

This is a companion notebook for the book [Deep Learning with Python, Second Edition](#). For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

**If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.**

This notebook was generated for TensorFlow 2.6.

## ▼ Introduction to Keras and TensorFlow

What's TensorFlow?

What's Keras?

Keras and TensorFlow: A brief history

## ▼ Setting up a deep-learning workspace

Jupyter notebooks: The preferred way to run deep-learning experiments

## ▼ Using Colaboratory

First steps with Colaboratory

Installing packages with pip

Using the GPU runtime

## ▼ First steps with TensorFlow

## ▼ Constant tensors and variables

## All-ones or all-zeros tensors

```
import tensorflow as tf
x = tf.ones(shape=(2, 1))
print(x)
```

```
x = tf.zeros(shape=(2, 1))
print(x)
```

## Random tensors

```
x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
print(x)
```

```
x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
print(x)
```

## NumPy arrays are assignable

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

## Creating a TensorFlow variable

```
v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
print(v)
```

## Assigning a value to a TensorFlow variable

```
v.assign(tf.ones((3, 1)))
```

## Assigning a value to a subset of a TensorFlow variable

```
v[0, 0].assign(3.)
```

## Using assign\_add

```
v.assign_add(tf.ones((3, 1)))
```

## ▼ Tensor operations: Doing math in TensorFlow

## A few basic math operations

```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
e *= d
```

### ▼ A second look at the GradientTape API

#### Using the GradientTape

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

#### Using GradientTape with constant tensor inputs

```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

#### Using nested gradient tapes to compute second-order gradients

```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position = 4.9 * time ** 2
        speed = inner_tape.gradient(position, time)
    acceleration = outer_tape.gradient(speed, time)
```

### ▼ An end-to-end example: A linear classifier in pure TensorFlow

#### Generating two classes of random points in a 2D plane

```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
```

```
size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

### Stacking the two classes into an array with shape (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

### Generating the corresponding targets (0 and 1)

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

### Plotting the two point classes

```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

### Creating the linear classifier variables

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

### The forward pass function

```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

### The mean squared error loss function

```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

### The training step function

```
learning_rate = 0.1

def training_step(inputs, targets):
```

```

with tf.GradientTape() as tape:
    predictions = model(inputs)
    loss = square_loss(predictions, targets)
grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
W.assign_sub(grad_loss_wrt_W * learning_rate)
b.assign_sub(grad_loss_wrt_b * learning_rate)
return loss

```

## The batch training loop

```

for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")

predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()

x = np.linspace(-1, 4, 100)
y = - W[0] / W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)

```

- ▼ Anatomy of a neural network: Understanding core Keras APIs
- ▼ Layers: The building blocks of deep learning
- ▼ The base Layer class in Keras

## A Dense layer implemented as a Layer subclass

```

from tensorflow import keras

class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,),
                                initializer="zeros")

```

```
def call(self, inputs):
    y = tf.matmul(inputs, self.W) + self.b
    if self.activation is not None:
        y = self.activation(y)
    return y
```

```
my_dense = SimpleDense(units=32, activation=tf.nn.relu)
input_tensor = tf.ones(shape=(2, 784))
output_tensor = my_dense(input_tensor)
print(output_tensor.shape)
```

## ▼ Automatic shape inference: Building layers on the fly

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu")
```

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

## From layers to models

## ▼ The "compile" step: Configuring the learning process

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])
```

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

## Picking a loss function

### ▼ Understanding the fit() method

#### Calling fit() with NumPy data

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

```
history.history
```

### ▼ Monitoring loss and metrics on validation data

#### Using the validation\_data argument

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

### ▼ Inference: Using a model after training

```
predictions = model.predict(val_inputs, batch_size=128)
print(predictions[:10])
```

# Summary

