# 3.模型训练

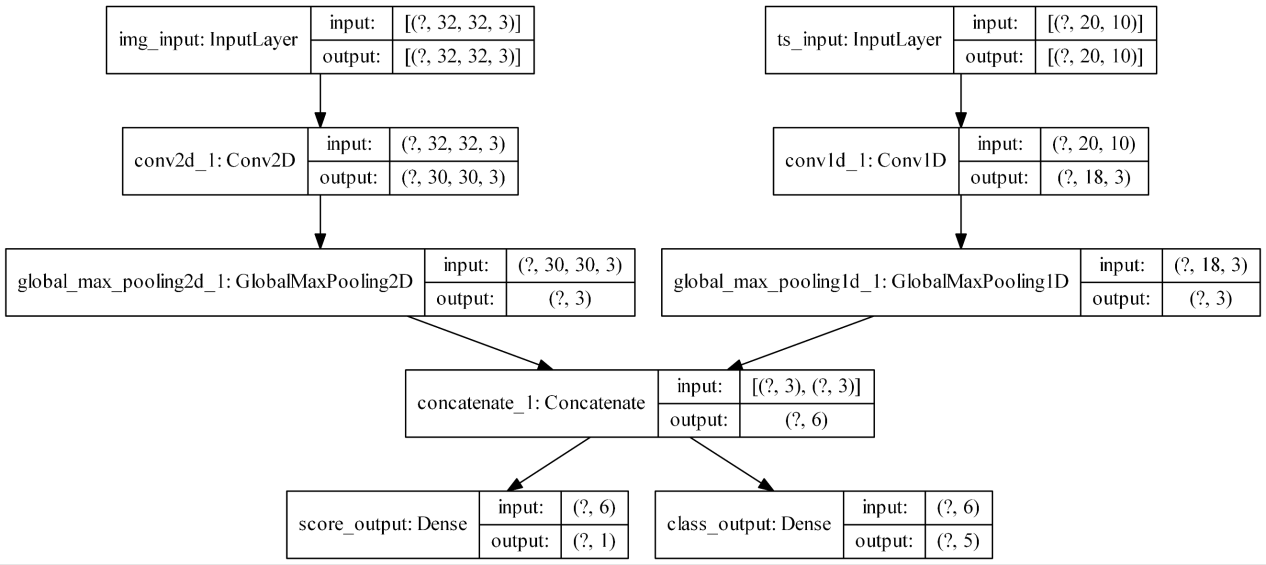
## 3.1.Keras版本模型训练

• 构建模型（顺序模型、函数式模型、子类模型）  
 • 模型训练： model.fit()  
 • 模型验证： model.evaluate()  
 • 模型预测： model.predict()  
 • 使用样本加权和类别加权

回调函数

• ModelCheckpoint：定期保存模型。  
 • EarlyStopping：当训练不再改善验证指标时，停止培训。  
 • TensorBoard：定期编写可在TensorBoard中可视化的模型日志（更多详细信息，请参见“可视化”部分）。  
 • CSVLogger：将损失和指标数据流式传输到CSV文件。

多输入、多输出模型



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

### 3.1.1.keras版本模型训练

相关函数

构建模型（顺序模型、函数式模型、子类模型）

模型训练：model.fit()

模型验证：model.evaluate()

模型预测： model.predict()

**1 构建模型**

inputs = tf.keras.Input(shape=(32,)) *#(数据维度32)*

x = tf.keras.layers.Dense(64, activation='relu')(inputs) *#（64个神经元，）*

x = tf.keras.layers.Dense(64, activation='relu')(x)*#（64个神经元）*

predictions = tf.keras.layers.Dense(10)(x) *#（输出是10类）*

*#- inputs(模型输入)*

*#- output(模型输出)*

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

*#指定损失函数 (loss) tf.keras.optimizers.RMSprop*

*#优化器 (optimizer) tf.keras.losses.SparseCategoricalCrossentropy*

*#指标 (metrics) ['accuracy']*

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

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=**True**), *#损失函数*

metrics=['accuracy']) *#评估函数*

*###构建数据集*

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

x\_train = np.random.random((1000, 32))

y\_train = np.random.randint(10, size=(1000, ))

x\_val = np.random.random((200, 32))

y\_val = np.random.randint(10, size=(200, ))

x\_test = np.random.random((200, 32))

y\_test = np.random.randint(10, size=(200, ))

**2 模型训练**

通过将数据切成大小为“ batch\_size”的“批”来训练模型，并针对给定数量的“epoch”重复遍历整个数据集

model.fit(x\_train, y\_train, batch\_size=32, epochs=5, validation\_data= (x\_val, y\_val) )

Train on 1000 samples, validate on 200 samples

Epoch 1/5

1000/1000 [==============================] - 0s 188us/sample - loss: 2.3188 - accuracy: 0.1040 - val\_loss: 2.3260 - val\_accuracy: 0.0900

...

Epoch 5/5

1000/1000 [==============================] - 0s 163us/sample - loss: 2.2661 - accuracy: 0.1660 - val\_loss: 2.3346 - val\_accuracy: 0.0650

自动划分验证集

在前面的例子中，我们使用validation\_data参数将Numpy数组的元组传递(x\_val, y\_val)给模型，以在每个时期结束时评估验证损失和验证指标。

还有一个选择：参数validation\_split允许您自动保留部分训练数据以供验证。参数值代表要保留用于验证的数据的一部分，因此应将其设置为大于0且小于1的数字。

例如，validation\_split=0.2表示“使用20％的数据进行验证”，而validation\_split=0.6表示“使用60％的数据用于验证”。

model.fit(x\_train, y\_train, batch\_size=64, validation\_split=0.2, epochs=1)

Train on 800 samples, validate on 200 samples

800/800 [==============================] - 0s 100us/sample - loss: 2.2449 - accuracy: 0.1787 - val\_loss: 2.2530 - val\_accuracy: 0.2050

**3 模型验证**

返回 test loss 和metrics

*# Evaluate the model on the test data using `evaluate`*

print('**\n**# Evaluate on test data')

results = model.evaluate(x\_test, y\_test, batch\_size=128)

print('test loss, test acc:', results)

*# Generate predictions (probabilities -- the output of the last layer)*

*# on new data using `predict`*

print('**\n**# Generate predictions for 3 samples')

predictions = model.predict(x\_test[:3])

print('predictions shape:', predictions.shape)

