# 4.模型的自定义

## 4.1.自定义层

使用的主要数据结构是Layer  
实现自定义层的最佳方法是扩展tf.keras.layers.Layer类并实现：  
 • \_\_init\_\_ ：可以在其中进行所有与输入无关的初始化，定义相关的层  
 • build： 知道输入张量的形状并可以进行其余的初始化  
 • call： 在这里进行前向传播  
 注意：不一定需要在build中创建变量时，也可以在\_\_init\_\_中创建它们。

tf.keras.Model和tf.keras.layers.Layer有什么区别和联系？  
 • 通过继承 tf.keras.Model 编写自己的模型类  
 • 通过继承 tf.keras.layers.Layer 编写自己的层  
 • tf.keras中的模型和层都是继承tf.Module实现的  
 • tf.keras.Model继承tf.keras.layers.Layer实现的

tf.Module： 定位为一个轻量级的状态容器，因为可以收集变量，所以这个类型可以用来建模，配合tf.GradientTape使用。

**自定义一个线性回归模型**

使用鸢尾花数据集

**from** **sklearn** **import** datasets

iris = datasets.load\_iris()

data = iris.data

target = iris.target

data.shape *# x*

(150, 4)

target.shape *#y*

(150,)

**方法1：最基础的方法**

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

*#自定义全连接层*

**class** **Linear**(tf.keras.layers.Layer):

**def** \_\_init\_\_(self, units=1, input\_dim=4):

super(Linear, self).\_\_init\_\_() *#*

w\_init = tf.random\_normal\_initializer()

self.w = tf.Variable(initial\_value=w\_init(shape=(input\_dim, units), dtype='float32'), trainable=**True**)

b\_init = tf.zeros\_initializer()

self.b = tf.Variable(initial\_value=b\_init(shape=(units,),dtype='float32'),trainable=**True**)

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

**return** tf.matmul(inputs, self.w) + self.b

x = tf.constant(data) *#(150,4)*

linear\_layer = Linear(units = 1, input\_dim=4) *#()*

y = linear\_layer(x)

print(y.shape) *#(150,1)*

(150, 1)

**方法2：使用self.add\_weight创建变量**

**class** **Linear**(tf.keras.layers.Layer):

**def** \_\_init\_\_(self, units=1, input\_dim=4):

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

self.w = self.add\_weight(shape=(input\_dim, units),

initializer='random\_normal',

trainable=**True**)

self.b = self.add\_weight(shape=(units,),

initializer='zeros',

trainable=**True**)

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

**return** tf.matmul(inputs, self.w) + self.b

x = tf.constant(data)

linear\_layer = Linear(units = 1, input\_dim=4)

y = linear\_layer(x)

print(y.shape)

(150, 1)

**方法三：build函数中创建变量**

**class** **Linear**(tf.keras.layers.Layer):

**def** \_\_init\_\_(self, units=32):

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

self.units = units

**def** build(self, input\_shape): *#(150,4)*

self.w = self.add\_weight(shape=(input\_shape[-1], self.units),

initializer='random\_normal',

trainable=**True**)

self.b = self.add\_weight(shape=(self.units,),

initializer='random\_normal',

trainable=**True**)

super(Linear,self).build(input\_shape)

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

**return** tf.matmul(inputs, self.w) + self.b

x = tf.constant(data) *#150\*4*

linear\_layer = Linear(units = 1)

y = linear\_layer(x)

print(y.shape)

(150, 1)

**添加不可训练的参数**

**class** **Linear**(tf.keras.layers.Layer):

**def** \_\_init\_\_(self, units=32):

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

self.units = units

**def** build(self, input\_shape):

self.w = self.add\_weight(shape=(input\_shape[-1], self.units),

initializer='random\_normal',

trainable=**True**)

self.b = self.add\_weight(shape=(self.units,),

initializer='random\_normal',

trainable=**False**)

super(Linear,self).build(input\_shape)

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

**return** tf.matmul(inputs, self.w) + self.b

x = tf.constant(data)

linear\_layer = Linear(units = 1)

y = linear\_layer(x)

print(y.shape)

(150, 1)

