Notebook credit: based on the F. Chollet's original notebook here.

Introduction to Keras and TensorFlow

Numpy, Matplotlib, Scikit-learn, TensorFlow 和 Keras 的关系

- 1. Python 是一个 interpreted high-level general-purpose programming language.
- 2. NumPy 是一个 Python library, 用于 manipulate 高维度数组和矩阵 (用于 numerical computing).
- 3. Matplotlib 是一个 Python library, 用于绘图像, 和 numpy 很适配. **它的一个 module `pyplot` 提供了一个和 MATLAB 很像的 interface.**
- 4. Scikit-learn 是一个**建立在 numpy 和 Matplotlib 之上的 Python library.** This is the main library for **machine learning**, 尤其是并不基于神经网络的 ML(也就是说和深度学习关系不大).
- 5. TensorFlow 是一个由 google 研发的 Python library, 用以研发 deep learning models on CPUs, GPUs, and TPUs. TF2 was released in 2019.
- 6. Keras 是一个 **built on top of TensorFlow** 的 Python library. 它(1) 提供 consistent & simple APIs, (2) 最大程度地简化了 common use cases, 并且(3) 提供了清晰的 actionable error messages.
  Keras 的作者名叫 Francois Chollet, 他也是 *Deep Learning with Python* 的作者.

TensorFlow 比起 numpy 有什么优势

Like NumPy, TF's main purpose is to enable engineers/researchers to manipulate numerical tensors. But TensorFlow goes far beyond the scope of NumPy:

\TensorFlow 比起 Numpy 好在这些地方.

- 1. 不用手算梯度, 可以**自动计算任何 differentiable expression 的 gradient**, 非常适合 machine learning.
- 2. 不仅可以在 CPU 上运行, 还可以在 GPUs 和 TPUs 这两个 highly parallel hardware accelerators 上运行, 跑大模型更快.
- 3. Computation defined in TensorFlow 可轻松地分布在多台机器上, 这样就可以分工计算, 速度会快很多.

- 4. TensorFlow programs 可以被 **exported to other runtimes**, 比如 C++, JavaScript (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. 因而 TensorFlow applications 更容易投入实际应用.

TF 不仅仅是一个 library, 而且是 ecosystem.
For example, there is:

- TF-Agents for reinforcement learning research
- TensorFlow Hub for pretrained models
- TensorFlow Quantum for quantum machine learning

Keras 是什么

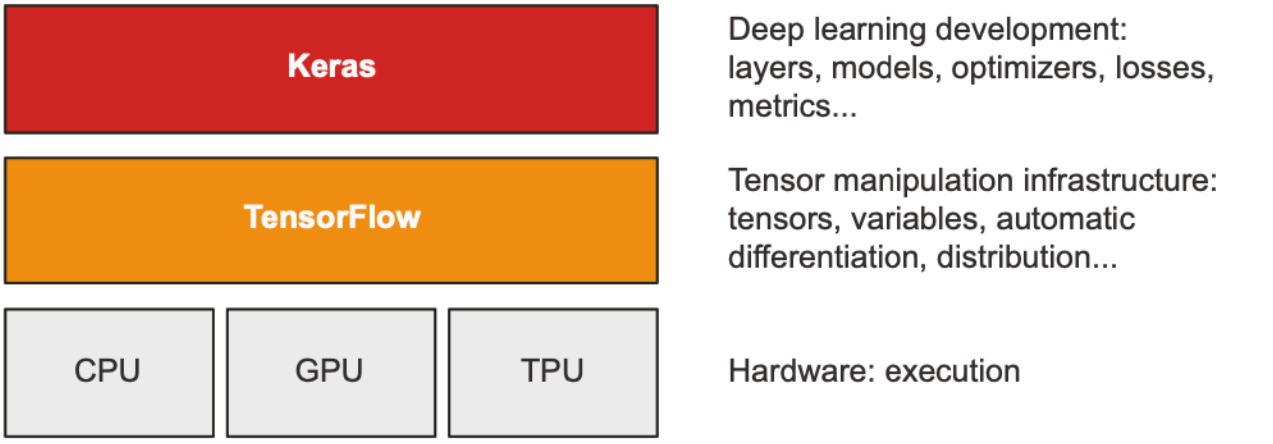


Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

Keras and TensorFlow 的发展历史 history

- Keras first released March 2015
- TF first released November 2015
- Keras originally built on top of Theano, a precursor of TF
- In late 2015, Keras refactored to allow multiple backend: Theano or TF
- In 2017, CNTK (Microsoft) and MXNet (Amazon) were added as backends
- 2016-2017: Keras becomes well-known as a user-friendly way to develop TF applications
- By late 2017, majority of TF users are using Kera

- In 2018, TF leadership decides to pick Keras as TF's official high-level API
- Keras API is front and center in TensorFlow 2.0, released in September 2019

Nowadays, both Theano and CNTK are out of development, and MXNet is not widely used outside of Amazon. Keras is back to being a single-backend API—on top of TensorFlow.

