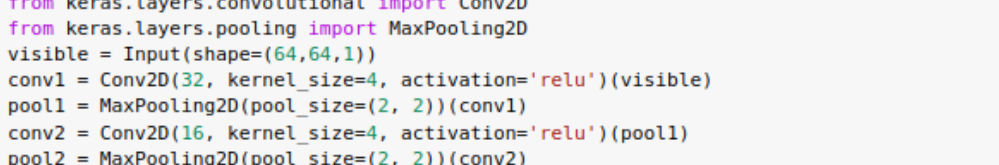

```

+ Code + Text

# Building a convolutional Neural Network using Functional API

[60] from tensorflow.keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
visible = Input(shape=(64,64,1))
conv1 = Conv2D(32, kernel_size=4, activation='relu')(visible)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(16, kernel_size=4, activation='relu')(pool1)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
flat = Flatten()(pool2)
hidden1 = Dense(10, activation='relu')(flat)
output = Dense(1, activation='sigmoid')(hidden1)
model = Model(inputs=visible, outputs=output)
# plot graph
plot_model(model, show_shapes=True)

```



Shared Layers Model

Shared the Input Layer

we define multiple convolutional layers with differently sized kernels to interpret an image input. The model takes black and white images with the size 64×64 pixels. There are two CNN feature extraction sub models that share this input; the first has a kernel size of 4 and the second a kernel size of 8. The outputs from these feature extraction sub models are flattened into vectors and concatenated into one long vector and passed on to a fully connected layer for interpretation before a final output layer makes a binary classification. This might be really used for Inception Architecture (personal opinion)

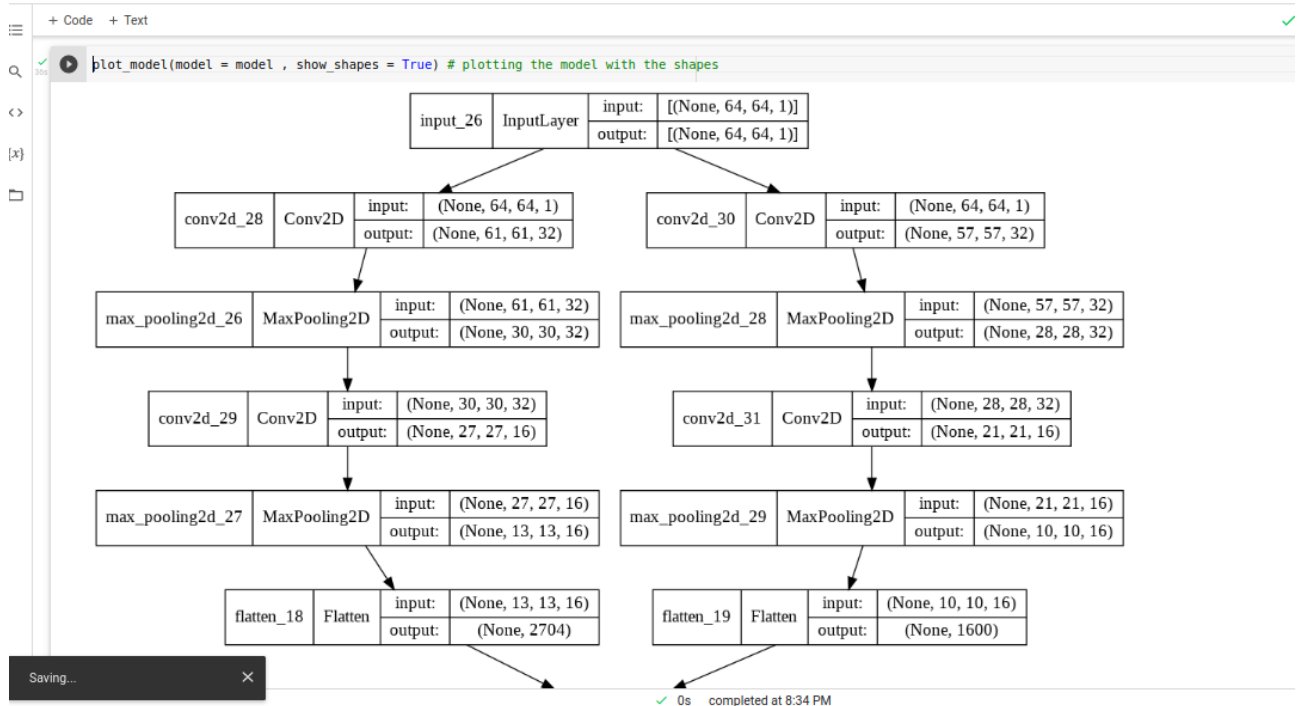
✓ [126] # sharing layers using functional API

