# 5.数据构建

tf.data简介

面对一堆格式不一的原始数据文件 ？  
 读入程序的过程往往十分繁琐 ？  
 运行的效率上不尽如人意 ？  
 T e n so r F l ow 提供了 tf.data 这一模块，包括了一套灵活的数据集构建 API，能够帮助我们快速 、高效地构建数据输入的流水线 ，尤其适用于数据量巨大的场景。

tf.data包含三个类：

• tf.data.Dataset类  
 • tf.data.TFRecordDataset类  
 • tf.data.TextLineDataset类

## 5.1.Dataset类

tf.data 的核心是 tf.data.Dataset 类，提供了对数据集的高层封装。  
 tf.data.Dataset 由一系列的可迭代访问的元素（element）组成，每个元素包含一个或多个张量。 Dataset可以看作是相同类型“元素”的有序列表。

注： Dataset可以看作是相同类型“元素”的有序列表。在实际使用时，单个“元素”  
可以是向量，也可以是字符串、图片，甚至是tuple或者dict。   
 比如说，对于一个由图像组成的数据集，每个元素可以是一个形状为 长×宽×通道数的图片张量，也可以是由图片张量和图片标签张量组成的元组（Tuple）。

常用创建tf.data.Dataset数据集的方法有：

tf.data.Dataset.from\_tensors() ：创建Dataset对象，返回具有单个元素的数据集。  
 tf.data.Dataset.from\_tensor\_slices() ：创建一个Dataset对象，但是会将第0维切分  
 tf.data.Dataset. from\_generator() ：迭代生成所需的数据集，一般数据量较大时使用。

from\_tensors() 函数会把传入的tensor当做一个元素，但是from\_tensor\_slices() 会把传入的tensor除开第0维之后的大小当做一个元素

最基础的建立 tf.data.Dataset 的方法是使用 tf.data.Dataset.from\_tensor\_slices() ，适用于数据量较小（能够整个装进内存）的情况

具体而言，如果我们的数据集中的所有元素通过张量的第 0 维，拼接成一个大的张量（例如，前节的 MNIST 数据集的训练集即为一个 [60000, 28, 28, 1] 的张量，表示了 60000 张 28\*28 的单通道灰度图像），那么我们提供一个这样的张量或者第 0 维大小相同的多个张量作为输入，即可按张量的第 0 维展开来构建数据集，数据集的元素数量为张量第 0 位的大小。

**import** **tensorflow** **as** **tf**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

mnist = np.load("mnist.npz")

x\_train, y\_train = mnist['x\_train'],mnist['y\_train']

x\_train.shape,y\_train.shape

((60000, 28, 28), (60000,))

x\_train = np.expand\_dims(x\_train, axis=-1)

mnist\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train))

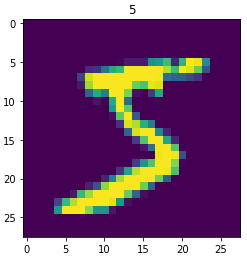
**for** image, label **in** mnist\_dataset:

plt.title(label.numpy())

plt.imshow(image.numpy()[:, :,0])

plt.show()

**break**



**Pandas数据读取**

**import** **pandas** **as** **pd**

df = pd.read\_csv('heart.csv')

df.head()

|  | **age** | **sex** | **cp** | **trestbps** | **chol** | **fbs** | **restecg** | **thalach** | **exang** | **oldpeak** | **slope** | **ca** | **thal** | **target** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 63 | 1 | 1 | 145 | 233 | 1 | 2 | 150 | 0 | 2.3 | 3 | 0 | fixed | 0 |
| **1** | 67 | 1 | 4 | 160 | 286 | 0 | 2 | 108 | 1 | 1.5 | 2 | 3 | normal | 1 |
| **2** | 67 | 1 | 4 | 120 | 229 | 0 | 2 | 129 | 1 | 2.6 | 2 | 2 | reversible | 0 |
| **3** | 37 | 1 | 3 | 130 | 250 | 0 | 0 | 187 | 0 | 3.5 | 3 | 0 | normal | 0 |
| **4** | 41 | 0 | 2 | 130 | 204 | 0 | 2 | 172 | 0 | 1.4 | 1 | 0 | normal | 0 |