如何 set up deep-learning workspace

最好有一个 modern NVIDIA GPU, 在 GPU 上跑. 因为比 CPU 快 5x-10x 倍. 而且 CPU 有些应用跑得非常慢.

为了用 GPU 做 DL, 有三个选项

1. Buy and install a physical NVIDIA GPU on your workstation.
2. Use GPU instances on Google Cloud or AWS EC2.
3. Use the free GPU runtime from Colaboratory (Colab), a hosted notebook service offered by Google

In 315, we will use option no. 3.

- 不用配置环境
- 不用买硬件
- Colab 只对 small workloads 免费, 恰好我们是 small workloads.

Jupyter notebooks 为什么是 DL exvironment 的 preferred way

- 1. They’re widely used in the data science and machine learning communities.
- 2. 同时可以运行 Python, R 和 Julia, 并支持丰富的 md, html 文本编辑.
- 3. 把 long experiments 切分为可以独立运行的 smaller pieces, 以至于
  - makes development interactive
  - 如果有东西错误了, 不需要 rerun all previous code.

> ("Jupyter" 名字的来源: The name “Jupyter” is a strong reference to Galileo, who detailed his discovery of the Moons of Jupiter in his astronomical notebooks. The name is also a play on the languages Julia, Python, and R, which are pillars of the modern scientific world. source)

如何使用 Colaboratory

Shift+Enter 运行 Code / 渲染 Text

- Code cells: Shift+Enter will execute them
- Text cells: Shift+Enter will render them. Can use markdown and HTML. Plus you can use LaTeX to typeset equations like $e^{i\pi} + 1 = 0$

在 Code 块用 !pip 起头, 在线安装额外的 library(一次性使用)

Colab has many packages including TensorFlow and Keras. A command in a code cell that begins with an exclamation mark passes the command to the shell.

```
!pip list -v | grep "keras\|tensorflow"
```

```
keras                         2.15.0               /usr/local/lib/python3.10
tensorflow                    2.15.0               /usr/local/lib/python3.10
tensorflow-datasets           4.9.4                /usr/local/lib/python3.10
tensorflow-estimator          2.15.0               /usr/local/lib/python3.10
tensorflow-gcs-config         2.15.0               /usr/local/lib/python3.10
tensorflow-hub                0.16.1               /usr/local/lib/python3.10
tensorflow-io-gcs-filesystem  0.36.0               /usr/local/lib/python3.10
tensorflow-metadata           1.14.0               /usr/local/lib/python3.10
tensorflow-probability        0.23.0               /usr/local/lib/python3.10
tf-keras                      2.15.0               /usr/local/lib/python3.10
```

If a package you need isn't installed you can install it using

```
!pip install <package-name>
```

but note that the installation will have to be repeated each time the notebook is run.

使用 GPU 跑代码

To use the GPU runtime with Colab, select `Runtime > Change Runtime Type` in the menu and select GPU for the Hardware Accelerator

```
import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
GPUs = tf.config.list_physical_devices('GPU')
gpu = [GPU for GPU in GPUs]
gpu[0]
```

```
Num GPUs Available:  1
PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')
```

TensorFlow 和 Keras 的 training 的关键成分

Training a neural network revolves around the following concepts:

First, low-level tensor manipulations which translate to TensorFlow APIs:

- *Tensors*, including special tensors that store the network's state (variables)
- *Tensor operations* such as addition, `relu`, `matmul`
- *Backpropagation*, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the `GradientTape` object)

Second, high-level deep learning concepts which translate to Keras APIs:

- *Layers*, which are combined into a *model*
- A *loss function*, which is used by the optimizer to assess how good a model is
- An *optimizer*, which determines how learning proceeds
- *Metrics* to evaluate model performance, such as accuracy
- A *training loop* that performs mini-batch stochastic gradient descent