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling1D
input_layer = Input(shape = (64 , 64 , 1))
# creating the first convolutional network
conv1 = Conv2D(32 , kernel_size=4, activation = keras.activations.relu)(input_layer)
pool1 = MaxPooling2D(pool_size=(2 , 2))(conv1)
conv2 = Conv2D(16 , kernel_size=4 , activation = keras.activations.relu)(pool1)
pool2 = MaxPooling2D(pool_size = (2 , 2))(conv2)
flat1 = Flatten()(pool2)
# creating the second convolutional network
conv11 = Conv2D(32 , kernel_size = 8 , activation = keras.activations.relu)(input_layer)
pool11 = MaxPooling2D(pool_size=(2 , 2))(conv11)
conv22 = Conv2D(16 , kernel_size = 8 , activation = keras.activations.relu)(pool11)
pool22 = MaxPooling2D(pool_size=(2 , 2))(conv22)
flat2 = Flatten()(pool22)
# concatenating both of the convolutional networks
concat = keras.layers.concatenate([flat1 , flat2])
hidden1 = keras.layers.Dense(units = 10, activation = keras.activations.relu , name = 'first_hidden')(concat)
hidden2 = keras.layers.Dense(units = 10, activation = keras.activations.relu , name = 'second_hidden')(hidden1)
output_layer = keras.layers.Dense(units = 1 , activation = keras.activations.sigmoid , name = 'output_layer')(hidden2)
# creating the model using the inputs and outputs
model = Model(inputs = [input_layer] , outputs = [output_layer])
print(model.summary())
```

<IPython.core.display.Image object>
Model: "model_10"

Output Shape Param # Connected to
✓ 0s completed at 8:34 PM

we can now plot the model



Multiple Input and Output Models

The functional API can also be used to develop more complex models with multiple inputs, possibly with different modalities. It can also be used to develop models that produce multiple outputs. **Why do we need multiple inputs and multiple outputs ?** The reason for the multiple inputs is because some problems might be solved through the end to end deep learning process , without passing thorough the deep path , what is the problem if those pass through the deep path ? because this simple pattern will be lost while they are passed through.

if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path (see the Figure)? In this case, one solution is to use multiple inputs. For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path (features 2 to 7):

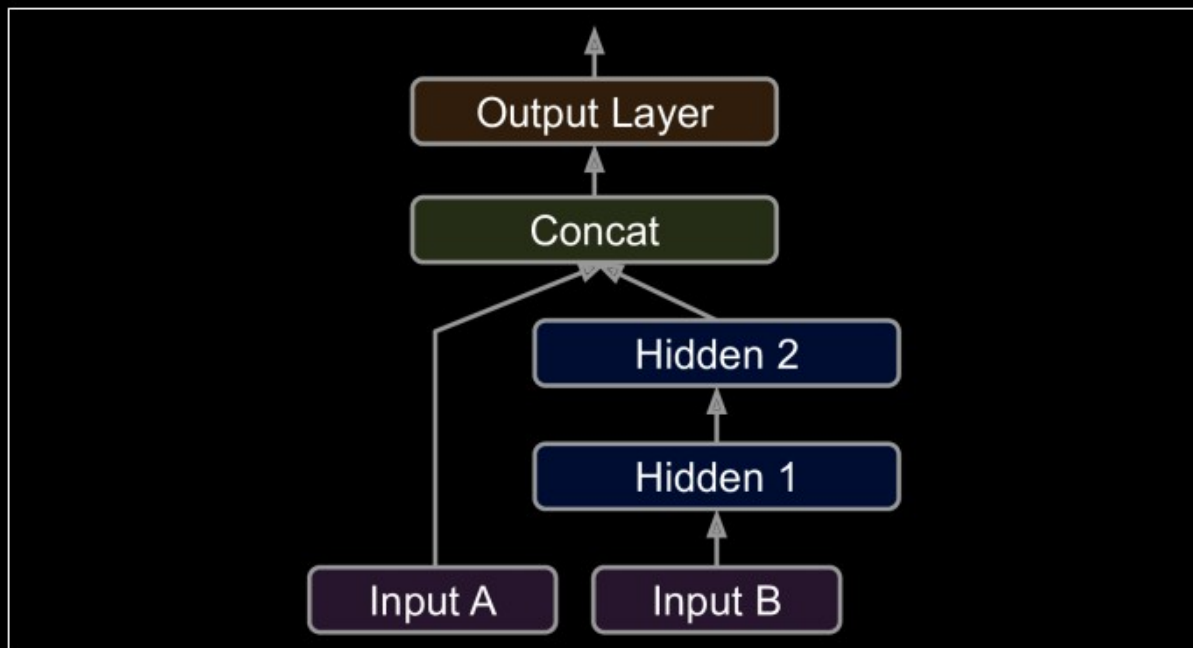


Figure 10-15. Handling multiple inputs

The code is self-explanatory. You should name at least the most important layers, especially when the model gets a bit complex like this. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train` , we must pass a pair of matrices (`X_train_A`, `X_train_B`) : one per input. 19 The same is true for `X_valid` , and also for `X_test` and `X_new` when you call `evaluate()` or `predict()` :

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model

# creating the two inputs
input_A = keras.layers.Input(shape = (5)) # the dimension(feature) of the input from 0 to 4
input_B = keras.layers.Input(shape = (6)) # the dimension(feature) of the input from 2 to 7
# now let's create two hidden layers
hidden_1 = keras.layers.Dense(units = , activation = keras.activations.relu , name = 'first_hidden')(input_B)
hidden_2 = keras.layers.Dense(units = , activation = keras.activations.relu , name = 'second_hidden')(hidden_1)
concat = keras.layers.concatenate([input_A , hidden_2])
output_layer = keras.layers.Dense(units = , activation = keras.activations.sigmoid , name = 'output_layer')(concat)
# creating the model
model = keras.Model(inputs = [input_A , input_B] , outputs = [output_layer])
# now using these model we can summarize , plot , fit , evaluate , predict
model.summary() # summarize the model
plot_model(model = model , show_shapes = True) # plotting the model
# before fitting the model , we need to specify the inputs datas in all of the three cases (training , validating and testing )
X_train_A , X_train_B = X_train[:, :5] , X_train[:, 2:7]
X_valid_A , X_valid_B = X_valid[:, :5] , X_valid[:, 2:7]
X_test_A , X_test_B = X_test[:, :5] , X_test[:, 2:7]
X_new_A , X_new_B = X_test_A[:3] , X_test_B[0:3]
# Now we can actually compile the model
model.compile(optimizer=keras.optimizers.Adam , loss=keras.losses.categorical_crossentropy , metrics=[keras.metrics.accuracy , categorical_crossentropy] )
# now fitting the data to the model
history = model.fit([X_train_A , X_train_B] , y_train , epochs = 50 , validation_data = ([X_valid_A , X_valid_B] , y_valid))
# now let's evaluate
model.evaluate([X_test_A , X_test_B] , y_test)
# we can predict
y_pred = model.predict(X_new_A , X_new_B)