*# 打印所有参数、不可训练参数、可训练参数*

print('weight:', linear\_layer.weights)

print('non-trainable weight:', linear\_layer.non\_trainable\_weights)

print('trainable weight:', linear\_layer.trainable\_weights)

weight: [<tf.Variable 'linear\_4/Variable:0' shape=(4, 1) dtype=float32, numpy=

array([[ 0.00276536],

[-0.07950259],

[ 0.01646506],

[ 0.00197834]], dtype=float32)>, <tf.Variable 'linear\_4/Variable:0' shape=(1,) dtype=float32, numpy=array([0.00255805], dtype=float32)>]

non-trainable weight: [<tf.Variable 'linear\_4/Variable:0' shape=(1,) dtype=float32, numpy=array([0.00255805], dtype=float32)>]

trainable weight: [<tf.Variable 'linear\_4/Variable:0' shape=(4, 1) dtype=float32, numpy=

array([[ 0.00276536],

[-0.07950259],

[ 0.01646506],

[ 0.00197834]], dtype=float32)>]

**自定义层的注意事项**

如果需要保存模型，则在自定义网络层时需要重写get\_config 方法

我们主要看传入\_\_init\_\_接口时有哪些配置参数，然后在get\_config内一一的将它们转为字典键值并且返回使用

get\_config的作用：获取该层的参数配置，以便模型保存时使用

自定义层的biuld 中创建初始矩阵时， 需要添加name属性

我们在实现自定义网络层时，最好统一在初始化时传入可变参数\*\*kwargs，这是因为在model推理时，有时我们需要对所有构成该模型的网络层进行统一的传参。

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

*#Dense*

**class** **MyDense**(tf.keras.layers.Layer):

**def** \_\_init\_\_(self, units=32, \*\*kwargs):

self.units = units

super(MyDense, self).\_\_init\_\_(\*\*kwargs)

*#build方法一般定义Layer需要被训练的参数。*

**def** build(self, input\_shape):

self.w = self.add\_weight(shape=(input\_shape[-1], self.units),

initializer='random\_normal',

trainable=**True**,

name='w')

self.b = self.add\_weight(shape=(self.units,),

initializer='random\_normal',

trainable=**True**,

name='b')

super(MyDense,self).build(input\_shape) *# 相当于设置self.built = True*

*#call方法一般定义正向传播运算逻辑，\_\_call\_\_方法调用了它。*

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

**return** tf.matmul(inputs, self.w) + self.b

*#如果要让自定义的Layer可以序列化，需要自定义get\_config方法。*

**def** get\_config(self):

config = super(MyDense, self).get\_config()

config.update({'units': self.units})

**return** config

**from** **sklearn** **import** datasets

iris = datasets.load\_iris()

data = iris.data

labels = iris.target

*#网络 函数式构建的网络*

inputs = tf.keras.Input(shape=(4,))

x = MyDense(units=16)(inputs)

x = tf.nn.tanh(x)

x = MyDense(units=3)(x) *#0,1,2*

predictions = tf.nn.softmax(x)

model = tf.keras.Model(inputs=inputs, outputs=predictions)

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

data = np.concatenate((data,labels.reshape(150,1)),axis=-1)

np.random.shuffle(data)

labels = data[:,-1]

data = data[:,:4]

*#优化器 Adam*

*#损失函数 交叉熵损失函数*

*#评估函数 #acc*

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=**True**),

metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

*#keras*

model.fit(data, labels, batch\_size=32, epochs=100,shuffle=**True**)

Train on 150 samples

Epoch 1/100

150/150 [==============================] - 1s 7ms/sample - loss: 1.0939 - sparse\_categorical\_accuracy: 0.5333

...

Epoch 100/100

150/150 [==============================] - 0s 133us/sample - loss: 0.6751 - sparse\_categorical\_accuracy: 0.9733

model.save('keras\_model\_tf\_version.h5')

当我们自定义网络层并且有效保存模型后，希望使用tf.keras.models.load\_model进行模型加载时， 首先，建立一个字典，该字典的键是自定义网络层时设定该层的名字，其值为  
该自定义网络层的类名，该字典将用于加载模型时使用！  
 然后，在tf.keras.models.load\_model内传入custom\_objects告知如何解析重建自定义网络层。

\_custom\_objects = {

"MyDense" : MyDense,

}

new\_model = tf.keras.models.load\_model("keras\_model\_tf\_version.h5",custom\_objects=\_custom\_objects)

y\_pred = new\_model.predict(data)

np.argmax(y\_pred,axis=1)