# Evaluate on test data

test loss, test acc: [2.3240616512298584, 0.08]

# Generate predictions for 3 samples

predictions shape: (3, 10)

### 3.1.2.使用样本加权和类别加权

除了输入数据和目标数据外，还可以在使用时将样本权重或类权重传递给模型fit

**1 构建模型**

*# 将生成模型封装为函数，并为每层起名*

**def** get\_uncompiled\_model():

inputs = tf.keras.Input(shape=(32,), name='digits')

x = tf.keras.layers.Dense(64, activation='relu', name='dense\_1')(inputs)

x = tf.keras.layers.Dense(64, activation='relu', name='dense\_2')(x)

outputs = tf.keras.layers.Dense(10, name='predictions')(x)

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

**return** model

**def** get\_compiled\_model():

model = get\_uncompiled\_model()

model.compile(optimizer=tf.keras.optimizers.RMSprop(learning\_rate=1e-3),

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

metrics=['sparse\_categorical\_accuracy'])

**return** model

**2 模型训练**

类别加权

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

*# 设置每个类别对应的权重，类别5：加权*

class\_weight = {0: 1., 1: 1., 2: 1., 3: 1., 4: 1.,

5: 2.,

6: 1., 7: 1., 8: 1., 9: 1.}

print('Fit with class weight')

model = get\_compiled\_model()

model.fit(x\_train, y\_train,

class\_weight=class\_weight,

batch\_size=64,

epochs=4)

Fit with class weight

Train on 1000 samples

Epoch 1/4

1000/1000 [==============================] - 1s 826us/sample - loss: 2.5088 - sparse\_categorical\_accuracy: 0.0800

...

Epoch 4/4

1000/1000 [==============================] - 0s 75us/sample - loss: 2.4648 - sparse\_categorical\_accuracy: 0.0900

样本加权

*# 设置与y\_train相同维度的权重*

sample\_weight = np.ones(shape=(len(y\_train),))

*# 对于所有y\_train为5的数值加权*

sample\_weight[y\_train == 5] = 2.

model = get\_compiled\_model()model.fit(x\_train, y\_train,

sample\_weight=sample\_weight,

batch\_size=64,

epochs=4)

Train on 1000 samples

Epoch 1/4

1000/1000 [==============================] - 1s 901us/sample - loss: 2.5171 - sparse\_categorical\_accuracy: 0.0820

...

Epoch 4/4

1000/1000 [==============================] - 0s 83us/sample - loss: 2.4733 - sparse\_categorical\_accuracy: 0.0880

### 3.1.3.使用回调函数

Keras中的回调是在训练期间（在某个时期开始时，在批处理结束时，在某个时期结束时等）在不同时间点调用的对象，这些对象可用于实现以下行为：

在训练过程中的不同时间点进行验证（除了内置的按时间段验证）

定期或在超过特定精度阈值时对模型进行检查

当训练似乎停滞不前时，更改模型的学习率

当训练似乎停滞不前时，对顶层进行微调

在训练结束或超出特定性能阈值时发送电子邮件或即时消息通知 等等。 回调可以作为列表传递给model.fit

**1 EarlyStopping(早停)**

monitor: 被监测的数据。

min\_delta: 在被监测的数据中被认为是提升的最小变化， 例如，小于 min\_delta 的绝对变化会被认为没有提升。

patience: 没有进步的训练轮数，在这之后训练就会被停止。

verbose: 详细信息模式。

mode: {auto, min, max} 其中之一。如果是 min 模式，学习速率会被降低如果被监测的数据已经停止下降； 在 max 模式，学习塑料会被降低如果被监测的数据已经停止上升； 在 auto 模式，方向会被从被监测的数据中自动推断出来。

model = get\_compiled\_model()

callbacks = [

tf.keras.callbacks.EarlyStopping(

*# 当‘val\_loss’不再下降时候停止训练*

monitor='val\_loss',

*# “不再下降”被定义为“减少不超过1e-2”*

min\_delta=1e-2,

*# “不再改善”进一步定义为“至少2个epoch”*

patience=2,

verbose=1)]

model.fit(x\_train, y\_train,

epochs=20,

batch\_size=64,

callbacks=callbacks,

validation\_split=0.2)

Train on 800 samples, validate on 200 samples

Epoch 1/20

800/800 [==============================] - 1s 1ms/sample - loss: 2.3292 - sparse\_categorical\_accuracy: 0.0938 - val\_loss: 2.3100 - val\_sparse\_categorical\_accuracy: 0.0750

Epoch 2/20

800/800 [==============================] - 0s 100us/sample - loss: 2.3007 - sparse\_categorical\_accuracy: 0.1238 - val\_loss: 2.3065 - val\_sparse\_categorical\_accuracy: 0.0800

Epoch 3/20

800/800 [==============================] - 0s 97us/sample - loss: 2.2889 - sparse\_categorical\_accuracy: 0.1312 - val\_loss: 2.3065 - val\_sparse\_categorical\_accuracy: 0.1100

**2 checkpoint模型**

在相对较大的数据集上训练模型时，至关重要的是要定期保存模型的checkpoint。

最简单的方法是使用ModelCheckpoint回调：

model = get\_compiled\_model()

callbacks = [

tf.keras.callbacks.ModelCheckpoint(

*# 模型保存路径*

filepath='mymodel\_**{epoch}**',

*# 下面的两个参数意味着当且仅当`val\_loss`分数提高时，我们才会覆盖当前检查点。*

save\_best\_only=**True**,

monitor='val\_loss',

*#加入这个仅仅保存模型权重*

save\_weights\_only=**True**,

verbose=1)

]

model.fit(x\_train, y\_train,

epochs=3,

batch\_size=64,

callbacks=callbacks,

validation\_split=0.2)

Train on 800 samples, validate on 200 samples

Epoch 1/3

64/800 [=>............................] - ETA: 6s - loss: 2.3204 - sparse\_categorical\_accuracy: 0.1406

Epoch 00001: val\_loss improved from inf to 2.31889, saving model to mymodel\_1

800/800 [==============================] - 1s 1ms/sample - loss: 2.3174 - sparse\_categorical\_accuracy: 0.0975 - val\_loss: 2.3189 - val\_sparse\_categorical\_accuracy: 0.0850