df.dtypes

age int64

sex int64

cp int64

trestbps int64

chol int64

fbs int64

restecg int64

thalach int64

exang int64

oldpeak float64

slope int64

ca int64

thal object

target int64

dtype: object

df['thal'] = pd.Categorical(df['thal']).codes

df.head()

|  | **age** | **sex** | **cp** | **trestbps** | **chol** | **fbs** | **restecg** | **thalach** | **exang** | **oldpeak** | **slope** | **ca** | **thal** | **target** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 63 | 1 | 1 | 145 | 233 | 1 | 2 | 150 | 0 | 2.3 | 3 | 0 | 2 | 0 |
| **1** | 67 | 1 | 4 | 160 | 286 | 0 | 2 | 108 | 1 | 1.5 | 2 | 3 | 3 | 1 |
| **2** | 67 | 1 | 4 | 120 | 229 | 0 | 2 | 129 | 1 | 2.6 | 2 | 2 | 4 | 0 |
| **3** | 37 | 1 | 3 | 130 | 250 | 0 | 0 | 187 | 0 | 3.5 | 3 | 0 | 3 | 0 |
| **4** | 41 | 0 | 2 | 130 | 204 | 0 | 2 | 172 | 0 | 1.4 | 1 | 0 | 3 | 0 |

target = df.pop('target')

dataset = tf.data.Dataset.from\_tensor\_slices((df.values, target.values))

**for** feat, targ **in** dataset.take(5):

print ('Features: **{}**, Target: **{}**'.format(feat, targ))

Features: [ 63. 1. 1. 145. 233. 1. 2. 150. 0. 2.3 3. 0. 2. ], Target: 0

Features: [ 67. 1. 4. 160. 286. 0. 2. 108. 1. 1.5 2. 3. 3. ], Target: 1

Features: [ 67. 1. 4. 120. 229. 0. 2. 129. 1. 2.6 2. 2. 4. ], Target: 0

Features: [ 37. 1. 3. 130. 250. 0. 0. 187. 0. 3.5 3. 0. 3. ], Target: 0

Features: [ 41. 0. 2. 130. 204. 0. 2. 172. 0. 1.4 1. 0. 3. ], Target: 0

**从Python generator构建数据管道**

img\_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255, rotation\_range=20)

flowers = './flower\_photos/flower\_photos/'

**def** Gen():

gen = img\_gen.flow\_from\_directory(flowers)

**for** (x,y) **in** gen:

**yield** (x,y)

ds = tf.data.Dataset.from\_generator(

Gen,

output\_types=(tf.float32, tf.float32)

*# output\_shapes=([32,256,256,3], [32,5])*

)

**for** image,label **in** ds:

print(image.shape,label.shape)

**break**

Found 3670 images belonging to 5 classes.

(32, 256, 256, 3) (32, 5)

tf.data.Dataset 类为我们提供了多种数据集预处理方法。最常用的如：  
 tf.data.Dataset.map(f) ：对数据集中的每个元素应用函数 f ，得到一个新的数据集

tf.data.Dataset.shuffle(buffer\_size) ：将数据集打乱；

tf.data.Dataset.batch(batch\_size) ：将数据集分成批次，即对每 batch\_size 个元素，使用 tf.stack() 在第 0 维合并，成为一个元素；

