Exercise 1

你会如何用一个简短的句子来描述 TensorFlow 呢？它的主要特点是什么？你能说出其他流行的深度学习库的名字吗？

答案：

TensorFlow是一个用于数值计算的开源库，特别适用于大规模机器学习和微调。它的核心与NumPy相似，但它还具有GPU支持、分布式计算支持、计算图分析和优化功能（具有可移植的图格式，允许您在一个环境中训练TensorFlow模型，并在另一个环境下运行它）、基于反向模式autodiff的优化API以及几个强大的API，如tf.keras、tf.data、tf.image、tf.signal、，以及更多。其他流行的深度学习库包括PyTorch、MXNet、Microsoft认知工具包、Theano、Caffe2和Chainer。

Exercise 2

TensorFlow 是 NumPy 的临时替代品吗？这两者之间的主要区别是什么？

答案：

尽管TensorFlow提供了NumPy提供的大部分功能，但出于几个原因，它并不是一个替代品。

1. 首先，函数的名称并不总是相同的（例如，tf.reduce_sum（）与np.sum（））。
2. 其次，一些函数的行为方式并不完全相同（例如，tf.reduce（）创建张量的转置副本，而NumPy的T属性创建转置视图，而实际上没有复制任何数据）。
3. 最后，NumPy数组是可变的，而TensorFlow张量不是可变的（但如果需要可变对象，可以使用tf.Variable）。

Exercise 3

使用 tf.range(10) 和 tf.constant(np.arange(10)) 获得的结果相同吗？

答案：

tf.range(10) 和 tf.constant(np.arange(10)) 都返回包含整数0到9的一维张量。然而，前者使用32位整数，后者使用64位整数。实际上，TensorFlow默认为32位，而NumPy默认为64位。

Exercise 4

除了规则张量之外，你还能说出TensorFlow中的其他六种数据结构吗？

答案：

除了常规张量，TensorFlow还提供了其他几种数据结构，包括稀疏张量、张量阵列、不规则张量、队列、字符串张量和集合。最后两个实际上表示为正则张量，但TensorFlow提供了特殊的函数来处理它们（在tf.strings和tf.sets中）。

Exercise 5

您可以通过编写函数或子类化 tf.keras.losses.Loss 来定义自定义损失函数。你什么时候会使用每个选项？

答案：

当您想要定义自定义损失函数时，通常可以将其作为常规Python函数实现。但是，如果您的自定义loss函数必须支持某些超参数（或任何其他状态），那么您应该将keras.losses.loss类子类化，并实现**init**（）和call（）方法。如果您希望损失函数的超参数与模型一起保存，那么还必须实现get_config（）方法。

Exercise 6

类似地，您可以在函数中定义自定义度量，或者作为 tf.keras.metrics.Metric 的子类。你什么时候会使用每个选项？

答案：

与自定义损失函数非常相似，大多数度量可以定义为常规Python函数。但是，如果您希望自定义度量支持某些超参数（或任何其他状态），那么应该将keras.metrics.metric类子类化。此外，如果在整个epoch上计算度量不等同于在该epoch中的所有batch上计算平均度量（例如，精度和召回度量），则应将keras.metrics.metric类子类化，并实现**init**（）、update_state（）和result（）方法，以跟踪每个历元期间的运行度量。您还应该实现reset_states（）方法，除非它只需要将所有变量重置为0.0。如果您希望状态与模型一起保存，那么也应该实现get_config（）方法。

Exercise 7

什么时候应该创建自定义层还是自定义模型？

答案：

您应该将模型的内部组件（即层或可重用的层块）与模型本身（即要训练的对象）区分开来。前者应子类keras.layers.Layer类，而后者应子类keras.models.Model类。

Exercise 8

有哪些用例需要编写您自己的自定义训练循环？

答案：

编写自己的自定义训练循环是相当先进的，因此只有在真正需要时才应该这样做。Keras提供了几个工具来定制训练，而不必编写自定义训练循环：回调、自定义正则化、自定义约束、自定义丢失等。您应该尽可能使用这些方法，而不是编写自定义训练循环：编写自定义训练环更容易出错，而且重用您编写的自定义代码会更困难。

然而，在某些情况下，编写自定义训练循环是必要的—例如，如果你想对神经网络的不同部分使用不同的优化器，比如在Wide&Deep论文中。自定义训练循环在调试或试图准确理解训练的工作方式时也很有用。