Epoch 2/3

64/800 [=>............................] - ETA: 0s - loss: 2.3125 - sparse\_categorical\_accuracy: 0.1562

Epoch 00002: val\_loss did not improve from 2.31889

800/800 [==============================] - 0s 100us/sample - loss: 2.2955 - sparse\_categorical\_accuracy: 0.1262 - val\_loss: 2.3218 - val\_sparse\_categorical\_accuracy: 0.0750

Epoch 3/3

768/800 [===========================>..] - ETA: 0s - loss: 2.2868 - sparse\_categorical\_accuracy: 0.1393

Epoch 00003: val\_loss improved from 2.31889 to 2.31446, saving model to mymodel\_3

800/800 [==============================] - 0s 302us/sample - loss: 2.2875 - sparse\_categorical\_accuracy: 0.1375 - val\_loss: 2.3145 - val\_sparse\_categorical\_accuracy: 0.0950

**3 使用回调实现动态学习率调整**

可以通过使用回调来修改优化程序上的当前学习率

ReduceLROnPlateau参数

monitor: 被监测的指标。

factor: 学习速率被降低的因数。新的学习速率 = 学习速率 \* 因数

patience: 没有进步的训练轮数，在这之后训练速率会被降低。

verbose: 整数。0：安静，1：更新信息。

mode: {auto, min, max} 其中之一。如果是 min 模式，学习速率会被降低如果被监测的数据已经停止下降； 在 max 模式，学习塑料会被降低如果被监测的数据已经停止上升； 在 auto 模式，方向会被从被监测的数据中自动推断出来。

min\_delta: 衡量新的最佳阈值，仅关注重大变化。

cooldown: 在学习速率被降低之后，重新恢复正常操作之前等待的训练轮数量。

min\_lr: 学习速率的下边界。

model = get\_compiled\_model()

callbacks = [

tf.keras.callbacks.ModelCheckpoint(

filepath='mymodel\_**{epoch}**',

*# 模型保存路径*

*# 下面的两个参数意味着当且仅当`val\_loss`分数提高时，我们才会覆盖当前检查点。*

save\_best\_only=**True**,

monitor='val\_loss',

*#加入这个仅仅保存模型权重*

save\_weights\_only=**True**,

verbose=1),

tf.keras.callbacks.ReduceLROnPlateau(monitor="val\_sparse\_categorical\_accuracy",

verbose=1,

mode='max',

factor=0.5,

patience=3)

]

model.fit(x\_train, y\_train,

epochs=3,

batch\_size=64,

callbacks=callbacks,

validation\_split=0.2

)

### 3.1.4.多输入，多输出模型

image\_input = tf.keras.Input(shape=(32, 32, 3), name='img\_input')

timeseries\_input = tf.keras.Input(shape=(20, 10), name='ts\_input')

*# 输出通道为3，卷积核为3*

x1 = tf.keras.layers.Conv2D(3, 3)(image\_input)

x1 = tf.keras.layers.GlobalMaxPooling2D()(x1)

x2 = tf.keras.layers.Conv1D(3, 3)(timeseries\_input)

x2 = tf.keras.layers.GlobalMaxPooling1D()(x2)

x = tf.keras.layers.concatenate([x1, x2])

score\_output = tf.keras.layers.Dense(1, name='score\_output')(x)

class\_output = tf.keras.layers.Dense(5, name='class\_output')(x)

model = tf.keras.Model(inputs=[image\_input, timeseries\_input],

outputs=[score\_output, class\_output])

**1 损失函数**

如果我们仅将单个损失函数传递给模型，则将相同的损失函数应用于每个输出

model.compile(

optimizer=tf.keras.optimizers.RMSprop(1e-3),

loss=[tf.keras.losses.MeanSquaredError(),

tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)])

**2指标函数**

定义多个指标

model.compile(

optimizer=tf.keras.optimizers.RMSprop(1e-3),

loss=[tf.keras.losses.MeanSquaredError(),

tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)],

metrics=[

[tf.keras.metrics.MeanAbsolutePercentageError(),

tf.keras.metrics.MeanAbsoluteError()],

[tf.keras.metrics.CategoricalAccuracy()]

]

)

使用字典的形式定义损失和指标

model.compile(

optimizer=tf.keras.optimizers.RMSprop(1e-3),

loss={'score\_output': tf.keras.losses.MeanSquaredError(),

'class\_output': tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)

},

metrics={'score\_output': [tf.keras.metrics.MeanAbsolutePercentageError(),

tf.keras.metrics.MeanAbsoluteError()],

'class\_output': [tf.keras.metrics.CategoricalAccuracy()]})

如果您有两个以上的输出，我们建议使用显式名称和字典。

可以使用以下参数对不同的特定于输出的损失赋予不同的权重

model.compile(

optimizer=tf.keras.optimizers.RMSprop(1e-3),

loss={'score\_output': tf.keras.losses.MeanSquaredError(),

'class\_output': tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)},

metrics={'score\_output': [tf.keras.metrics.MeanAbsolutePercentageError(),

tf.keras.metrics.MeanAbsoluteError()],

'class\_output': [tf.keras.metrics.CategoricalAccuracy()]},

loss\_weights={'score\_output': 2., 'class\_output': 1.})

**3完整运行**

image\_input = tf.keras.Input(shape=(32, 32, 3), name='img\_input')

timeseries\_input = tf.keras.Input(shape=(20, 10), name='ts\_input')

x1 = tf.keras.layers.Conv2D(3, 3)(image\_input)

x1 = tf.keras.layers.GlobalMaxPooling2D()(x1)

x2 = tf.keras.layers.Conv1D(3, 3)(timeseries\_input)

x2 = tf.keras.layers.GlobalMaxPooling1D()(x2)

x = tf.keras.layers.concatenate([x1, x2])

score\_output = tf.keras.layers.Dense(1, name='score\_output')(x)

class\_output = tf.keras.layers.Dense(5, name='class\_output')(x)

model = tf.keras.Model(inputs=[image\_input, timeseries\_input],

outputs=[score\_output, class\_output])

model.compile(

optimizer=tf.keras.optimizers.RMSprop(1e-3),

loss=[tf.keras.losses.MeanSquaredError(),

tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)])

*# Generate dummy Numpy data*

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

img\_data = np.random.random\_sample(size=(100, 32, 32, 3))

ts\_data = np.random.random\_sample(size=(100, 20, 10))

score\_targets = np.random.random\_sample(size=(100, 1))

class\_targets = np.random.random\_sample(size=(100, 5))

*# Fit on lists*

model.fit([img\_data, ts\_data], [score\_targets, class\_targets],

batch\_size=32,

epochs=3)

*# Alternatively, fit on dicts*

model.fit({'img\_input': img\_data, 'ts\_input': ts\_data},

{'score\_output': score\_targets, 'class\_output': class\_targets},

batch\_size=32,

epochs=3)

Train on 100 samples

Epoch 1/3

100/100 [==============================] - 3s 26ms/sample - loss: 4.6500 - score\_output\_loss: 0.0963 - class\_output\_loss: 4.6443

...