## 5.2.TFRecordDataset类

对于特别巨大而无法完整载入内存的数据集，我们可以先将数据集处理为TFRecord 格式，然后使用 tf.data.TFRecordDataset() 进行载入。  
 TFRecord 是 TensorFlow 中的数据集存储格式。当我们将数据集整理成TFRecord 格式后， TensorFlow 就可以高效地读取和处理这些数据集，从而帮助我们更高效地进行大规模的模型训练。



filenames： tf.string张量，值为一个或多个文件路径。  
 compression\_type： tf.string标量，值为 “” （不压缩）、 "ZLIB"或"GZIP"之一。  
 buffer\_size： tf.int64标量，表示读取缓冲区中的字节数。  
 num\_parallel\_reads： tf.int64标量，表示要并行读取的文件数。

feature\_description = { *# 定义Feature结构，告诉解码器每个Feature的类型是什么*

'image': tf.io.FixedLenFeature([], tf.string),

'label': tf.io.FixedLenFeature([], tf.int64),

}

**def** \_parse\_example(example\_string): *# 将 TFRecord 文件中的每一个序列化的 tf.train.Example 解码*

feature\_dict = tf.io.parse\_single\_example(example\_string, feature\_description)

feature\_dict['image'] = tf.io.decode\_jpeg(feature\_dict['image']) *# 解码JPEG图片*

feature\_dict['image'] = tf.image.resize(feature\_dict['image'], [256, 256]) / 255.0

**return** feature\_dict['image'], feature\_dict['label']

batch\_size = 32

train\_dataset = tf.data.TFRecordDataset("sub\_train.tfrecords") *# 读取 TFRecord 文件*

*# filename label*

train\_dataset = train\_dataset.map(\_parse\_example)

**for** image,label **in** train\_dataset:

print(image,label)

**break**

tf.Tensor(

[[[0.7940257 0.64108455 0.3391238 ]

[0.80168504 0.64874387 0.3467831 ]

...

[0.00392157 0.00784314 0. ]]], shape=(256, 256, 3), dtype=float32) tf.Tensor(0, shape=(), dtype=int64)

## 5.3.TextLineDataset类

tf.data.TextLineDataset简介

tf.data.TextLineDataset 提供了一种从一个或多个文本文件中提取行的简单方法。  
 给定一个或多个文件名， TextLineDataset 会为这些文件的每行生成一个字符串值元素。像 TFRecordDataset 一样， TextLineDataset 将filenames 视为 tf.Tensor。  
 类中保存的元素： 文中一行，就是一个元素，是string类型的tensor。



filenames： tf.string张量，值为一个或多个文件名。  
compression\_type： tf.string标量，值为 “” （不压缩）、 "ZLIB"或"GZIP"之一。  
buffer\_size： tf.int64标量，表示读取缓冲区中的字节数。  
num\_parallel\_reads： tf.int64标量，表示要并行读取的文件数。

titanic\_lines = tf.data.TextLineDataset(['train.csv','eval.csv'])

**def** data\_func(line):

line = tf.strings.split(line, sep = ",")

**return** line

titanic\_data = titanic\_lines.skip(1).map(data\_func)

**for** line **in** titanic\_data:

print(line)

**break**

tf.Tensor(

[b'0' b'male' b'22.0' b'1' b'0' b'7.25' b'Third' b'unknown' b'Southampton'

b'n'], shape=(10,), dtype=string)

## 5.4.猫狗分类

项目网址： https://www.kaggle.com/c/dogs-vs-cats

任务目标 ： C at s v s . D og s（猫狗大战）是Ka gg l e大数据竞赛某一年的一道赛题，利用给定的数据集，用算法实现猫和狗的识别。



**import** **tensorflow** **as** **tf**

**import** **os**

data\_dir = './datasets'

train\_cats\_dir = data\_dir + '/train/cats/'

train\_dogs\_dir = data\_dir + '/train/dogs/'

test\_cats\_dir = data\_dir + '/valid/cats/

'test\_dogs\_dir = data\_dir + '/valid/dogs/'

*# 构建训练数据集*

train\_cat\_filenames = tf.constant([train\_cats\_dir + filename **for** filename **in** os.listdir(train\_cats\_dir)])

train\_dog\_filenames = tf.constant([train\_dogs\_dir + filename **for** filename **in** os.listdir(train\_dogs\_dir)])

train\_filenames = tf.concat([train\_cat\_filenames, train\_dog\_filenames], axis=-1)