```

We will develop an image classification model that takes two versions of the image as input, each of a different size. Specifically a black and white 64×64 version and a color 32×32 version. Separate feature extraction CNN models operate on each, then the results from both models are concatenated for interpretation and ultimate prediction.

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D

# creating the first input
input_A = keras.layers.Input(shape = (64 , 64 , 1))
# let's create the convolutional layer
conv1 = Conv2D(32 , kernel_size= 4 , activation = keras.activations.relu)(input_A)
pool1 = MaxPooling2D(pool_size=(2,2))(conv1)
conv2 = Conv2D(16 , kernel_size=4 , activation = keras.activations.relu)(pool1)
pool2 = MaxPooling2D(pool_size = (2 , 2))(conv2)
flat1 = Flatten()(pool2)

# creating the second input
input_B = keras.layers.Input(shape = (32 , 32 , 3))
# let's create the second convolutional layer
conv11 = Conv2D(32 , kernel_size=4 , activation = keras.activations.relu)(input_B)
pool11 = MaxPooling2D(pool_size=(2 , 2))(conv11)
conv22 = Conv2D(16 , kernel_size = 4 , activation=keras.activations.relu)(pool11)
pool22 = MaxPooling2D(pool_size=(2,2))(conv22)
flat2 = Flatten()(pool22)