Epoch 3/3

100/100 [==============================] - 0s 300us/sample - loss: 4.4291 - score\_output\_loss: 0.1037 - class\_output\_loss: 4.3664

## 3.2.自动求导

tf. GradientTape是TensorFlow 2.0模式下计算梯度用的

tf.GradientTape(  
 persistent=False,  
 watch\_accessed\_variables=True  
 )

⚫persistent : 用来指定新创建的gradient tape是否是可持续性的。默认是False，  
意味着只能够调用一次gradient（）函数。  
 ⚫ watch\_accessed\_variables: 表明这个GradientTape是不是会自动追踪任何能  
被训练（trainable）的变量。默认是True。要是为False的话，意味着你需要手动去指定你想追踪的那些变量。

gradient(target, sources)  
 作用：根据tape上面的上下文来计算某个或者某些tensor的梯度

参数:  
 target: 被微分的Tensor，你可以理解为loss值(针对深度学习训练来说)  
 sources: Tensors 或者Variables列表（当然可以只有一个值） .  
 返回:  
 一个列表表示各个变量的梯度值，和source中的变量列表一一对应，表明这个变量的梯度。

watch(tensor)

作用：确保某个tensor被tape追踪

参数:

tensor: 一个Tensor或者一个Tensor列表

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

x = tf.constant(3.0)

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

g.watch(x)

y = x \* x

dy\_dx = g.gradient(y, x) *# y’ = 2\*x = 2\*3 = 6*

dy\_dx

<tf.Tensor: id=5, shape=(), dtype=float32, numpy=6.0>

案例1、模型自动求导

构建模型（神经网络的前向传播） --> 定义损失函数 --> 定义优化函数 --> 定义tape --> 模型得到预测值 --> 前向传播得到loss --> 反向传播 --> 用优化函数将计算出来的梯度更新到变量上面去

**class** **MyModel**(tf.keras.Model):

**def** \_\_init\_\_(self, num\_classes=10):

super(MyModel, self).\_\_init\_\_(name='my\_model')

self.num\_classes = num\_classes

*# 定义自己需要的层*

self.dense\_1 = tf.keras.layers.Dense(32, activation='relu') *#隐藏层*

self.dense\_2 = tf.keras.layers.Dense(num\_classes)*#输出层*

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

*#定义前向传播*

*# 使用在 (in `\_\_init\_\_`)定义的层*

x = self.dense\_1(inputs)

**return** self.dense\_2(x)

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

*# 10分类问题*

data = np.random.random((1000, 32))

labels = np.random.random((1000, 10))

model = MyModel(num\_classes=10)

loss\_object = tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)

optimizer = tf.keras.optimizers.Adam()

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

predictions = model(data)

loss = loss\_object(labels, predictions)

gradients = tape.gradient(loss, model.trainable\_variables) *#求梯度*

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

model.trainable\_variables

[<tf.Variable 'my\_model/dense/kernel:0' shape=(32, 32) dtype=float32, numpy=

array([[ 0.15029043, 0.15808833, 0.29314938, ..., -0.20451355,

...

<tf.Variable 'my\_model/dense\_1/bias:0' shape=(10,) dtype=float32, numpy=

array([0.00100002, 0.00100002, 0.00100002, 0.00100002, 0.00100002,

0.00100002, 0.00100002, 0.00100002, 0.00100002, 0.00100002],

dtype=float32)>]

apply\_gradients(grads\_and\_vars,name=None)  
 作用：把计算出来的梯度更新到变量上面去。  
 参数含义:  
 grads\_and\_vars: (gradient, variable) 对的列表.  
 name: 操作名

**案例2：使用GradientTape自定义训练模型**

**class** **MyModel**(tf.keras.Model):

**def** \_\_init\_\_(self, num\_classes=10):

super(MyModel, self).\_\_init\_\_(name='my\_model')

self.num\_classes = num\_classes

*# 定义自己需要的层*

self.dense\_1 = tf.keras.layers.Dense(32, activation='relu')

self.dense\_2 = tf.keras.layers.Dense(num\_classes)

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

*#定义前向传播*

*# 使用在 (in `\_\_init\_\_`)定义的层*

x = self.dense\_1(inputs)

**return** self.dense\_2(x)

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

data = np.random.random((1000, 32))

labels = np.random.random((1000, 10))

model = MyModel(num\_classes=10)

*# Instantiate an optimizer.*

optimizer = tf.keras.optimizers.SGD(learning\_rate=1e-3)

*# Instantiate a loss function.*

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

*# Prepare the training dataset.*

batch\_size = 64

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

train\_dataset = train\_dataset.shuffle(buffer\_size=1024).batch(batch\_size)

epochs = 3

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

print('Start of epoch **%d**' % (epoch,))

*# 遍历数据集的batch\_size*

**for** step, (x\_batch\_train, y\_batch\_train) **in** enumerate(train\_dataset):

*# 打开GradientTape以记录正向传递期间运行的操作，这将启用自动区分。*

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

*# 运行该模型的前向传播。 模型应用于其输入的操作将记录在GradientTape上。*

logits = model(x\_batch\_train, training=**True**) *# 这个minibatch的预测值*

*# 计算这个minibatch的损失值*

loss\_value = loss\_fn(y\_batch\_train, logits)

*# 使用GradientTape自动获取可训练变量相对于损失的梯度。*

grads = tape.gradient(loss\_value, model.trainable\_weights)

*# 通过更新变量的值来最大程度地减少损失，从而执行梯度下降的一步。*

optimizer.apply\_gradients(zip(grads, model.trainable\_weights))

*# 每200 batches打印一次.*

**if** step % 200 == 0:

print('Training loss (for one batch) at step **%s**: **%s**' % (step, float(loss\_value)))

print('Seen so far: **%s** samples' % ((step + 1) \* 64))

Start of epoch 0

...