Exercise 9

自定义Keras组件是否包含任意Python代码，或者它们是否必须可转换为TF函数？

答案：

自定义Keras组件应可转换为TF函数，这意味着它们应尽可能遵守TF操作，并遵守第12章（TF函数规则部分）中列出的所有规则。如果您绝对需要在自定义组件中包含任意Python代码，则可以将其包装在tf.py_function（）操作中（但这会降低性能并限制模型的可移植性），或者在创建自定义层或模型时设置dynamic=True（或者在调用模型的compile（）方法时设置run_eagerly=True）。

Exercise 10

如果您希望函数转换为TF函数，需要尊重的主要规则是什么？

答案：

请参阅第12章，了解创建TF功能时应遵守的规则列表（在TF功能规则部分）。

Exercise 11

你什么时候需要创建一个动态的Keras模型呢？你该怎么做呢？为什么不让你所有的模型都动态化呢？

答案：

创建动态Keras模型对于调试非常有用，因为它不会将任何自定义组件编译为TF函数，您可以使用任何Python调试器来调试代码。如果您希望在模型（或培训代码）中包含任意Python代码，包括对外部库的调用，那么它也很有用。

要使模型成为动态的，必须在创建模型时设置dynamic=True。或者，可以在调用模型的compile（）方法时设置run_eagerly=True。

使模型动态化会阻止Keras使用TensorFlow的任何图形功能，因此它会降低训练和推理速度，并且您将无法导出计算图，这将限制模型的可移植性。

Exercise 12

实现一个执行层规范化的自定义层（我们将在第15章中使用这种类型的层）：

1. build() 方法应该定义两个可训练的权值 α 和 β，形状input_shape[-1:]并且数据类型为tf.float32。α 应该初始化为1，β 应该初始化为0。
2. call() 方法应该计算每个实例特征的平均 μ 和标准差 σ。为此，您可以使用tf.nn.moments(inputs, axes=-1, keepdims=True)，它返回所有实例的均值 μ 和方差 σ^2（计算方差的平方根来得到标准差）。然后函数应该计算并返回 $\alpha \otimes （X - \mu）/（\sigma + \varepsilon） + \beta$，其中⊗表示逐乘法（*），$\varepsilon$ 是一个平滑项（一个小常数以避免除零，例如，0.001）。
3. 确保自定义图层生成与tf.keras.layers.LayerNormalization相同（或非常接近相同）的输出。

答案：

In [2]:
```
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
from pathlib import Path
```

```
from time import strftime
import tensorboard
```

In [4]:
```
# load the CIFAR10 dataset

cifar10 = tf.keras.datasets.cifar10.load_data()
(X_train_full, y_train_full), (X_test, y_test) = cifar10

X_train = X_train_full[5000:]
y_train = y_train_full[5000:]
X_valid = X_train_full[:5000]
y_valid = y_train_full[:5000]
```

In [3]:
```
class LayerNormalization(tf.keras.layers.Layer):
    def __init__(self, eps=0.001, **kwargs):
        super().__init__(**kwargs)
        self.eps = eps

    def build(self, batch_input_shape):
        self.alpha = self.add_weight(
            name="alpha", shape=batch_input_shape[-1:],
            initializer="ones")
        self.beta = self.add_weight(
            name="beta", shape=batch_input_shape[-1:],
            initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        mean, variance = tf.nn.moments(X, axes=-1, keepdims=True)
        return self.alpha * (X - mean) / (tf.sqrt(variance + self.eps)) + self.beta

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "eps": self.eps}
```

In [5]:
```
# Let's create one instance of each class
# apply them to some data (e.g., the training set)
# and ensure that the difference is negligeable.

X = X_train.astype(np.float32)

custom_layer_norm = LayerNormalization()
keras_layer_norm = tf.keras.layers.LayerNormalization()

tf.reduce_mean(tf.keras.losses.mean_absolute_error(
    keras_layer_norm(X), custom_layer_norm(X)))
```

Out[5]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=6.5335115e-07>
```

Exercise 13

使用一个自定义的训练循环来训练一个模型，以处理Fashion MNIST数据集。（见第10章）：

1. 显示每个epoch、迭代、平均训练损失和平均准确性（每次迭代时更新），以及每个历元结束时的验证损失和准确性。
2. 尝试对上层和下层使用不同的优化器和不同的学习速率。

答案：

（a）：

In [7]:
```
(X_train_full, y_train_full), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_da

X_train_full = X_train_full.astype(np.float32) / 255.