# now let's concat the two flat layers in to one big vector
concat = keras.layers.concatenate([flat1 , flat2])
# creating the hidden layers
hidden1 = keras.layers.Dense(units = 10 , activation = keras.activations.relu , name = 'first_hidden')(concat)
hidden2 = keras.layers.Dense(units = 10 , activation = keras.activations.relu , name = 'second_name')(hidden1)
output_layer = keras.layers.Dense(units = 1 , activation = keras.activations.sigmoid , name = 'output_layer')(hidden2)
# creating the model
model = keras.Model(inputs = [input_A , input_B] , outputs = [output_layer])
model.summary()

```

...

Multiple Output Model

There are many use cases in which you may want to have multiple outputs:

- The task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object's center, as well as its width and height) and a classification task.

- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform multitask classification on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or

not.

- Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce over-fitting and thus improve the model's ability to generalize). For example, you may want to add some auxiliary outputs in a neural network architecture (see Figure 10-16) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

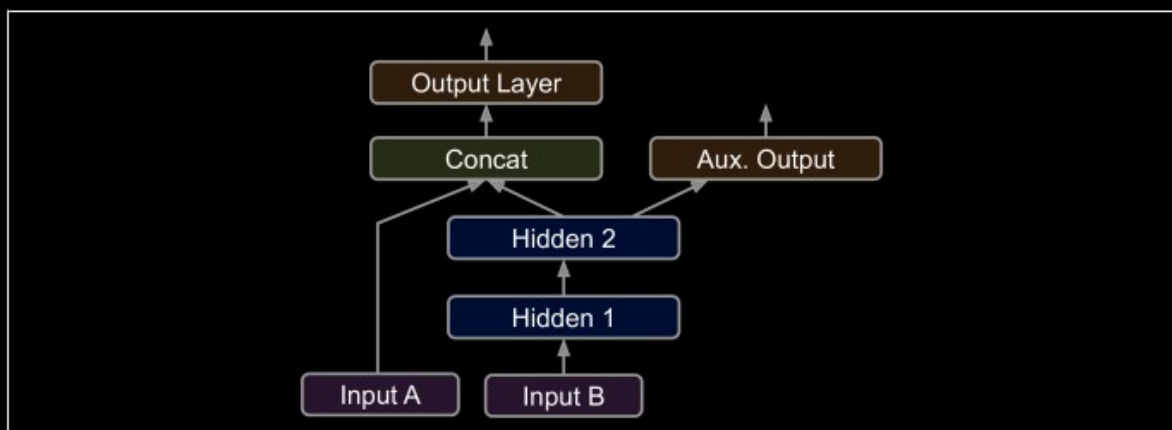


Figure 10-16. Handling multiple outputs, in this example to add an auxiliary output for regularization

Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model's list of outputs.

```
+ Code + Text
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model

# creating the two inputs
input_A = keras.layers.Input(shape = (5))
input_B = keras.layers.Input(shape = (5))

# creating the deep net
hidden1 = keras.layers.Dense(units = 10, activation=keras.activations.relu, name = "first_hidden")(input_B)
hidden2 = keras.layers.Dense(units = 10, activation = keras.activations.relu, name = "Second_hidden")(hidden1)

# concatenating the wide and the deep path
concat = keras.layers.concatenate([input_A, hidden2])
# creating the output
main_output = keras.layers.Dense(units = 1, activation = keras.activations.sigmoid, name = "main_output")(concat)
Aux_output = keras.layers.Dense(units = 1, activation = keras.activations.sigmoid, name = "Aux_output")(hidden2)

# creating the model
model = keras.Model(inputs = [input_A, input_B], outputs = [main_output, Aux_output])

# now let's create the training , validation and test data sets
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:7]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:7]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:7]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

# now let's compile the model
model.compile(loss = [keras.losses.sparse_categorical_crossentropy, keras.losses.sparse_categorical_crossentropy], loss_weights=[0.9, 0.1], optimizer=keras.optimizers.SGD())
# Now let's fit the data to the model
history = model.fit(X_train_A, X_train_B, (y_train, y_train), epochs = 10, validation_data = (X_valid_A, X_valid_B), (y_valid, y_valid))
# now let's evaluate the model , keras will calculate the total loss and the individual output loss
total_loss, main_loss, Aux_loss = model.evaluate((X_test_A, X_test_B), (y_valid, y_valid))
# now let's predict
main_predict, Aux_predict = model.predict(X_new_A, X_new_B)
# we can summarize the model
model.summary()
# we can plot the model
plot_model(model=model, show_shapes = True)
```

0s completed at 10:14 AM

Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses (if we pass a single loss, Keras will assume that the same loss must be used for all outputs). By default, Keras will compute all these losses and simply add them up to get the final loss used for training. We care much more about the main output than about the auxiliary output (as it is just used for regularization), so we want to give the main output's loss a much greater weight. Fortunately, it is possible to set all the loss weights when compiling the model.

Now when we train the model, we need to provide labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels. So instead of passing `y_train`, we need to pass `(y_train, y_train)` (and the same goes for `y_valid` and `y_test`)

Using the Subclassing API to Build Dynamic Models