Start of epoch 2

Training loss (for one batch) at step 0: 29.85064697265625

Seen so far: 64 samples

**案例3：使用GradientTape自定义训练模型进阶（加入评估函数）**

让我们将metric添加到组合中。下面可以在从头开始编写的训练循环中随时使用内置指标（或编写的自定义指标）。

流程如下：

在循环开始时初始化metrics

metric.update\_state():每batch之后更新

metric.result():需要显示metrics的当前值时调用

metric.reset\_states():需要清除metrics状态时重置（通常在每个epoch的结尾）

**class** **MyModel**(tf.keras.Model):

**def** \_\_init\_\_(self, num\_classes=10):

super(MyModel, self).\_\_init\_\_(name='my\_model')

self.num\_classes = num\_classes

*# 定义自己需要的层*

self.dense\_1 = tf.keras.layers.Dense(32, activation='relu')

self.dense\_2 = tf.keras.layers.Dense(num\_classes)

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

*#定义前向传播*

*# 使用在 (in `\_\_init\_\_`)定义的层*

x = self.dense\_1(inputs)

**return** self.dense\_2(x)

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

x\_train = np.random.random((1000, 32))

y\_train = np.random.random((1000, 10))

x\_val = np.random.random((200, 32))

y\_val = np.random.random((200, 10))

x\_test = np.random.random((200, 32))

y\_test = np.random.random((200, 10))

*# 优化器*

optimizer = tf.keras.optimizers.SGD(learning\_rate=1e-3)

*# 损失函数*

loss\_fn = tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)

*# 准备metrics函数*

train\_acc\_metric = tf.keras.metrics.CategoricalAccuracy()

val\_acc\_metric = tf.keras.metrics.CategoricalAccuracy()

*# 准备训练数据集*

batch\_size = 64

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

train\_dataset = train\_dataset.shuffle(buffer\_size=1024).batch(batch\_size)

*# 准备验证数据集*

val\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_val, y\_val))

val\_dataset = val\_dataset.batch(64)

model = MyModel(num\_classes=10)

epochs = 3

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

print('Start of epoch **%d**' % (epoch,))

*# 遍历数据集的batch\_size*

**for** step, (x\_batch\_train, y\_batch\_train) **in** enumerate(train\_dataset):

*#一个batch*

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

logits = model(x\_batch\_train)

loss\_value = loss\_fn(y\_batch\_train, logits)

grads = tape.gradient(loss\_value, model.trainable\_weights)

optimizer.apply\_gradients(zip(grads, model.trainable\_weights))*####*

*# 更新训练集的metrics*

train\_acc\_metric(y\_batch\_train, logits)

*# 在每个epoch结束时显示metrics。*

train\_acc = train\_acc\_metric.result()

print('Training acc over epoch: **%s**' % (float(train\_acc),))

*# 在每个epoch结束时重置训练指标*

train\_acc\_metric.reset\_states()

*# 在每个epoch结束时运行一个验证集。*

**for** x\_batch\_val, y\_batch\_val **in** val\_dataset:

val\_logits = model(x\_batch\_val)

*# 更新验证集merics*

val\_acc\_metric(y\_batch\_val, val\_logits)

val\_acc = val\_acc\_metric.result()

print('Validation acc: **%s**' % (float(val\_acc),))

val\_acc\_metric.reset\_states()

Start of epoch 0

...

Start of epoch 2

Training acc over epoch: 0.09000000357627869

Validation acc: 0.09000000357627869

## 3.3.AutoGraph机制

有哪些计算图？

有三种计算图的构建方式： 静态计算图， 动态计算图，以及AutoGraph。  
 静态计算图：静态计算则意味着程序在编译执行时将先生成神经网络的结构，然后再执行相应操作。从理论上讲，静态计算这样的机制允许编译器进行更大程度的优化，但是这也意味着你所期望的程序与编译器实际执行之间存在着更多的代沟。这也意味着，代码中的错误将更加难以发现（比如，如果计算图的结构出现问题，你可能只有在代码执行到相应操作的时候才能发现它）  
 动态计算图：动态计算意味着程序将按照我们编写命令的顺序进行执行。这种机制将使得调试更加容易，并且也使得我们将大脑中的想法转化为实际代码变得更加容易。

什么是AutoGraph？

TensorFlow 2.0主要使用的是动态计算图和Autograph。  
 而Autograph机制可以将动态图转换成静态计算图，兼收执行效率和编码效率之利。

动态计算图易于调试，编码效率较高，但执行效率偏低。  
 静态计算图执行效率很高，但较难调试。  
 AutoGraph在TensorFlow 2.0通过@tf.function实现的。

AutoGraph使用规范

• 被@tf.function修饰的函数应尽量使用TensorFlow中的函数而不是Python中的其他函数。  
 • 避免在@tf.function修饰的函数内部定义tf.Variable.  
 • 被@tf.function修饰的函数不可修改该函数外部的Python列表或字典等结构类型变量。

1、被@tf.function修饰的函数应尽量使用TensorFlow中的函数而不是Python中的其他函数。

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

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

@tf.function**def** np\_random():

a = np.random.randn(3,3)

tf.print(a)

@tf.function**def** tf\_random():

a = tf.random.normal((3,3))

tf.print(a)

*#np\_random每次执行都是一样的结果。*

np\_random()

np\_random()

array([[-0.8533249 , -0.65982598, 0.46978706],

[-1.19132929, -0.80967162, 0.33337003],

[ 1.06020708, -0.94404936, -0.1212558 ]])

array([[-0.8533249 , -0.65982598, 0.46978706],

[-1.19132929, -0.80967162, 0.33337003],

[ 1.06020708, -0.94404936, -0.1212558 ]])

tf\_random()

tf\_random()

[[0.9276793 -0.528112411 1.88184559]

[-1.20099926 -1.04864311 0.245232761]

[-1.71584272 -1.03118122 -0.0328678861]]

[[-0.0503324643 -0.459995121 -1.84465837]

[1.13207293 0.68550396 -0.458074749]

[1.17981255 1.2004118 -0.352737874]]

2、避免在@tf.function修饰的函数内部定义tf.Variable.

x = tf.Variable(1.0,dtype=tf.float32)

@tf.function

**def** outer\_var():

x.assign\_add(1.0)

tf.print(x)

**return**(x)

outer\_var()

outer\_var()

2

3

*#报错*

@tf.function

**def** inner\_var():

x = tf.Variable(1.0,dtype = tf.float32)

x.assign\_add(1.0)

tf.print(x)

**return**(x)

inner\_var()

**ValueError**: in converted code:

ValueError: tf.function-decorated function tried to create variables on non-first call.