Constant tensors

创建全0和全1 tensors**

```
x = tf.ones(shape=(2, 3)) # NumPy equivalent: np.ones(shape=(2, 3))
print(x)
```

```
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
```

```
x = tf.zeros(shape=(2, 1)) # NumPy equivalent: np.zeros(shape=(2, 1))
print(x)
```

```
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

创建**Random tensors**

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

```
tf.Tensor(
[[0.17085923]
 [1.235519  ]
 [1.2031974 ]], shape=(3, 1), dtype=float32)
```

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

```
tf.Tensor(
[[0.66221464]
 [0.69043124]
 [0.88149273]], shape=(3, 1), dtype=float32)
```

在 tensor 和 numpy array 之间转换

Converting between a TensorFlow tf.Tensors and a NumPy ndarray is easy:

- TensorFlow operations 会**自动转化 NumPy ndarrays 为 Tensors.**
- NumPy operations **也会自动转化 Tensors 为 NumPy ndarrays**.
- Tensors 可以被 explicitly 转化为 NumPy ndarrays, 通过 **`.numpy()`** method.

```
import numpy as np

ndarray = np.ones([3, 3])

print("TensorFlow operations convert numpy arrays to Tensors automatically")
tensor = tf.multiply(ndarray, 42)
print(tensor)

print("And NumPy operations convert Tensors to numpy arrays automatically")
print(np.add(tensor, 1))

print("The .numpy() method explicitly converts a Tensor to a numpy array")
print(tensor.numpy())
```

```
TensorFlow operations convert numpy arrays to Tensors automatically
tf.Tensor(
[[42. 42. 42.]
 [42. 42. 42.]
 [42. 42. 42.]], shape=(3, 3), dtype=float64)
And NumPy operations convert Tensors to numpy arrays automatically
[[43. 43. 43.]
 [43. 43. 43.]
 [43. 43. 43.]]
The .numpy() method explicitly converts a Tensor to a numpy array
[[42. 42. 42.]
 [42. 42. 42.]
 [42. 42. 42.]]
```

NumPy arrays are assignable, 但是 tensor 却不是 assignable 的.

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

```
[[1. 1.]
 [1. 1.]]
[[0. 1.]
 [1. 1.]]
```

TensorFlow tensors are not assignable

```
try:
  x = tf.ones(shape=(2, 2))
  x[0, 0] = 0
except Exception as e:
  print(e)
```

```
'tensorflow.python.framework.ops.EagerTensor' object does not support item assig
```

在 tensorflow 中, tensor 就是 constant 的. 对于变量的操作, 我们采用 tf.Variable.

TensorFlow variable

创建一个 tensorflow variable

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

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[-1.1514548 ],
       [-1.411351  ],
       [-0.26656958]], dtype=float32)>
```

给 TensorFlow variable 赋值

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

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

给 TensorFlow variable 的一部分赋值

```
v[0:2, 0].assign(3.)
```

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[3.],
       [3.],
       [1.]], dtype=float32)>
```

assign_add (形状必须和本来一样)

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

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

assign_sub (可以对于任何形状的一部分)

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

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Tensor operations

Like NumPy, TensorFlow offers many math operations on tensors

A few basic math operations

```
a = tf.ones((2, 2))
print("a=\n", a)
b = tf.square(a)    # elementwise square
print("b=\n", b)
c = tf.sqrt(a)      # elementwise square root
print("c=\n", c)
d = b + c           # elementwise sum
print("d=\n", d)
e = tf.matmul(a, b) # matrix multiplication
print("e=\n", e)
e *= d              # elementwise multiplication
print("e=\n", e)
```