array([2, 1, 0, 0, 0, 2, 1, 0, 0, 0, 2, 0, 2, 0, 0, 2, 0, 0, 2, 2, 0, 0,

0, 2, 0, 0, 2, 0, 2, 0, 2, 2, 2, 1, 2, 2, 2, 0, 1, 1, 1, 1, 2, 1,

2, 2, 1, 0, 2, 0, 2, 1, 1, 1, 2, 2, 1, 1, 0, 0, 1, 0, 1, 1, 2, 1,

1, 2, 2, 0, 1, 1, 0, 2, 0, 1, 0, 0, 2, 2, 1, 1, 2, 2, 1, 2, 0, 0,

1, 0, 2, 2, 0, 2, 0, 1, 2, 1, 1, 2, 2, 0, 1, 0, 1, 0, 0, 1, 0, 1,

1, 0, 2, 2, 1, 2, 1, 0, 1, 1, 0, 2, 1, 0, 0, 1, 0, 2, 2, 1, 1, 2,

1, 2, 1, 2, 2, 0, 2, 2, 0, 2, 2, 2, 2, 0, 1, 0, 1, 0], dtype=int64)

labels

array([2., 1., 0., 0., 0., 2., 1., 0., 0., 0., 2., 0., 2., 0., 0., 2., 0.,

0., 2., 2., 0., 0., 0., 2., 0., 0., 1., 0., 2., 0., 2., 2., 2., 1.,

1., 2., 2., 0., 1., 1., 1., 1., 2., 1., 2., 2., 1., 0., 2., 0., 2.,

1., 1., 1., 2., 2., 1., 1., 0., 0., 1., 0., 1., 1., 2., 1., 1., 2.,

2., 0., 1., 1., 0., 2., 0., 1., 0., 0., 2., 2., 1., 1., 2., 2., 1.,

2., 0., 0., 1., 0., 2., 2., 0., 2., 0., 1., 2., 1., 1., 2., 2., 0.,

1., 0., 1., 0., 0., 1., 0., 1., 1., 0., 2., 2., 1., 2., 1., 0., 1.,

1., 0., 2., 1., 0., 0., 1., 0., 2., 2., 1., 1., 1., 1., 1., 1., 2.,

2., 0., 2., 2., 0., 2., 2., 2., 2., 0., 1., 0., 1., 0.])

## 4.2.损失函数

常用损失函数：

• mean\_squared\_error（平方差误差损失，用于回归，简写为 mse, 类实现形式为MeanSquaredError 和 MSE）  
 • binary\_crossentropy(二元交叉熵，用于二分类，类实现形式为 BinaryCrossentropy)  
 • categorical\_crossentropy(类别交叉熵，用于多分类，要求label为onehot编码，类实现形式为 CategoricalCrossentropy)  
 • sparse\_categorical\_crossentropy(稀疏类别交叉熵，用于多分类，要求label为序号编码形式，类实现形式为 SparseCategoricalCrossentropy)

自定义损失函数，两种方法自定义函数：

函数的实现形式

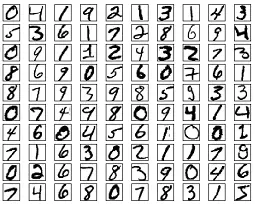
def MeanSquaredError(y\_true, y\_pred):  
 return tf.reduce\_mean(tf.square(y\_pred - y\_true))

类的实现形式：

class MeanSquaredError(tf.keras.losses.Loss):  
 def call(self, y\_true, y\_pred):  
 return tf.reduce\_mean(tf.square(y\_pred - y\_true))

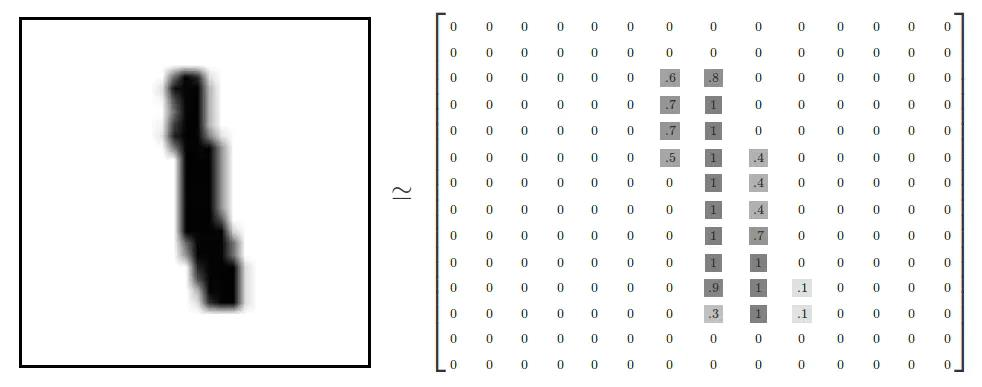
MNIST数据集案例

MNIST是一个入门级的计算机视觉数据集，它包含各种手写数字图片，如图：



它也包含每一张图片对应的标签，告诉我们这个是数字几。比如，第一行这10张图片的标签分别是0， 4， 1， 9， 2，1， 3， 1， 4， 3。

每一张图都是由（28， 28， 1）的矩阵组成：



**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function, unicode\_literals

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

**from** **tensorflow.keras.layers** **import** Dense, Flatten, Conv2D

**from** **tensorflow.keras** **import** Model

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

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

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

x\_train.shape

(60000, 28, 28)

x\_test.shape

(10000, 28, 28)