3、被@tf.function修饰的函数不可修改该函数外部的Python列表或字典等结构类型变量。

tensor\_list = []

*#@tf.function #加上这一行切换成Autograph结果将不符合预期！！！*

**def** append\_tensor(x):

tensor\_list.append(x)

**return** tensor\_list

append\_tensor(tf.constant(5.0))

append\_tensor(tf.constant(6.0))

print(tensor\_list)

[<tf.Tensor: id=81, shape=(), dtype=float32, numpy=5.0>, <tf.Tensor: id=82, shape=(), dtype=float32, numpy=6.0>]

tensor\_list = []

@tf.function *#加上这一行切换成Autograph结果将不符合预期！！！*

**def** append\_tensor(x):

tensor\_list.append(x)

**return** tensor\_list

append\_tensor(tf.constant(5.0))

append\_tensor(tf.constant(6.0))

print(tensor\_list)

[<tf.Tensor 'x:0' shape=() dtype=float32>]

AutoGraph机制原理

当我们使用@tf.function装饰一个函数的时候，后面到底发生了什么呢？  
 第一件事情是创建计算图。  
 第二件事情是执行计算图。

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

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

@tf.function(autograph=**True**)

**def** myadd(a,b):

**for** i **in** tf.range(3):

tf.print(i)

c = a+b

print("tracing")

**return** c

myadd(tf.constant("hello"),tf.constant("world"))

tracing

0

1

2

<tf.Tensor: id=49, shape=(), dtype=string, numpy=b'helloworld'>

因此我们先看到的是第一个步骤的结果：即Python调用标准输出流打印"tracing"语句。

然后看到第二个步骤的结果：TensorFlow调用标准输出流打印1,2,3。

当我们再次用相同的输入参数类型调用这个被@tf.function装饰的函数时，后面到底发生了什么？

myadd(tf.constant("good"),tf.constant("morning"))

0

1

2

<tf.Tensor: id=52, shape=(), dtype=string, numpy=b'goodmorning'>

只会发生一件事情，那就是上面步骤的第二步，执行计算图。 所以这一次我们没有看到打印"tracing"的结果。

当我们再次用不同的的输入参数类型调用这个被@tf.function装饰的函数时，后面到底发生了什么？

myadd(tf.constant(1),tf.constant(2))

tracing

0

1

2

<tf.Tensor: id=100, shape=(), dtype=int32, numpy=3>

由于输入参数的类型已经发生变化，已经创建的计算图不能够再次使用。

需要重新做2件事情：创建新的计算图、执行计算图。

所以我们又会先看到的是第一个步骤的结果：即Python调用标准输出流打印"tracing"语句。

然后再看到第二个步骤的结果：TensorFlow调用标准输出流打印1,2,3。

需要注意的是，如果调用被@tf.function装饰的函数时输入的参数不是Tensor类型，则每次都会重新创建计算图。

例如我们写下如下代码。两次都会重新创建计算图。因此，一般建议调用@tf.function时应传入Tensor类型。

myadd("hello","world")

myadd("good","morning")

tracing

0

1

2

tracing

0

1

2

<tf.Tensor: id=188, shape=(), dtype=string, numpy=b'goodmorning'>

**AutoGraph使用案例**

在tf.function中用input\_signature限定输入张量的签名类型：shape和dtype

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

x = tf.Variable(1.0,dtype=tf.float32)

*#在tf.function中用input\_signature限定输入张量的签名类型：shape和dtype*

@tf.function(input\_signature=[tf.TensorSpec(shape = [], dtype = tf.float32)])

**def** add\_print(a):

x.assign\_add(a)

tf.print(x)

**return**(x)

add\_print(tf.constant(3.0))

*#add\_print(tf.constant(3)) #输入不符合张量签名的参数将报错*

4

<tf.Tensor: id=18, shape=(), dtype=float32, numpy=4.0>

下面利用tf.Module的子类化将其封装一下。

**class** **DemoModule**(tf.Module):

**def** \_\_init\_\_(self,init\_value = tf.constant(0.0),name=**None**):

super(DemoModule, self).\_\_init\_\_(name=name)

**with** self.name\_scope: *#相当于with tf.name\_scope("demo\_module")*

self.x = tf.Variable(init\_value,dtype = tf.float32,trainable=**True**)

@tf.function(input\_signature=[tf.TensorSpec(shape = [], dtype = tf.float32)])

**def** addprint(self,a):

**with** self.name\_scope:

self.x.assign\_add(a)

tf.print(self.x)

**return**(self.x)

*#执行*

demo = DemoModule(init\_value = tf.constant(1.0))

result = demo.addprint(tf.constant(5.0))

6

*#查看模块中的全部变量和全部可训练变量*

print(demo.variables)

print(demo.trainable\_variables)

(<tf.Variable 'demo\_module/Variable:0' shape=() dtype=float32, numpy=6.0>,)

(<tf.Variable 'demo\_module/Variable:0' shape=() dtype=float32, numpy=6.0>,)

*#查看模块中的全部子模块*

demo.submodules

()

*#使用tf.saved\_model 保存模型，并指定需要跨平台部署的方法*

tf.saved\_model.save(demo,"./data/",signatures = {"serving\_default":demo.addprint})

INFO:tensorflow:Assets written to: ./data/assets

*#加载模型*

demo2 = tf.saved\_model.load("./data/")

demo2.addprint(tf.constant(5.0))

11

<tf.Tensor: id=196, shape=(), dtype=float32, numpy=11.0>

*# 查看模型文件相关信息，红框标出来的输出信息在模型部署和跨平台使用时有可能会用到*!

saved\_model\_cli show --dir ./data/ --all

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

...