```
a=
 tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
b=
 tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
c=
 tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
d=
 tf.Tensor(
[[2. 2.]
 [2. 2.]], shape=(2, 2), dtype=float32)
e=
 tf.Tensor(
[[2. 2.]
 [2. 2.]], shape=(2, 2), dtype=float32)
e=
 tf.Tensor(
[[4. 4.]
 [4. 4.]], shape=(2, 2), dtype=float32)
```

The GradientTape API: 自动运算梯度

API: Application Programming Interface, 用于不同软件程序之间的交互, 允许不同的软件应用程序或系统共享数据和功能，而无需开发者深入了解它们的内部工作原理.

Backpropagation(后向传播)

Training a model via (minibatch stochastic) gradient descent involves computing gradients of the loss function w.r.t. model parameters.

比如下面这个 two dense layer architecture (之后会介绍Keras)

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

如果使用 `W1, b1, W2, b2` 来表示这两个 layer 之间的 weights and biases, 那么 loss value 可以用下面这段代码表示:

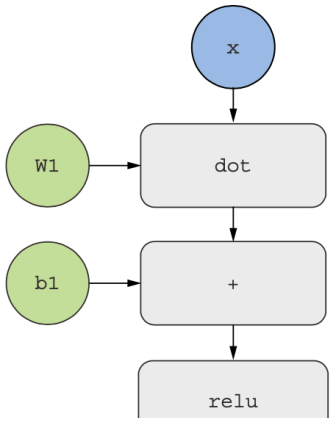
```
loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
```

因为这些 functions `loss`, `softmax`, `dot` (matrix multiplication) 以及 `relu` ($f(x) = max(0, x)$, derivative 在 $x > 0$ 时为1, $x < 0$ 时为0, $x = 0$ 时需设置) 都是 differentiable的, 我们可以使用 *chain rule* 来计算 derivate of `loss_value` w.r.t. model parameters.

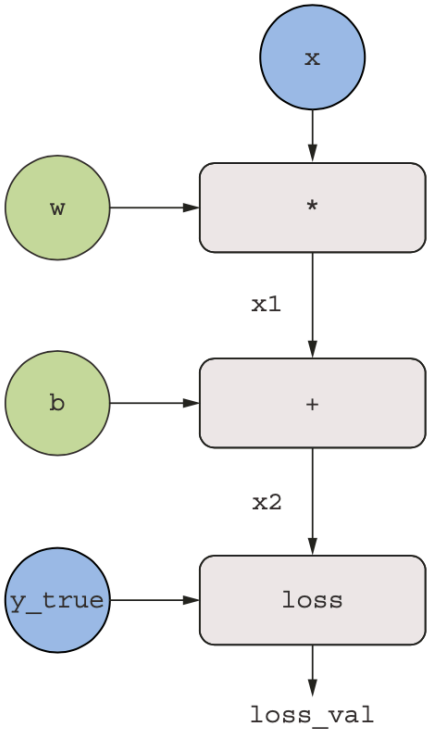
Backpropagation 就是使用 chain rule 计算一个 neural network 的 gradient values.

Automatic differentiation with computation graphs

Below is the computation graph of the simple two layer NN we saw above.



Let's consider a simplified version of the above figure:




```
x1 = w * x
x2 = x1 + b
loss_val = abs(y_true - x2)
```

- we have only one layer
- all variables are scalars
- we will use absolute value error: `loss_val = abs(y_true - x2)`
- we are interested in computing `grad(loss_val, b)` and `grad(loss _val, w)`

Suppose `x = 2, y_true = 4` and we want to evaluate the derivatives at `w = 3` and `b = 1`.

Let's first do it "by hand".

$$f(w, b) = |y_{true} - (wx + b)| = |4 - (2w + b)|$$

Diff. w.r.t. w

$f(w) = |4 - (2w + 1)| = g(h(H(w)))$ where $g(u) = |4 - u|, h(u) = u + 1, H(u) = 2u$

$\frac{\partial f}{\partial w} = g'(h(H(w))) \times h'(H(w)) \times H'(w) = g'(7) \times h'(6) \times H'(3) = 1 \times 1 \times 2 = 2$

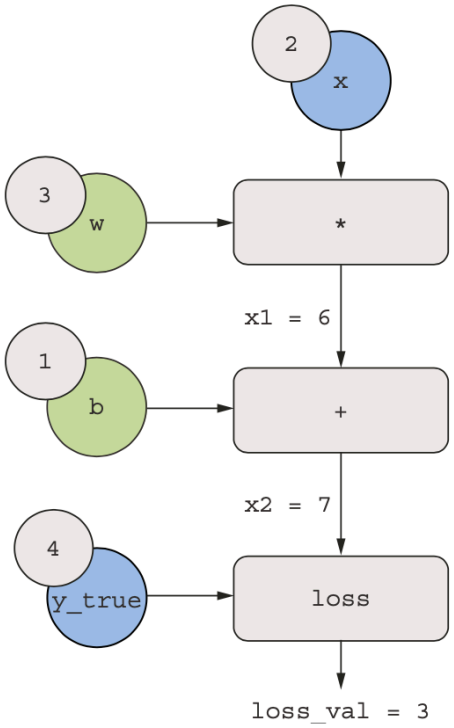
Diff. w.r.t. b

$f(b) = |4 - (6 + b)| = g(h(b))$ where $g(u) = |4 - u|, h(u) = 6 + u$

$\frac{\partial f}{\partial b} = g'(h(b)) \times h'(b) = g'(7) \times h'(1) = 1 \times 1 = 1$

Forward pass:

```
x1 = w * x = 3 * 2 = 6
x2 = x1 + b = 6 + 1 = 7
loss_val = abs(y_true - x2) = abs(4-7) = 3
```



Backward pass:

- `grad(loss_val, x2) = 1` because $\frac{\partial|4-x_2|}{\partial x_2} = 1$ at $x_2 = 7$
- `grad(x2,x1)=1` because $\frac{\partial(x_1+b)}{\partial x_1} = 1$ at $x_1 = 6$
- `grad(x2,b)=1` because $\frac{\partial(x_1+b)}{\partial b} = 1$ at $b = 1$

- `grad(x1,w)=2` because $\frac{\partial(w*2)}{\partial w} = 2$ at $w = 3$

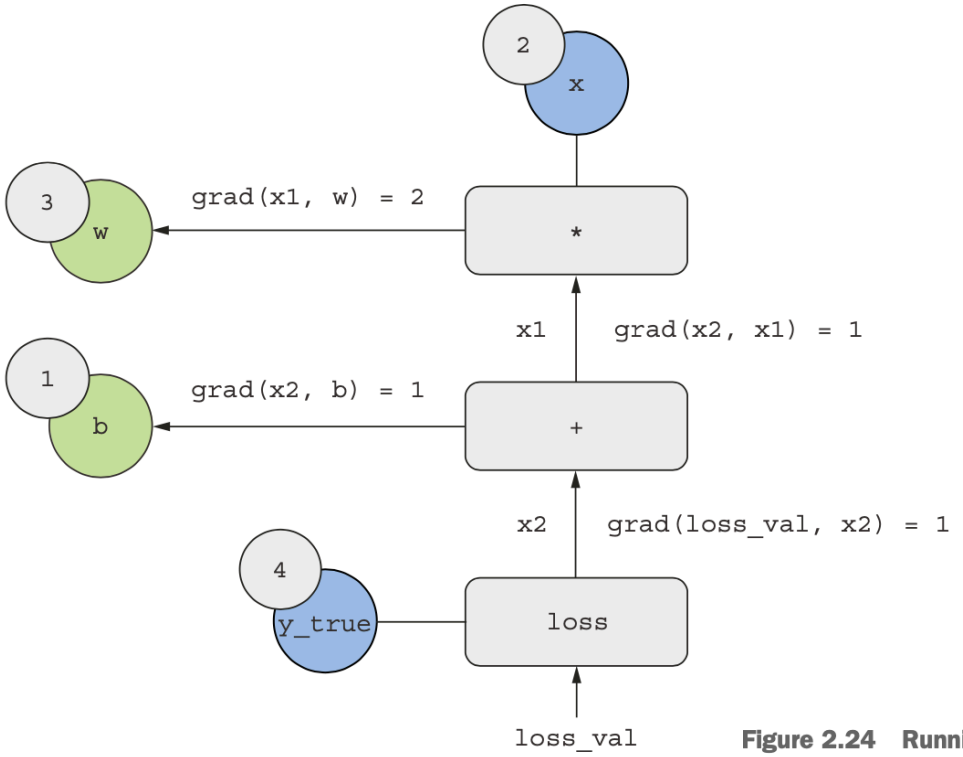


Figure 2.24 Running a backward pass

```
grad(loss_val,w) = grad(loss_val, x2)*grad(x2,x1)*grad(x1,w) = 1 * 1 * 2 = 2
grad(loss_val,b) = grad(loss_val, x2)*grad(x2,b) = 1*1 = 1
```

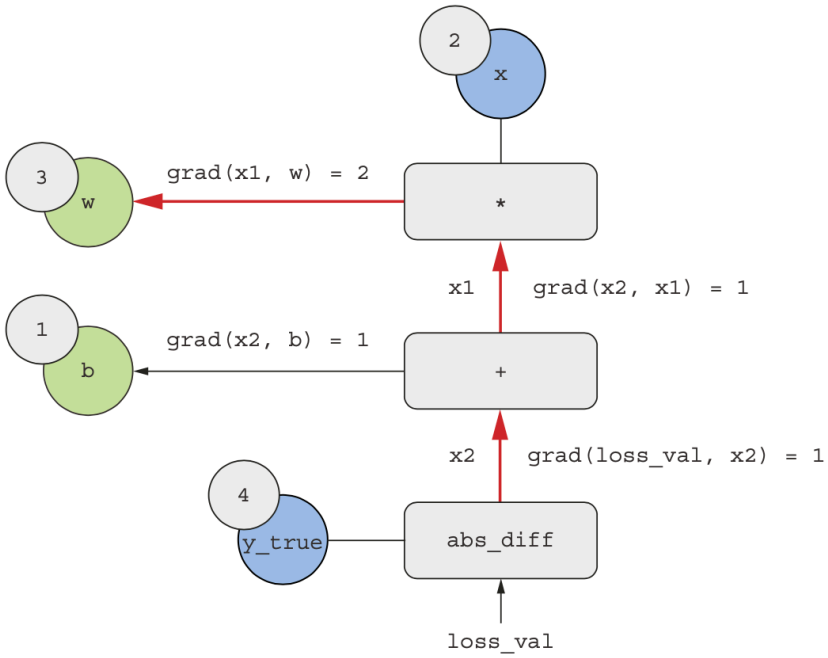


Figure 2.25 Path from `loss_val` to `w` in the backward graph

使用 GradientTape 自动计算梯度