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

*# 数据可视化*

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

fig, ax = plt.subplots(

nrows=2,

ncols=5,

sharex=**True**,

sharey=**True**, )

ax = ax.flatten()

**for** i **in** range(10):

img = x\_train[y\_train == i][0].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()



*# Add a channels dimension*

x\_train = x\_train[..., tf.newaxis]

x\_test = x\_test[..., tf.newaxis]

y\_train = tf.one\_hot(y\_train,depth=10)

y\_test = tf.one\_hot(y\_test,depth=10)

train\_ds = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train)).shuffle(10000).batch(32)

test\_ds = tf.data.Dataset.from\_tensor\_slices((x\_test, y\_test)).batch(32)

**class** **MyModel**(Model):

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

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

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

self.flatten = Flatten()

self.d1 = Dense(128, activation='relu')

self.d2 = Dense(10, activation='softmax')

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

x = self.conv1(x)

x = self.flatten(x)

x = self.d1(x)

**return** self.d2(x)

model = MyModel()

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

optimizer = tf.keras.optimizers.Adam()

*# 选择衡量指标来度量模型的损失值（loss）和准确率（accuracy）*

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

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

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

test\_accuracy = tf.keras.metrics.CategoricalAccuracy(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)

@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 = 5

**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\_ds:

train\_step(images, labels)

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

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: 1.0062854290008545, Accuracy: 88.1866683959961, Test Loss: 0.899224042892456, Test Accuracy: 97.05000305175781

Epoch 2, Loss: 0.8931533098220825, Accuracy: 97.54166412353516, Test Loss: 0.8904874920845032, Test Accuracy: 97.68000030517578

Epoch 3, Loss: 0.8832218647003174, Accuracy: 98.30166625976562, Test Loss: 0.8857290744781494, Test Accuracy: 98.02999877929688

Epoch 4, Loss: 0.879651665687561, Accuracy: 98.54666900634766, Test Loss: 0.8816375732421875, Test Accuracy: 98.4000015258789

Epoch 5, Loss: 0.8753460049629211, Accuracy: 98.89666748046875, Test Loss: 0.8853973746299744, Test Accuracy: 98.06999969482422

## 4.3.评估函数

常用评估函数：

回归相关评估函数

• tf.keras.metrics.MeanSquaredError （平方差误差，用于回归，可以简写为MSE，函数形式为mse）  
 • tf.keras.metrics.MeanAbsoluteError (绝对值误差，用于回归，可以简写为MAE，函数形式为mae)  
 • tf.keras.metrics.MeanAbsolutePercentageError (平均百分比误差，用于回归，可以简写为MAPE，函数形式为mape)  
 • tf.keras.metrics.RootMeanSquaredError (均方根误差，用于回归)

分类相关评估函数

• tf.keras.metrics.Accuracy (准确率，用于分类，可以用字符串"Accuracy"表示，Accuracy=(TP+TN)/(TP+TN+FP+FN)，要求y\_true和y\_pred都为类别序号编码)  
 • tf.keras.metrics.AUC (ROC曲线(TPR vs FPR)下的面积，用于二分类，直观解释为随机抽取一个正样本和一个负样本，正样本的预测值大于负样本的概率)  
 • tf.keras.metrics.Precision (精确率，用于二分类， Precision = TP/(TP+FP))  
 • tf.keras.metrics.Recall (召回率，用于二分类， Recall = TP/(TP+FN))  
 • tf.keras.metrics.TopKCategoricalAccuracy(多分类TopK准确率，要求y\_true(label)为onehot编码形式)

• tf.keras.metrics.CategoricalAccuracy（分类准确率，与Accuracy含义相同，要求y\_true(label)为onehot编码形式）  
 • tf.keras.metrics. SparseCategoricalAccuracy (稀疏分类准确率，与Accuracy含义相同，要求y\_true(label)为序号编码形式)

更多参考：  
 https://www.tensorflow.org/versions/r2.0/api\_docs/python/tf/keras/metrics

案例：

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

m=tf.keras.metrics.Accuracy()

m.update\_state([1,2,3,4],[0,2,3,4])