*# cat 0 dog :1*

train\_labels = tf.concat([

tf.zeros(train\_cat\_filenames.shape, dtype=tf.int32),

tf.ones(train\_dog\_filenames.shape, dtype=tf.int32)],

axis=-1)

*#构建训练集*

**def** \_decode\_and\_resize(filename, label):

image\_string = tf.io.read\_file(filename) *# 读取原始文件*

image\_decoded = tf.image.decode\_jpeg(image\_string) *# 解码JPEG图片*

image\_resized = tf.image.resize(image\_decoded, [256, 256]) / 255.0

**return** image\_resized, label

batch\_size = 32

train\_dataset = tf.data.Dataset.from\_tensor\_slices((train\_filenames, train\_labels))

train\_dataset = train\_dataset.map(

map\_func=\_decode\_and\_resize,

*#并行机制*

num\_parallel\_calls=tf.data.experimental.AUTOTUNE)

*# 取出前buffer\_size个数据放入buffer，并从其中随机采样，采样后的数据用后续数据替换*

train\_dataset = train\_dataset.shuffle(buffer\_size=23000)

*# 重复3次*

train\_dataset = train\_dataset.repeat(count=3)

*# 批大小*

train\_dataset = train\_dataset.batch(batch\_size)

*# 并行机制*

train\_dataset = train\_dataset.prefetch(tf.data.experimental.AUTOTUNE)

*# 构建测试数据集*

test\_cat\_filenames = tf.constant([test\_cats\_dir + filename **for** filename **in** os.listdir(test\_cats\_dir)])

test\_dog\_filenames = tf.constant([test\_dogs\_dir + filename **for** filename **in** os.listdir(test\_dogs\_dir)])

test\_filenames = tf.concat([test\_cat\_filenames, test\_dog\_filenames], axis=-1)

test\_labels = tf.concat([

tf.zeros(test\_cat\_filenames.shape, dtype=tf.int32),

tf.ones(test\_dog\_filenames.shape, dtype=tf.int32)],

axis=-1)

test\_dataset = tf.data.Dataset.from\_tensor\_slices((test\_filenames, test\_labels))

test\_dataset = test\_dataset.map(\_decode\_and\_resize)

test\_dataset = test\_dataset.batch(batch\_size)

**class** **CNNModel**(tf.keras.models.Model):

**def** \_\_init\_\_(self):

super(CNNModel, self).\_\_init\_\_()

self.conv1 = tf.keras.layers.Conv2D(32, 3, activation='relu')

self.maxpool1 = tf.keras.layers.MaxPooling2D()

self.conv2 = tf.keras.layers.Conv2D(32, 5, activation='relu')

self.maxpool2 = tf.keras.layers.MaxPooling2D()

self.flatten = tf.keras.layers.Flatten()

self.d1 = tf.keras.layers.Dense(64, activation='relu')

self.d2 = tf.keras.layers.Dense(2, activation='softmax') *#sigmoid 和softmax*

**def** call(self, x):

x = self.conv1(x)

x = self.maxpool1(x)

x = self.conv2(x)

x = self.maxpool2(x)

x = self.flatten(x)

x = self.d1(x)

x = self.d2(x)

**return** x

learning\_rate = 0.001

model = CNNModel()

*#label 没有one-hot*

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy()

*#优化器*

optimizer = tf.keras.optimizers.Adam(learning\_rate=learning\_rate)

*#评估函数*

train\_loss = tf.keras.metrics.Mean(name='train\_loss')

train\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train\_accuracy')

test\_loss = tf.keras.metrics.Mean(name='test\_loss')

test\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test\_accuracy')

@tf.function

**def** train\_step(images, labels):

**with** tf.GradientTape() **as** tape:

predictions = model(images)

loss = loss\_object(labels, predictions)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

train\_loss(loss)

train\_accuracy(labels, predictions)

**def** test\_step(images, labels):

predictions = model(images)

t\_loss = loss\_object(labels, predictions)

test\_loss(t\_loss)

test\_accuracy(labels, predictions)