```
w = tf.Variable(3.)
b = tf.Variable(1.)
x = tf.constant(2.)
y_true = tf.constant(4.)

with tf.GradientTape() as tape:
  x1 = w*x
  x2 = x1 + b
  loss_val = tf.abs(y_true - x2)

grad_w, grad_b, grad_x1, grad_x2 = tape.gradient(loss_val, [w, b, x1, x2]) # gettin
print("deriv of loss_val w.r.t. w = ", grad_w.numpy())
print("deriv of loss_val w.r.t. b = ", grad_b.numpy())
print("deriv of loss_val w.r.t. x1 = ", grad_x1.numpy())
print("deriv of loss_val w.r.t. x2 = ", grad_x2.numpy())
```

```
deriv of loss_val w.r.t. w =  2.0
deriv of loss_val w.r.t. b =  1.0
deriv of loss_val w.r.t. x1 =  1.0
deriv of loss_val w.r.t. x2 =  1.0
```

$$r = x^2$$

$$\frac{dr}{dx} = 2x = 6 \text{ at } x = 3$$

```
x = tf.Variable(3.)
with tf.GradientTape() as tape:
   r = tf.square(x)
gradient = tape.gradient(r, x)
print("deriv of r w.r.t. x = ", gradient.numpy())
```

```
deriv of r w.r.t. x =  6.0
```

针对 constant tensor inputs 的 GradientTape

- **GradientTape 默认只会 track trainable *variables***, 为了防止 wasting resources
- 如果要使用 *constant* tensor 作为 inputs, 需要call `tape.watch()`

```
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)
print("deriv of result w.r.t. input_const = ", gradient.numpy())
```

```
deriv of result w.r.t. input_const =  6.0
```

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

生成两个 2-D classes of random points

```
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)
```

通过 **np.stack** 把两个 classes 叠起来

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

```
(2000, 2)
```

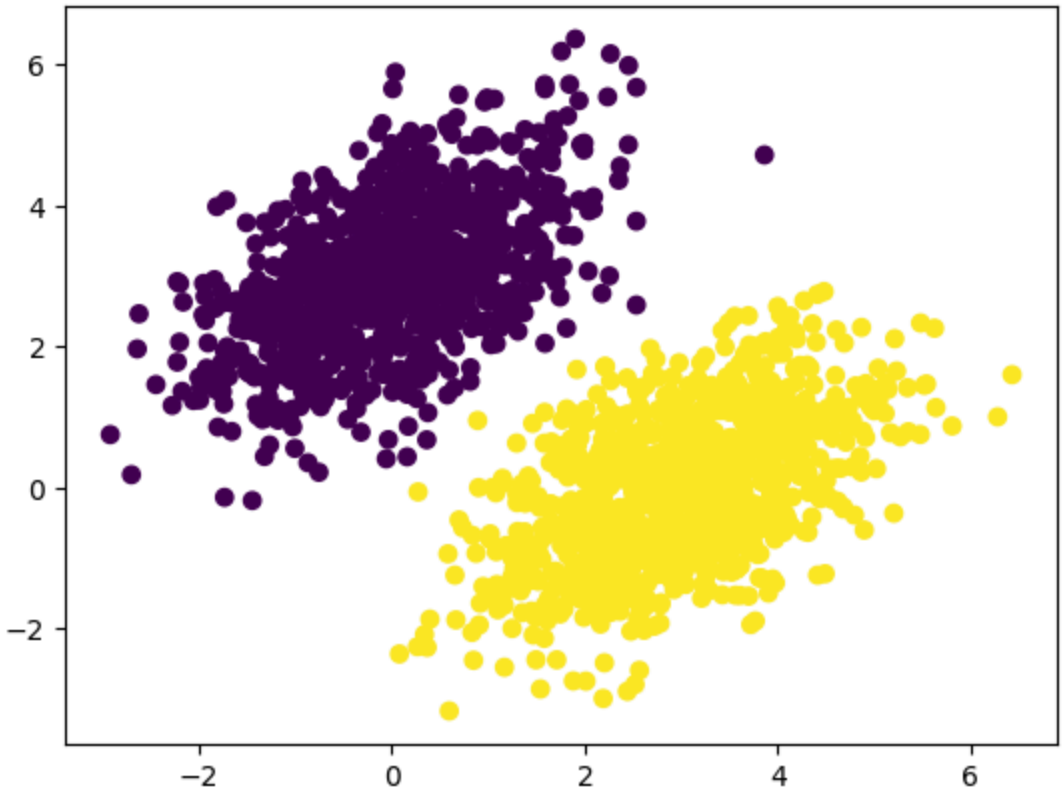
Generating the corresponding targets (0 and 1)

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