*# 可简写为*

*# m([1,2,3,4],[0,2,3,1])*

print('Final result: ',m.result().numpy())*# Final result: 0.75*

m.update\_state([1,2,3,4],[0,2,3,1])

print('Final result: ',m.result().numpy())

Final result:  0.75

Final result:  0.625

m.reset\_states()

m.update\_state([1,2,3,4],[0,2,3,4])

print('Final result: ',m.result().numpy())

Final result:  0.75

**自定义评估函数**

两种实现形式： 基于类的实现和基于函数的实现，大部分使用基于类的实现

自定义评估指标需要继承 tf.keras.metrics.Metric 类，并重写 \_\_init\_\_ 、update\_state 和 result 三个方法。  
 • \_\_init\_\_():所有状态变量都应通过以下方法在此方法中创建self.add\_weight()  
 • update\_state(): 对状态变量进行所有更新  
 • result(): 根据状态变量计算并返回指标值。  
案例：

**class** **SparseCategoricalAccuracy\_**(tf.keras.metrics.Metric):

**def** \_\_init\_\_(self, name='SparseCategoricalAccuracy', \*\*kwargs):

super(SparseCategoricalAccuracy\_, self).\_\_init\_\_(name=name, \*\*kwargs)

self.total = self.add\_weight(name='total', dtype=tf.int32, initializer=tf.zeros\_initializer())

self.count = self.add\_weight(name='count', dtype=tf.int32, initializer=tf.zeros\_initializer())

**def** update\_state(self, y\_true, y\_pred,sample\_weight=**None**):

values = tf.cast(tf.equal(y\_true, tf.argmax(y\_pred, axis=-1, output\_type=tf.int32)), tf.int32)

self.total.assign\_add(tf.shape(y\_true)[0])

self.count.assign\_add(tf.reduce\_sum(values))

**def** result(self):

**return** self.count / self.total

**def** reset\_states(self):

*# The state of the metric will be reset at the start of each epoch.*

self.total.assign(0)

self.count.assign(0)

*# 利用自定义评估函数进行评估*

s = SparseCategoricalAccuracy\_()

*# s.reset\_states()*

s.update\_state(tf.constant([2, 1]), tf.constant([[0.1, 0.9, 0.8], [0.05, 0.95, 0]]))

print('Final result: ', s.result().numpy()) *# Final result: 0.5*

Final result: 0.5

*# 利用官方评估函数进行评估*

m = tf.keras.metrics.SparseCategoricalAccuracy()

m.update\_state([2,1], [[0.1, 0.9, 0.8], [0.05, 0.95, 0]])

print('Final result: ', m.result().numpy()) *# Final result: 0.5*

Final result: 0.5

**class** **CatgoricalTruePositives**(tf.keras.metrics.Metric):

**def** \_\_init\_\_(self, name='categorical\_true\_positives', \*\*kwargs):

super(CatgoricalTruePositives, self).\_\_init\_\_(name=name, \*\*kwargs)

self.true\_positives = self.add\_weight(name='tp', initializer='zeros')

**def** update\_state(self, y\_true, y\_pred, sample\_weight=**None**):

y\_pred = tf.argmax(y\_pred,axis=-1)

values = tf.equal(tf.cast(y\_true, 'int32'), tf.cast(y\_pred, 'int32'))

values = tf.cast(values, 'float32')

**if** sample\_weight **is** **not** **None**:

sample\_weight = tf.cast(sample\_weight, 'float32')

values = tf.multiply(values, sample\_weight)

self.true\_positives.assign\_add(tf.reduce\_sum(values))

**def** result(self):

**return** self.true\_positives

**def** reset\_states(self):

*# The state of the metric will be reset at the start of each epoch.*

self.true\_positives.assign(0.)

y\_pred = tf.nn.softmax(tf.random.uniform((4,3)))

tf.argmax(y\_pred,axis=-1)

<tf.Tensor: id=19, shape=(4,), dtype=int64, numpy=array([1, 2, 1, 0], dtype=int64)>

y\_true = tf.constant([2,0,0,0])

m=CatgoricalTruePositives()

m.update\_state(y\_true,y\_pred)

print('Final result: ',m.result().numpy())

Final result:  1.0

**自定义评估函数在MNIST数据集中的使用**

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function, unicode\_literals

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

**from** **tensorflow.keras.layers** **import** Dense, Flatten, Conv2D

**from** **tensorflow.keras** **import** Model

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

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

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

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

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

fig, ax = plt.subplots(

nrows=2,

ncols=5,

sharex=**True**,

sharey=**True**, )

ax = ax.flatten()