EPOCHS=10

**for** epoch **in** range(EPOCHS):

*# 在下一个epoch开始时，重置评估指标*

train\_loss.reset\_states()

train\_accuracy.reset\_states()

test\_loss.reset\_states()

test\_accuracy.reset\_states()

**for** images, labels **in** train\_dataset:

train\_step(images, labels)

**for** test\_images, test\_labels **in** test\_dataset:

test\_step(test\_images, test\_labels)

template = 'Epoch **{}**, Loss: **{}**, Accuracy: **{}**, Test Loss: **{}**, Test Accuracy: **{}**'

print(template.format(epoch + 1,

train\_loss.result(),

train\_accuracy.result() \* 100,

test\_loss.result(),

test\_accuracy.result() \* 100

))

Epoch 1, Loss: 0.6417950987815857, Accuracy: 63.08551025390625, Test Loss: 0.5889803767204285, Test Accuracy: 69.9000015258789

...

Epoch 10, Loss: 0.008066367357969284, Accuracy: 99.8086929321289, Test Loss: 2.8660709857940674, Test Accuracy: 67.4000015258789

## 5.5.TFRecord

TFRecord 是Google官方推荐的一种数据格式 ，是 Google专门为TensorFlow 设计的一种数据格式 。  
 实际上， TFRecord 是一种二进制文件 ，其能更好的利用内存 ，其内部包含了多个tf.train.Example，而Example是protocol buffer数据标准的实现，在一个Exampl e消息体中包含了一系列的 tf . train.feature 属性，而 每一个feature 是一个key -value的键值对，其中， key是string类型，而value 的取值有三种：

• bytes\_list: 可以存储string 和byte两种数据类型 。

bytes\_list ： tf.train.Feature(bytes\_list=tf.train.BytesList(value=输入))   
• float\_list: 可以存储float(float32) 与double(float64) 两种数据类型 。

float\_list: tf.train.Feature(float\_list = tf.train.FloatList(value=输入))   
• int64\_list: 可以存储： bool,enum,int32,uint32,int64,uint64。

int64\_list: tf.train.Feature(int64\_list = tf.train.Int64List(value=输入))

TFRecord 并非是TensorFlow唯一支持的数据格式，你也可以使用CSV或文本等格式，但是对于TensorFlow来说， TFRecord 是最友好的，也是最方便的。前面提到，TFRecord内部是一系列实现了protocol buffer数据标准的Example。对于大型数据，相比其余数据格式，protocol buffer类型的数据优势很明显。  
 在数据集较小时，我们会把数据全部加载到内存里方便快速导入，但当数据量超过内存大小时，就只能放在硬盘上来一点点读取，这时就不得不考虑数据的移动、读取、处理等速度。使用TFRecord就是为了提速和节约空间的。

TFRecord 可以理解为一系列序列化的tf.train.Example 元素所组成的列表文件，而每一个 tf.train.Example 又由若干个 tf.train.Feature 的字典组成。

[  
 { # example 1 (tf.train.Example)  
 'feature\_1': tf.train.Feature,  
 ...  
 'feature\_k': tf.train.Feature  
 },  
 ...  
 { # example N (tf.train.Example)  
 'feature\_1': tf.train.Feature,  
 ...  
 'feature\_k': tf.train.Feature  
 }  
 ]

**生成TFRecord格式数据**

为了将形式各样的数据集整理为 TFRecord 格式，我们可以对数据集中的每个元素进行以下步骤：  
 • 读取该数据元素到内存；  
 • 建立 Feature 的字典；  
 • 将该元素转换为tf.train.Example对象（每一个tf.train.Example由若干个tf.train.Feature 的字典组成）；  
 • 将该tf.train.Example对象序列化为字符串，并通过一个预先定义的tf.io.TFRecordWriter 写入 TFRecord 文件。