Method name is: tensorflow/serving/predict

构建全连接模型案例

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

**class** **MyModel**(tf.keras.Model):

**def** \_\_init\_\_(self, num\_classes=10):

super(MyModel, self).\_\_init\_\_(name='my\_model')

self.num\_classes = num\_classes

*# 定义自己需要的层*

self.dense\_1 = tf.keras.layers.Dense(32, activation='relu')

self.dense\_2 = tf.keras.layers.Dense(num\_classes)

@tf.function(input\_signature=[tf.TensorSpec([**None**,32], tf.float32)])

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

*#定义前向传播*

*# 使用在 (in `\_\_init\_\_`)定义的层*

x = self.dense\_1(inputs)

**return** self.dense\_2(x)

data = np.random.random((1000, 32))

labels = np.random.random((1000, 10))

*# Instantiate an optimizer.*

optimizer = tf.keras.optimizers.SGD(learning\_rate=1e-3)

*# Instantiate a loss function.*

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

*# Prepare the training dataset.*

batch\_size = 64

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

train\_dataset = train\_dataset.shuffle(buffer\_size=1024).batch(batch\_size)

model = MyModel(num\_classes=10)

epochs = 3

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

print('Start of epoch **%d**' % (epoch,))

*# 遍历数据集的batch\_size*

**for** step, (x\_batch\_train, y\_batch\_train) **in** enumerate(train\_dataset):

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

logits = model(x\_batch\_train)

loss\_value = loss\_fn(y\_batch\_train, logits)

grads = tape.gradient(loss\_value, model.trainable\_weights)

optimizer.apply\_gradients(zip(grads, model.trainable\_weights))

*# 每200 batches打印一次.*

**if** step % 200 == 0:

print('Training loss (for one batch) at step **%s**: **%s**' % (step, float(loss\_value)))

print('Seen so far: **%s** samples' % ((step + 1) \* 64))

Start of epoch 0

...

Start of epoch 2

Training loss (for one batch) at step 0: 23.022199630737305

Seen so far: 64 samples

tf.saved\_model.save(model,'my\_saved\_model')

INFO:tensorflow:Assets written to: my\_saved\_model\assets

## 3.4.模型保存与加载

### 3.4.1.Keras版本模型保存与加载

保存模型权重  
 • 方法一：保存模型权重（ model.save\_weights）  
保存整个模型  
 • 方法二：保存HDF5文件（model.save）  
 • 方法三：保存pb文件（tf.saved\_model）  
 区别： saved\_model 没有保存优化器配置

方法一仅仅保存了模型中的权重(weights)。

方法二模型和优化器都可以一起保存，包括权重(weights)、模型配置(architecture)和优化器配置(optimizer configuration)。  
 这样做的好处是，当你恢复模型时，完全不依赖于原来搭建模型的代码。

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

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

x\_train = np.random.random((1000, 32))

y\_train = np.random.randint(10, size=(1000, ))

x\_val = np.random.random((200, 32))

y\_val = np.random.randint(10, size=(200, ))

x\_test = np.random.random((200, 32))

y\_test = np.random.randint(10, size=(200, ))

**def** get\_uncompiled\_model():

inputs = tf.keras.Input(shape=(32,), name='digits')

x = tf.keras.layers.Dense(64, activation='relu', name='dense\_1')(inputs)

x = tf.keras.layers.Dense(64, activation='relu', name='dense\_2')(x)

outputs = tf.keras.layers.Dense(10, name='predictions')(x)

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

**return** model

**def** get\_compiled\_model():

model = get\_uncompiled\_model()

model.compile(optimizer=tf.keras.optimizers.RMSprop(learning\_rate=1e-3),

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

metrics=['sparse\_categorical\_accuracy'])

**return** model

model = get\_compiled\_model()

model.fit(x\_train, y\_train, batch\_size=32, epochs=5, validation\_data=(x\_val, y\_val))

Train on 1000 samples, validate on 200 samples

Epoch 1/5

1000/1000 [==============================] - 2s 2ms/sample - loss: 2.3223 - sparse\_categorical\_accuracy: 0.1020 - val\_loss: 2.3150 - val\_sparse\_categorical\_accuracy: 0.0850

...

Epoch 5/5

1000/1000 [==============================] - 0s 150us/sample - loss: 2.2668 - sparse\_categorical\_accuracy: 0.1590 - val\_loss: 2.3210 - val\_sparse\_categorical\_accuracy: 0.0750

model.summary()

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

digits (InputLayer) [(None, 32)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 64) 2112

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 64) 4160

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

predictions (Dense) (None, 10) 650

=================================================================

Total params: 6,922

Trainable params: 6,922

Non-trainable params: 0

**方法一**

model.save\_weights("adasd.h5")

model.load\_weights("adasd.h5")

model.predict(x\_test)

array([[-4.79599386e-01, -7.43045211e-02, 7.53526539e-02, ...,

-8.08903947e-02, -1.06541492e-01, -3.22730273e-01],

...

[-3.90737444e-01, 1.99008718e-01, 1.61002487e-01, ...,

1.07340984e-01, 1.46755893e-02, -4.48317945e-01]], dtype=float32)

model.save\_weights('./checkpoints/mannul\_checkpoint')

model.load\_weights('./checkpoints/mannul\_checkpoint')

model.predict(x\_test)

array([[-4.79599386e-01, -7.43045211e-02, 7.53526539e-02, ...,

-8.08903947e-02, -1.06541492e-01, -3.22730273e-01],

...

[-3.90737444e-01, 1.99008718e-01, 1.61002487e-01, ...,

1.07340984e-01, 1.46755893e-02, -4.48317945e-01]], dtype=float32)

**方法二**

model.save('keras\_model\_hdf5\_version.h5')

new\_model = tf.keras.models.load\_model('keras\_model\_hdf5\_version.h5')

new\_model.predict(x\_test)

array([[-4.79599386e-01, -7.43045211e-02, 7.53526539e-02, ...,

-8.08903947e-02, -1.06541492e-01, -3.22730273e-01],

...

[-3.90737444e-01, 1.99008718e-01, 1.61002487e-01, ...,

1.07340984e-01, 1.46755893e-02, -4.48317945e-01]], dtype=float32)

**方法三**

*# 通常用于部署*

tf.saved\_model.save(model,'tf\_saved\_model\_version')

restored\_saved\_model = tf.saved\_model.load('tf\_saved\_model\_version')

f = restored\_saved\_model.signatures["serving\_default"]

INFO:tensorflow:Assets written to: tf\_saved\_model\_version\assets

f(digits = tf.constant(x\_test.tolist()) )

{'predictions': <tf.Tensor: id=7078, shape=(200, 10), dtype=float32, numpy=

array([[-4.79599386e-01, -7.43045211e-02, 7.53526539e-02, ...,

-8.08903947e-02, -1.06541492e-01, -3.22730273e-01],

...

