TensorFlow2教程-Eager Execution

> 最全TensorFlow 2.0 入門教程持續更新：https://zhuanlan.zhihu.com/p/59507137 (https://zhuanlan.zhihu.com/p/59507137)
>
> 完整TensorFlow2.0教程代碼請看 https://github.com/czy36mengfei/tensorflow2_tutorials_chinese (https://github.com/czy36mengfei/tensorflow2_tutorials_chinese) (歡迎star)
>
> 最新TensorFlow 2教程和相關資源，請關注微信公眾號：DoitNLP， 後面我會在DoitNLP上，持續更新深度學習、NLP、Tensorflow的相關教程和前沿資訊，它將成為我們一起學習TensorFlow2的大本營。
>
> 本教程主要由tensorflow2.0官方教程的個人學習複現筆記整理而來，中文講解，方便喜歡閱讀中文教程的朋友，tensorflow官方教程：https://www.tensorflow.org (https://www.tensorflow.org)

TensorFlow 的 Eager Execution 是一種命令式程式設計環境，可立即評估操作，無需構建圖：操作會返回具體的值，而不是構建以後再運行的計算圖。這樣可以輕鬆地使用 TensorFlow 和調試模型，並且還減少了樣板代碼。

Eager Execution 是一個靈活的機器學習平臺，用於研究和實驗，可提供：

- 直觀的介面 - 自然地組織代碼結構並使用 Python 資料結構。快速反覆運算小模型和小型資料集。
- 更輕鬆的調試功能 - 直接調用操作以檢查正在運行的模型並測試更改。使用標準 Python 調試工具進行即時錯誤報告。
- 自然控制流程 - 使用 Python 控制流程而不是圖控制流程，簡化了動態模型的規範。

In [1]:
```
from __future__ import absolute_import, division, print_function

import tensorflow as tf
print(tf.__version__)
# 在tensorflow2中默認使用Eager Execution
tf.executing_eagerly()
```

```
2.0.0-beta1
```

Out[1]:
```
True
```

Eager Execution下運算

在Eager Execution下可以直接進行運算，結果會立即返回

In [2]:
```
x = [[3.]]
m = tf.matmul(x, x)
print(m)
```

```
tf.Tensor([[9.]], shape=(1, 1), dtype=float32)
```

啟用Eager Execution會改變TensorFlow操作的行為 - 現在他們會立即評估並將其值返回給Python。tf.Tensor對象引用具體值，而不再是指向計算圖中節點的符號控制碼。由於在會話中沒有構建和運行的計算圖，因此使用print()或偵錯工具很容易檢查結果。評估，列印和檢查張量值不會破壞計算梯度的流程。

Eager Execution可以與NumPy很好地協作。NumPy操作接受tf.Tensor參數。TensorFlow 數學運算將Python物件和NumPy陣列轉換為tf.Tensor對象。tf.Tensor.numpy方法將物件的值作為NumPy的ndarray類型返回。

In [3]:
```
# tf.Tensor對象引用具體值
a = tf.constant([[1,9],[3,6]])
print(a)

# 支持broadcasting（廣播：不同shape的資料進行數學運算）
b = tf.add(a, 2)
print(b)

# 支持運算子重載
print(a*b)

# 可以當做numpy資料使用
import numpy as np
s = np.multiply(a,b)
print(s)

# 轉換為numpy類型
print(a.numpy())
```

```
tf.Tensor(
[[1 9]
 [3 6]], shape=(2, 2), dtype=int32)
tf.Tensor(
[[ 3 11]
 [ 5  8]], shape=(2, 2), dtype=int32)
tf.Tensor(
[[ 3 99]
 [15 48]], shape=(2, 2), dtype=int32)
[[ 3 99]
 [15 48]]
[[1 9]
 [3 6]]
```

2.動態控制流

Eager Execution的一個主要好處是在執行模型時可以使用主機語言(Python)的所有功能。所以，例

如，寫fizzbuzz很容易：

In [4]:
```
def fizzbuzz(max_num):
    counter = tf.constant(0)
    max_num = tf.convert_to_tensor(max_num)
    # 使用range遍歷
    for num in range(1, max_num.numpy()+1):
        # 重新轉為tensor類型
        num = tf.constant(num)
        # 使用if-elif 做判斷
        if int(num % 3) == 0 and int(num % 5) == 0:
            print('FizzBuzz')
        elif int(num % 3) == 0:
            print('Fizz')
        elif int(num % 5) == 0:
            print('Buzz')
        else:
            print(num.numpy())
        counter += 1  # 自加運算
fizzbuzz(16)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
```