**for** i **in** range(10):

img = x\_train[y\_train == i][0].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()



*# Add a channels dimension*

x\_train = x\_train[..., tf.newaxis]

x\_test = x\_test[..., tf.newaxis]

train\_ds = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train)).shuffle(10000).batch(32)

test\_ds = tf.data.Dataset.from\_tensor\_slices((x\_test, y\_test)).batch(32)

**class** **MyModel**(Model):

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

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

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

self.flatten = Flatten()

self.d1 = Dense(128, activation='relu')

self.d2 = Dense(10, activation='softmax')

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

x = self.conv1(x)

x = self.flatten(x)

x = self.d1(x)

**return** self.d2(x)

*#返回的是一个正确的个数*

**class** **CatgoricalTruePositives**(tf.keras.metrics.Metric):

**def** \_\_init\_\_(self, name='categorical\_true\_positives', \*\*kwargs):

super(CatgoricalTruePositives, self).\_\_init\_\_(name=name, \*\*kwargs)

self.true\_positives = self.add\_weight(name='tp', initializer='zeros')

**def** update\_state(self, y\_true, y\_pred, sample\_weight=**None**):

y\_pred = tf.argmax(y\_pred,axis=-1)

values = tf.equal(tf.cast(y\_true, 'int32'), tf.cast(y\_pred, 'int32'))

values = tf.cast(values, 'float32')

**if** sample\_weight **is** **not** **None**:

sample\_weight = tf.cast(sample\_weight, 'float32')

values = tf.multiply(values, sample\_weight)

self.true\_positives.assign\_add(tf.reduce\_sum(values))

**def** result(self):

**return** self.true\_positives

**def** reset\_states(self):

self.true\_positives.assign(0.)

model = MyModel()

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy() *#损失函数*

optimizer = tf.keras.optimizers.Adam() *#优化器*

*#评估函数*

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

train\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train\_accuracy') *#准确率*

train\_tp = CatgoricalTruePositives(name="train\_tp") *#返回正确的个数*

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

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

test\_tp = CatgoricalTruePositives(name='test\_tp')

@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))

*#评估函数的结果*

train\_loss(loss)

train\_accuracy(labels, predictions)

train\_tp(labels, predictions)

@tf.function

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

predictions = model(images)

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

test\_loss(t\_loss)

test\_accuracy(labels, predictions)

test\_tp(labels, predictions)

EPOCHS = 5

**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()

train\_tp.reset\_states()

test\_loss.reset\_states()

test\_accuracy.reset\_states()

test\_tp.reset\_states()

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

train\_step(images, labels)

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

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

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

print(template.format(epoch + 1,

train\_loss.result(),

train\_accuracy.result() \* 100,

train\_tp.result(),

test\_loss.result(),

test\_accuracy.result() \* 100,

test\_tp.result()))

Epoch 1, Loss: 0.1417243927717209, Accuracy: 95.7750015258789, TP: 57465.0,Test Loss: 0.06078110635280609, Test Accuracy: 97.94999694824219, Test TP:9795.0

Epoch 2, Loss: 0.04365191608667374, Accuracy: 98.64833068847656, TP: 59189.0,Test Loss: 0.052343666553497314, Test Accuracy: 98.29000091552734, Test TP:9829.0

Epoch 3, Loss: 0.023991659283638, Accuracy: 99.23333740234375, TP: 59540.0,Test Loss: 0.05575888603925705, Test Accuracy: 98.22000122070312, Test TP:9822.0

Epoch 4, Loss: 0.014321192167699337, Accuracy: 99.52832794189453, TP: 59717.0,Test Loss: 0.056586846709251404, Test Accuracy: 98.3699951171875, Test TP:9837.0

Epoch 5, Loss: 0.00959738902747631, Accuracy: 99.67166900634766, TP: 59803.0,Test Loss: 0.05430058762431145, Test Accuracy: 98.5199966430664, Test TP:9852.0

**自定义评估函数加入model.fit中**

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function, unicode\_literals

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

**from** **tensorflow.keras.layers** **import** Dense, Flatten, Conv2D

**from** **tensorflow.keras** **import** Model

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

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

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

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

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

fig, ax = plt.subplots(

nrows=2,

ncols=5,

sharex=**True**,

sharey=**True**, )

ax = ax.flatten()

**for** i **in** range(10):

img = x\_train[y\_train == i][0].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()



*# Add a channels dimension*

x\_train = x\_train[..., tf.newaxis]

x\_test = x\_test[..., tf.newaxis]