```
(2000, 1)
```

绘制 2-D 图像

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



创建 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,)))
```

定义**forward pass function** (前向传播, 也就是计算 $\hat{y}$)

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

使用 **logistic loss function** 的分类

Note that book's example uses the squared loss function. For true label $y \in 0, 1$ and model output $o \in \mathbb{R}$, squared loss is defined as:

$$(y - o)^2 .$$

Question: We studied squared loss in a regression problem. Why is it meaningful to use squared loss in a classification problem?

We will instead use the logistic loss which is defined as

$$\log(1 + \exp(o)) - y \cdot o \ .$$

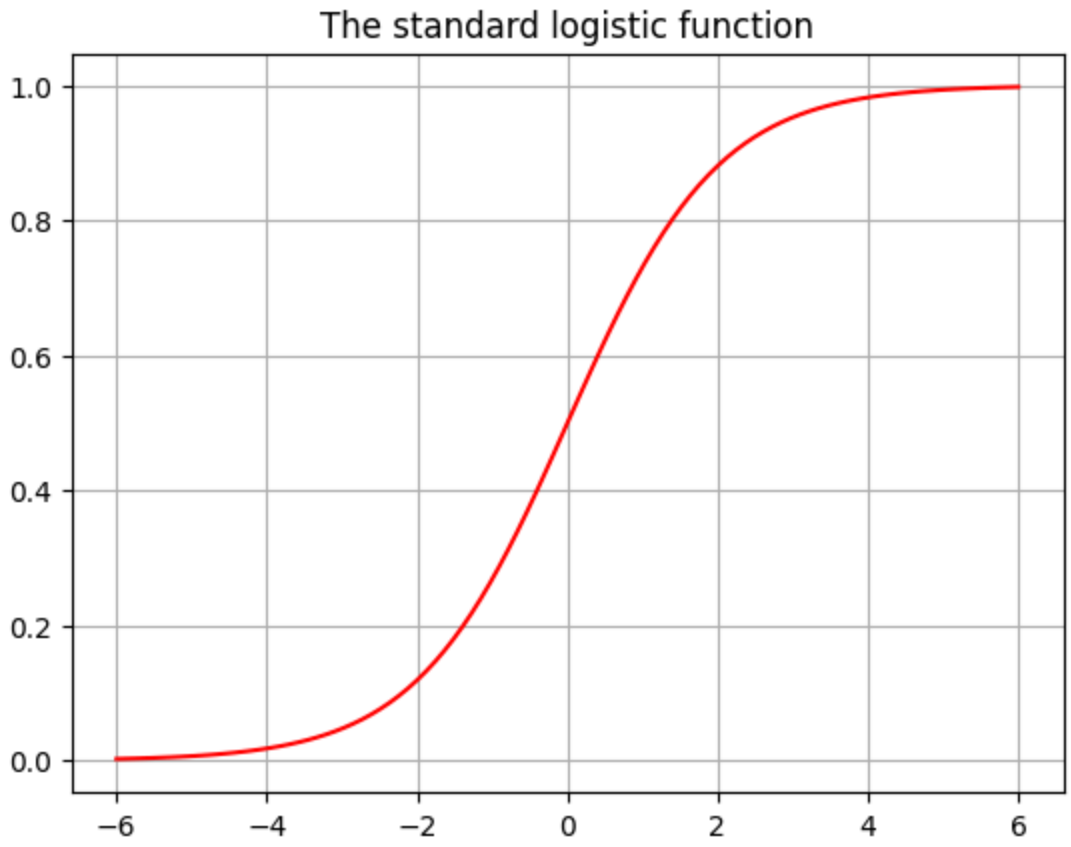
This is nothing but the cross-entropy loss in the binary classification case where we use a linear model $o = \mathbf{w}^T \mathbf{x} + b$ and convert this output o into a probability using the nonlinear standard logistic map:

$$\hat{y} = \frac{\exp(o)}{1 + \exp(o)} = \frac{1}{1 + \exp(-o)}$$

We can then interpret our overall model as modelling the probability of $y = 1$ given $\mathbf{x}$ as

$$P(y = 1|\mathbf{x}) = \frac{\exp(\mathbf{w}^T \mathbf{x} + b)}{1 + \exp(\mathbf{w}^T \mathbf{x} + b)}$$

```
o = np.linspace(-6, 6, 100)
y_hat = 1./(1 + np.exp(-o))
plt.plot(o, y_hat, "-r")
plt.grid()
plt.title("The standard logistic function")
plt.show()
```



使用 tf 变量的 logistic loss function

```
def logistic_loss(targets, predictions):
    per_sample_losses = tf.math.log(1.0 + tf.math.exp(predictions)) - targets * pre
    return tf.reduce_mean(per_sample_losses)
```

使用 GradientTape 的 training step function

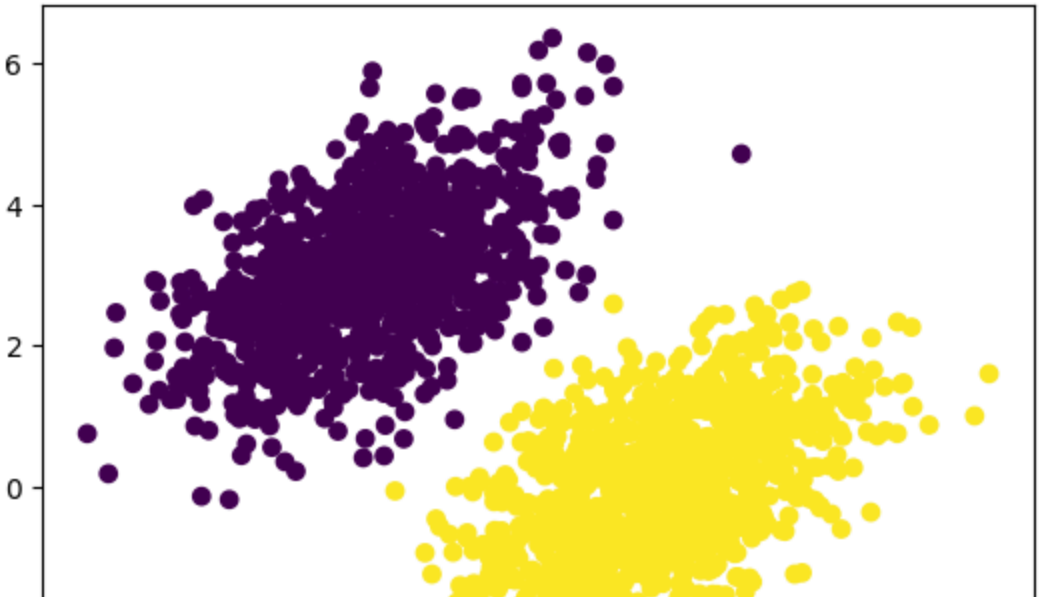
```
def training_step(inputs, targets, learning_rate):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = logistic_loss(targets, predictions)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b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

The batch training loop

```
learning_rate = 0.1
for step in range(100): # increase this to get better results
    loss = training_step(inputs, targets, learning_rate)
    if (step+1) % 10 == 0:
        print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 9: 0.2128
Loss at step 19: 0.1332
Loss at step 29: 0.1007
Loss at step 39: 0.0824
Loss at step 49: 0.0704
Loss at step 59: 0.0619
Loss at step 69: 0.0555
Loss at step 79: 0.0506
Loss at step 89: 0.0465
Loss at step 99: 0.0432
```

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




```
x = np.linspace(-3, 5, 100)
y = - W[0] /  W[1] * x - b / W[1]
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0)
plt.plot(x, y, "-r")
plt.show()
```