Cats vs dogs数据集示例



**读取TFRecord文件**

读取 TFRecord 数据则可按照以下步骤：  
 • 通过tf.data.TFRecordDataset读入原始的TFRecord文件（此时文件中的tf.train.Example 对象尚未被反序列化），获得一个 tf.data.Dataset 数据集对象；  
 • 定义Feature结构，告诉解码器每个Feature的类型是什么；  
 • 通过 Dataset.map 方法，对该数据集对象中的每一个序列化的tf.train.Example 字符串执行 tf.io.parse\_single\_example 函数，从而实现反序列化。



Cats vs. Dogs比赛项目为例：

**import** **tensorflow** **as** **tf**

**import** **os**

data\_dir = './datasets'

train\_cats\_dir = data\_dir + '/train/cats/'

train\_dogs\_dir = data\_dir + '/train/dogs/'

train\_tfrecord\_file = data\_dir + '/train/train.tfrecords'

test\_cats\_dir = data\_dir + '/valid/cats/'

test\_dogs\_dir = data\_dir + '/valid/dogs/'

test\_tfrecord\_file = data\_dir + '/valid/test.tfrecords'

将数据集存储为 TFRecord 文件

train\_cat\_filenames = [train\_cats\_dir + filename **for** filename **in** os.listdir(train\_cats\_dir)]

train\_dog\_filenames = [train\_dogs\_dir + filename **for** filename **in** os.listdir(train\_dogs\_dir)]

train\_filenames = train\_cat\_filenames + train\_dog\_filenames

*# 将 cat 类的标签设为0，dog 类的标签设为1*

train\_labels = [0] \* len(train\_cat\_filenames) + [1] \* len(train\_dog\_filenames)

**with** tf.io.TFRecordWriter(train\_tfrecord\_file) **as** writer:

**for** filename, label **in** zip(train\_filenames, train\_labels):

image = open(filename, 'rb').read() *# 读取数据集图片到内存，image 为一个 Byte 类型的字符串*

feature = { *# 建立 tf.train.Feature 字典*

'image': tf.train.Feature(bytes\_list=tf.train.BytesList(value=[image])), *# 图片是一个 Bytes 对象*

'label': tf.train.Feature(int64\_list=tf.train.Int64List(value=[label])) *# 标签是一个 Int 对象*

}

example = tf.train.Example(features=tf.train.Features(feature=feature)) *# 通过字典建立 Example*

writer.write(example.SerializeToString()) *# 将Example序列化并写入 TFRecord 文件*

test\_cat\_filenames = [test\_cats\_dir + filename **for** filename **in** os.listdir(test\_cats\_dir)]

test\_dog\_filenames = [test\_dogs\_dir + filename **for** filename **in** os.listdir(test\_dogs\_dir)]

test\_filenames = test\_cat\_filenames + test\_dog\_filenames

*# 将 cat 类的标签设为0，dog 类的标签设为1*

test\_labels = [0] \* len(test\_cat\_filenames) + [1] \* len(test\_dog\_filenames)

**with** tf.io.TFRecordWriter(test\_tfrecord\_file) **as** writer:

**for** filename, label **in** zip(test\_filenames, test\_labels):

image = open(filename, 'rb').read() *# 读取数据集图片到内存，image 为一个 Byte 类型的字符串*

feature = { *# 建立 tf.train.Feature 字典*

'image': tf.train.Feature(bytes\_list=tf.train.BytesList(value=[image])), *# 图片是一个 Bytes 对象*

'label': tf.train.Feature(int64\_list=tf.train.Int64List(value=[label])) *# 标签是一个 Int 对象*

}

example = tf.train.Example(features=tf.train.Features(feature=feature)) *# 通过字典建立 Example*

serialized = example.SerializeToString() *#将Example序列化*

writer.write(serialized) *# 写入 TFRecord 文件*

读取 TFRecord 文件

train\_dataset = tf.data.TFRecordDataset(train\_tfrecord\_file) *# 读取 TFRecord 文件*

feature\_description = { *# 定义Feature结构，告诉解码器每个Feature的类型是什么*

'image': tf.io.FixedLenFeature([], tf.string),

'label': tf.io.FixedLenFeature([], tf.int64),

}

**def** \_parse\_example(example\_string): *# 将 TFRecord 文件中的每一个序列化的 tf.train.Example 解码*

feature\_dict = tf.io.parse\_single\_example(example\_string, feature\_description)

feature\_dict['image'] = tf.io.decode\_jpeg(feature\_dict['image']) *# 解码JPEG图片*

feature\_dict['image'] = tf.image.resize(feature\_dict['image'], [256, 256]) / 255.0