[-3.90737444e-01, 1.99008718e-01, 1.61002487e-01, ...,

1.07340984e-01, 1.46755893e-02, -4.48317945e-01]], dtype=float32)>}

!saved\_model\_cli show --dir tf\_saved\_model\_version --all

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

...

Method name is: tensorflow/serving/predict

### 3.4.2.自定义版本模型保存与加载

保存模型权重  
 • 方法一：保存checkpoint模型权重  
保存整个模型  
 • 方法二：保存HDF5文件（model.save）  
 • 方法三：保存pb文件（tf.saved\_model）

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

**class** **MyModel**(tf.keras.Model):

**def** \_\_init\_\_(self, num\_classes=10):

super(MyModel, self).\_\_init\_\_(name='my\_model')

self.num\_classes = num\_classes

*# 定义自己需要的层*

self.dense\_1 = tf.keras.layers.Dense(32, activation='relu')

self.dense\_2 = tf.keras.layers.Dense(num\_classes)

@tf.function(input\_signature=[tf.TensorSpec([**None**,32], tf.float32,name='digits')])

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

*#定义前向传播*

*# 使用在 (in `\_\_init\_\_`)定义的层*

x = self.dense\_1(inputs)

**return** self.dense\_2(x)

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

x\_train = np.random.random((1000, 32))

y\_train = np.random.random((1000, 10))

x\_val = np.random.random((200, 32))

y\_val = np.random.random((200, 10))

x\_test = np.random.random((200, 32))

y\_test = np.random.random((200, 10))

*# 优化器*

optimizer = tf.keras.optimizers.SGD(learning\_rate=1e-3)

*# 损失函数*

loss\_fn = tf.keras.losses.CategoricalCrossentropy(from\_logits=**True**)

*# 准备metrics函数*

train\_acc\_metric = tf.keras.metrics.CategoricalAccuracy()

val\_acc\_metric = tf.keras.metrics.CategoricalAccuracy()

*# 准备训练数据集*

batch\_size = 64

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

train\_dataset = train\_dataset.shuffle(buffer\_size=1024).batch(batch\_size)

*# 准备测试数据集*

val\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_val, y\_val))

val\_dataset = val\_dataset.batch(64)

model = MyModel(num\_classes=10)

epochs = 3

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

print('Start of epoch **%d**' % (epoch,))

*# 遍历数据集的batch\_size*

**for** step, (x\_batch\_train, y\_batch\_train) **in** enumerate(train\_dataset):

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

logits = model(x\_batch\_train)

loss\_value = loss\_fn(y\_batch\_train, logits)

grads = tape.gradient(loss\_value, model.trainable\_weights)

optimizer.apply\_gradients(zip(grads, model.trainable\_weights))

*# 更新训练集的metrics*

train\_acc\_metric(y\_batch\_train, logits)

*# 每200 batches打印一次.*

**if** step % 200 == 0:

print('Training loss (for one batch) at step **%s**: **%s**' % (step, float(loss\_value)))

print('Seen so far: **%s** samples' % ((step + 1) \* 64))

*# 在每个epoch结束时显示metrics。*

train\_acc = train\_acc\_metric.result()

print('Training acc over epoch: **%s**' % (float(train\_acc),))

*# 在每个epoch结束时重置训练指标*

train\_acc\_metric.reset\_states()

*# 在每个epoch结束时运行一个验证集。*

**for** x\_batch\_val, y\_batch\_val **in** val\_dataset:

val\_logits = model(x\_batch\_val)

*# 更新验证集merics*

val\_acc\_metric(y\_batch\_val, val\_logits)

val\_acc = val\_acc\_metric.result()

val\_acc\_metric.reset\_states()

print('Validation acc: **%s**' % (float(val\_acc),))

Start of epoch 0

...

Start of epoch 2

Training loss (for one batch) at step 0: 12.48736572265625

Seen so far: 64 samples

Training acc over epoch: 0.10300000011920929

Validation acc: 0.12999999523162842

**方法一**

model.save\_weights("adasd.h5")

model.load\_weights("adasd.h5")

model.predict(x\_test)

array([[ 0.59265125, 0.12652864, 0.50779265, ..., 0.38236466,

-0.48236898, -0.29403484],

...

[ 0.62080455, 1.1130711 , 0.523637 , ..., 0.31726438,

-0.10991763, -0.76191264]], dtype=float32)

model.save\_weights('./checkpoints/mannul\_checkpoint')

model.load\_weights('./checkpoints/mannul\_checkpoint')

model.predict(x\_test)

array([[ 0.59265125, 0.12652864, 0.50779265, ..., 0.38236466,

-0.48236898, -0.29403484],

...

[ 0.62080455, 1.1130711 , 0.523637 , ..., 0.31726438,

-0.10991763, -0.76191264]], dtype=float32)

**方法二**

model.save('path\_to\_my\_model',save\_format='tf')

INFO:tensorflow:Assets written to: path\_to\_my\_model\assets

new\_model = tf.keras.models.load\_model('path\_to\_my\_model')

new\_model.predict(x\_test)

array([[ 0.59265125, 0.12652864, 0.50779265, ..., 0.38236466,

-0.48236898, -0.29403484],

...

[ 0.62080455, 1.1130711 , 0.523637 , ..., 0.31726438,

-0.10991763, -0.76191264]], dtype=float32)

**方法三**

tf.saved\_model.save(model,'my\_saved\_model')

restored\_saved\_model = tf.saved\_model.load('my\_saved\_model')

f = restored\_saved\_model.signatures["serving\_default"]

INFO:tensorflow:Assets written to: my\_saved\_model\assets

f(digits = tf.constant(x\_test.tolist()) )

{'output\_0': <tf.Tensor: id=20472, shape=(200, 10), dtype=float32, numpy=

array([[0.36048555, 0.77584916, 0.5105363 , ..., 0.31300798, 0.15851551,

0.9504347 ],

...

[0.39257365, 0.8122336 , 1.0617772 , ..., 0.25777856, 0.25043324,

0.7244146 ]], dtype=float32)>}