*# 使用model.fit最好使用one\_hot，不要使用Sparse*

y\_train = tf.one\_hot(y\_train,depth=10)

y\_test = tf.one\_hot(y\_test,depth=10)

train\_ds = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train)).shuffle(10000).batch(32)

test\_ds = tf.data.Dataset.from\_tensor\_slices((x\_test, y\_test)).shuffle(100).batch(32)

**class** **MyModel**(Model):

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

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

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

self.flatten = Flatten()

self.d1 = Dense(128, activation='relu')

self.d2 = Dense(10, activation='softmax')

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

x = self.conv1(x)

x = self.flatten(x)

x = self.d1(x)

**return** self.d2(x)

*#返回的是一个正确的个数*

**class** **CatgoricalTruePositives**(tf.keras.metrics.Metric):

**def** \_\_init\_\_(self, name='categorical\_true\_positives', \*\*kwargs):

super(CatgoricalTruePositives, self).\_\_init\_\_(name=name, \*\*kwargs)

self.true\_positives = self.add\_weight(name='tp', initializer='zeros')

**def** update\_state(self, y\_true, y\_pred, sample\_weight=**None**):

y\_pred = tf.argmax(y\_pred,axis=-1)

y\_true = tf.argmax(y\_true,axis=-1)

values = tf.equal(tf.cast(y\_true, 'int32'), tf.cast(y\_pred, 'int32'))

values = tf.cast(values, 'float32')

**if** sample\_weight **is** **not** **None**:

sample\_weight = tf.cast(sample\_weight, 'float32')

values = tf.multiply(values, sample\_weight)

self.true\_positives.assign\_add(tf.reduce\_sum(values))

**def** result(self):

**return** self.true\_positives

**def** reset\_states(self):

self.true\_positives.assign(0.)

model = MyModel()

model.compile(optimizer = tf.keras.optimizers.Adam(0.001), *#优化器*

loss = tf.keras.losses.CategoricalCrossentropy(), *#损失函数*

metrics = [tf.keras.metrics.CategoricalAccuracy(),

CatgoricalTruePositives(),

]

) *#评估函数*

model.fit(train\_ds, epochs=5,validation\_data=test\_ds)

Epoch 1/5

1875/1875 [==============================] - 179s 96ms/step - loss: 0.1335 - categorical\_accuracy: 0.9598 - categorical\_true\_positives: 57587.0000 - val\_loss: 0.0000e+00 - val\_categorical\_accuracy: 0.0000e+00 - val\_categorical\_true\_positives: 0.0000e+00

...

Epoch 5/5

1875/1875 [==============================] - 182s 97ms/step - loss: 0.0096 - categorical\_accuracy: 0.9968 - categorical\_true\_positives: 59809.0000 - val\_loss: 0.0631 - val\_categorical\_accuracy: 0.9847 - val\_categorical\_true\_positives: 9847.0000

## 4.4.TensorBoard

TensorBoard是一个在深度学习中很好的可视化训练过程和模型结构的工具，那么，要怎么才能在TensorFlow2.0中使用它呢？  
 在TensorFlow2.0中，训练一个神经网络模型主要有两种方式：  
 • 使用tf.keras模块的Model.fit()；  
 • 使用tf.GradientTape()求解梯度，这样可以自定义训练过程。  
 对于这两种方案，都可以使用TensorBoard

Keras在回调函数中内置Tensorboard函数：

tf.keras.callbacks.TensorBoard(  
 log\_dir='logs',  
 histogram\_freq=0,  
 write\_graph=True,  
 write\_images=False,  
 update\_freq='epoch',  
 profile\_batch=2,  
 embeddings\_freq=0,  
 embeddings\_metadata=None  
 )

|  |  |
| --- | --- |
| 参数 | 解释 |
| log\_dir | 保存TensorBoard要解析的日志文件的目录的路径。 |
| histogram\_freq | 频率（在epoch中），计算模型层的激活和权重直方图。如果设置为0，则不会计算直方图。必须为直方图可视化指定验证数据（或拆分）。 |
| write\_graph | 是否在TensorBoard中可视化图像。当write\_graph设置为True时，日志文件可能会变得非常大。 |
| write\_images | 是否在TensorBoard中编写模型权重以显示为图像。 |
| update\_freq | ‘batch’ 或’ epoch’ 或整数。使用‘batch’ 时，在每个batch后将损失和指标(评估函数)写入TensorBoard。这同样适用’ epoch’ 。如果使用整数，比方说1000，回调将会在每1000个样本后将指标和损失写入TensorBoard。请注意，过于频繁地写入TensorBoard会降低您的训练速度。 |
| profile\_batch | 分析批次以采样计算特征。 profile\_batch必须是非负整数或正整数对的逗号分隔字符串。一对正整数表示要分析的批次范 围。默认情况下，它将配置第二批。将profile\_batch = 0设置为禁用性能分析。必须在TensorFlow eager模式下运行。 |
| embeddings\_freq | 可视化嵌入层的频率（以epoch为单位）。如果设置为0，则嵌入将不可见。 |
| embeddings\_metadata | 字典，它将层名称映射到文件名，该嵌入层的元数据保存在该文件名中。 |