**return** feature\_dict['image'], feature\_dict['label']

train\_dataset = train\_dataset.map(\_parse\_example)

batch\_size = 32

train\_dataset = train\_dataset.shuffle(buffer\_size=23000)

train\_dataset = train\_dataset.batch(batch\_size)

train\_dataset = train\_dataset.prefetch(tf.data.experimental.AUTOTUNE)

test\_dataset = tf.data.TFRecordDataset(test\_tfrecord\_file) *# 读取 TFRecord 文件*

test\_dataset = test\_dataset.map(\_parse\_example)

test\_dataset = test\_dataset.batch(batch\_size)

**class** **CNNModel**(tf.keras.models.Model):

**def** \_\_init\_\_(self):

super(CNNModel, self).\_\_init\_\_()

self.conv1 = tf.keras.layers.Conv2D(32, 3, activation='relu')

self.maxpool1 = tf.keras.layers.MaxPooling2D()

self.conv2 = tf.keras.layers.Conv2D(32, 5, activation='relu')

self.maxpool2 = tf.keras.layers.MaxPooling2D()

self.flatten = tf.keras.layers.Flatten()

self.d1 = tf.keras.layers.Dense(64, activation='relu')

self.d2 = tf.keras.layers.Dense(2, activation='softmax')

**def** call(self, x):

x = self.conv1(x)

x = self.maxpool1(x)

x = self.conv2(x)

x = self.maxpool2(x)

x = self.flatten(x)

x = self.d1(x)

x = self.d2(x)

**return** x

learning\_rate = 0.001

model = CNNModel()

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy()

optimizer = tf.keras.optimizers.Adam(learning\_rate=learning\_rate)

train\_loss = tf.keras.metrics.Mean(name='train\_loss')

train\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train\_accuracy')

test\_loss = tf.keras.metrics.Mean(name='test\_loss')

test\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test\_accuracy')

*#batch*

@tf.function

**def** train\_step(images, labels):

**with** tf.GradientTape() **as** tape:

predictions = model(images)

loss = loss\_object(labels, predictions)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

train\_loss(loss) *#update*

train\_accuracy(labels, predictions)*#update*

@tf.function

**def** test\_step(images, labels):

predictions = model(images)

t\_loss = loss\_object(labels, predictions)

test\_loss(t\_loss)

test\_accuracy(labels, predictions)

EPOCHS=10

**for** epoch **in** range(EPOCHS):

*# 在下一个epoch开始时，重置评估指标*

train\_loss.reset\_states()

train\_accuracy.reset\_states()

test\_loss.reset\_states()

test\_accuracy.reset\_states()

**for** images, labels **in** train\_dataset:

train\_step(images, labels) *#mini-batch 更新*

**for** test\_images, test\_labels **in** test\_dataset:

test\_step(test\_images, test\_labels)

template = 'Epoch **{}**, Loss: **{}**, Accuracy: **{}**, Test Loss: **{}**, Test Accuracy: **{}**'

print(template.format(epoch + 1,

train\_loss.result(),

train\_accuracy.result() \* 100,

test\_loss.result(),

test\_accuracy.result() \* 100

))

Epoch 1, Loss: 0.6762517690658569, Accuracy: 62.32608413696289, Test Loss: 0.6289066076278687, Test Accuracy: 64.6500015258789

...

Epoch 10, Loss: 0.015399742871522903, Accuracy: 99.49130249023438, Test Loss: 2.065403461456299, Test Accuracy: 70.1500015258789