X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

X_test = X_test.astype(np.float32) / 255.
```

In [8]:
```
tf.keras.backend.clear_session()

np.random.seed(42)
tf.random.set_seed(42)
```

In [9]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax"),
])
```

In [10]:
```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.sparse_categorical_crossentropy
mean_loss = tf.keras.metrics.Mean()
metrics = [tf.keras.metrics.SparseCategoricalAccuracy()]
```

In [14]:
```
from tqdm.notebook import trange
from collections import OrderedDict

def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]

with trange(1, n_epochs + 1, desc="All epochs") as epochs:
    for epoch in epochs:
        with trange(1, n_steps + 1, desc=f"Epoch {epoch}/{n_epochs}") as steps:
            for step in steps:
                X_batch, y_batch = random_batch(X_train, y_train)
                with tf.GradientTape() as tape:
                    y_pred = model(X_batch)
                    main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
                    loss = tf.add_n([main_loss] + model.losses)
                gradients = tape.gradient(loss, model.trainable_variables)
                optimizer.apply_gradients(zip(gradients, model.trainable_variables))
                for variable in model.variables:
                    if variable.constraint is not None:
                        variable.assign(variable.constraint(variable))
                status = OrderedDict()
                mean_loss(loss)
                status["loss"] = mean_loss.result().numpy()
                for metric in metrics:
                    metric(y_batch, y_pred)
                    status[metric.name] = metric.result().numpy()
                steps.set_postfix(status)
            y_pred = model(X_valid)
            status["val_loss"] = np.mean(loss_fn(y_valid, y_pred))
            status["val_accuracy"] = np.mean(tf.keras.metrics.sparse_categorical_accurac
                tf.constant(y_valid, dtype=np.float32), y_pred))
```

```
            steps.set_postfix(status)
        for metric in [mean_loss] + metrics:
            metric.reset_states()
```

```
All epochs:   0%|          | 0/5 [00:00<?, ?it/s]
Epoch 1/5:    0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 2/5:    0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 3/5:    0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 4/5:    0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 5/5:    0%|          | 0/1718 [00:00<?, ?it/s]
```

（b）：

In [15]:
```
tf.keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

In [16]:
```
lower_layers = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu"),
])
upper_layers = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation="softmax"),
])
model = tf.keras.Sequential([
    lower_layers, upper_layers
])
```

In [17]:
```
lower_optimizer = tf.keras.optimizers.SGD(learning_rate=1e-4)
upper_optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-3)
```

In [18]:
```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
loss_fn = tf.keras.losses.sparse_categorical_crossentropy
mean_loss = tf.keras.metrics.Mean()
metrics = [tf.keras.metrics.SparseCategoricalAccuracy()]
```

In [19]:
```
with trange(1, n_epochs + 1, desc="All epochs") as epochs:
    for epoch in epochs:
        with trange(1, n_steps + 1, desc=f"Epoch {epoch}/{n_epochs}") as steps:
            for step in steps:
                X_batch, y_batch = random_batch(X_train, y_train)
                with tf.GradientTape(persistent=True) as tape:
                    y_pred = model(X_batch)
                    main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
                    loss = tf.add_n([main_loss] + model.losses)
                for layers, optimizer in ((lower_layers, lower_optimizer),
                                          (upper_layers, upper_optimizer)):
                    gradients = tape.gradient(loss, layers.trainable_variables)
                    optimizer.apply_gradients(zip(gradients, layers.trainable_variables))
                del tape
                for variable in model.variables:
                    if variable.constraint is not None:
                        variable.assign(variable.constraint(variable))
                status = OrderedDict()
                mean_loss(loss)
                status["loss"] = mean_loss.result().numpy()
                for metric in metrics:
                    metric(y_batch, y_pred)
                    status[metric.name] = metric.result().numpy()
                steps.set_postfix(status)
            y_pred = model(X_valid)
            status["val_loss"] = np.mean(loss_fn(y_valid, y_pred))
```

```
            status["val_accuracy"] = np.mean(tf.keras.metrics.sparse_categorical_accurac
                tf.constant(y_valid, dtype=np.float32), y_pred))
            steps.set_postfix(status)
        for metric in [mean_loss] + metrics:
            metric.reset_states()
```

```
All epochs:   0%|          | 0/5 [00:00<?, ?it/s]
Epoch 1/5:   0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 2/5:   0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 3/5:   0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 4/5:   0%|          | 0/1718 [00:00<?, ?it/s]
Epoch 5/5:   0%|          | 0/1718 [00:00<?, ?it/s]
```