下面以在MNIST数据集上训练一个图像分类模型为例介绍。

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function, unicode\_literals

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

**from** **tensorflow.keras.layers** **import** Dense, Flatten, Conv2D

**from** **tensorflow.keras** **import** Model

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

**import** **datetime**

print(tf.\_\_version\_\_)

print(np.\_\_version\_\_)

2.0.0

1.19.0

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

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

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

*# Add a channels dimension*

x\_train = x\_train[..., tf.newaxis]

x\_test = x\_test[..., tf.newaxis]

**class** **MyModel**(Model):

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

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

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

self.flatten = Flatten()

self.d1 = Dense(128, activation='relu')

self.d2 = Dense(10, activation='softmax')

@tf.function

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

x = self.conv1(x)

x = self.flatten(x)

x = self.d1(x)

**return** self.d2(x)

model = MyModel()

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir="keras\_logv1",

histogram\_freq=1,

profile\_batch = 100000000)

model.fit(x=x\_train,

y=y\_train,

epochs=20,

validation\_data=(x\_test, y\_test),

callbacks=[tensorboard\_callback])

Train on 60000 samples, validate on 10000 samples

Epoch 1/20

60000/60000 [==============================] - 6s 104us/sample - loss: 0.1355 - accuracy: 0.9589 - val\_loss: 0.0645 - val\_accuracy: 0.9798

...

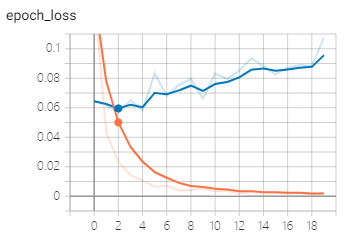
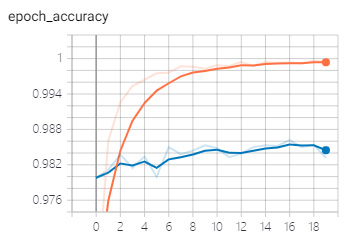
Epoch 20/20

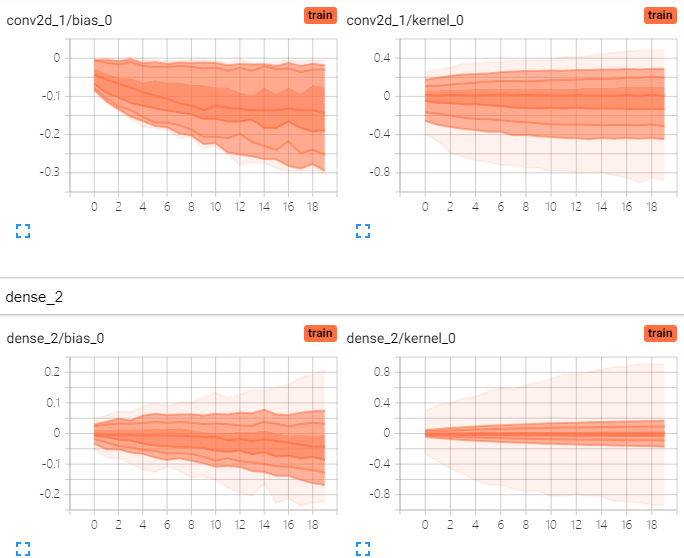
60000/60000 [==============================] - 7s 119us/sample - loss: 0.0020 - accuracy: 0.9994 - val\_loss: 0.1073 - val\_accuracy: 0.9832

执行完成后，可以在cmd中通过命令启动客户端：

tensorboard --bind\_all --logdir D:\...\keras\_logv1

浏览器访问：http://DESKTOP-QBI0CUK:6006/





Tensorboard界面解释：

Scalars : 显示了如何将loss与每个时间段改变。还可以使用它来跟踪训练速度，学习率和其他标量值。  
 Graphs： 进行可视化模型。在这种情况下，将显示层的Keras图，这可以帮助你确保模型正确构建。  
 Distributions 和 Histograms ：显示张量随时间的分布。