3.在Eager Execution下訓練

計算梯度

自動微分對於實現機器學習演算法（例如用於訓練神經網路的反向傳播）來說是很有用的。在Eager Execution中，使用 tf.GradientTape 來跟蹤操作以便稍後計算梯度。

可以用tf.GradientTape來訓練和/或計算梯度。它對複雜的訓練迴圈特別有用。

由於在每次調用期間可能發生不同的操作，所有前向傳遞操作都被記錄到“磁帶”中。要計算梯度，請向反向播放磁帶，然後丟棄。特定的tf.GradientTape只能計算一個梯度; 後續調用會引發執行階段錯誤。

In [5]:
```
w = tf.Variable([[1.0]])
# 用tf.GradientTape()記錄梯度
with tf.GradientTape() as tape:
    loss = w*w
grad = tape.gradient(loss, w)  # 計算梯度
print(grad)
```

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

訓練模型

In [6]:
```
# 導入mnist數據
(mnist_images, mnist_labels), _ = tf.keras.datasets.mnist.load_data()
# 資料轉換
dataset = tf.data.Dataset.from_tensor_slices(
  (tf.cast(mnist_images[...,tf.newaxis]/255, tf.float32),
   tf.cast(mnist_labels,tf.int64)))
# 數據打亂與分批次
dataset = dataset.shuffle(1000).batch(32)
# 使用Sequential構建一個卷積網路
mnist_model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(16,[3,3], activation='relu',
                         input_shape=(None, None, 1)),
  tf.keras.layers.Conv2D(16,[3,3], activation='relu'),
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(10)
])
# 展示數據
# 即使沒有經過培訓，也可以調用模型並在Eager Execution中檢查輸出
for images,labels in dataset.take(1):
    print("Logits: ", mnist_model(images[0:1]).numpy())

# 優化器與損失函數
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# 按批次訓練
# 雖然 keras 模型具有內置訓練迴圈（fit 方法），但有時需要更多自訂設置。下面是一個用 eag
loss_history = []
for (batch, (images, labels)) in enumerate(dataset.take(400)):
    if batch % 10 == 0:
        print('.', end='')
    with tf.GradientTape() as tape:
        # 獲取預測結果
        logits = mnist_model(images, training=True)
        # 獲取損失
        loss_value = loss_object(labels, logits)

    loss_history.append(loss_value.numpy().mean())
    # 獲取本批資料梯度
    grads = tape.gradient(loss_value, mnist_model.trainable_variables)
    # 反向傳播優化
    optimizer.apply_gradients(zip(grads, mnist_model.trainable_variables))

# 繪圖展示Loss變化
import matplotlib.pyplot as plt
plt.plot(loss_history)
plt.xlabel('Batch #')
plt.ylabel('Loss [entropy]')
```

```
Logits:  [[-0.0330125   0.00557926  0.01437856 -0.07978292 -0.05083258 -0.02875
553
  -0.02977117 -0.02533271  0.0575003   0.03532691]]
........................................
```

Out[6]: Text(0, 0.5, 'Loss [entropy]')

4.變數求導優化

tf.Variable物件存儲在訓練期間訪問的可變tf.Tensor值，以使自動微分更容易。模型的參數可以作為變數封裝在類中。

將tf.Variable 和tf.GradientTape 結合，可以更好地封裝模型參數。例如，可以重寫上面的自動微分示例為：

In [7]:
```
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.W = tf.Variable(5., name='weight')
        self.B = tf.Variable(10., name='bias')
    def call(self, inputs):
        return inputs * self.W + self.B

# 滿足函數3 * x + 2的資料
NUM_EXAMPLES = 2000
training_inputs = tf.random.normal([NUM_EXAMPLES])
noise = tf.random.normal([NUM_EXAMPLES])
training_outputs = training_inputs * 3 + 2 + noise

# 損失函數
def loss(model, inputs, targets):
    error = model(inputs) - targets
    return tf.reduce_mean(tf.square(error))

# 梯度函數
def grad(model, inputs, targets):
    with tf.GradientTape() as tape:
        loss_value = loss(model, inputs, targets)
    return tape.gradient(loss_value, [model.W, model.B])

# 模型與優化器
model = MyModel()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

print("Initial loss: {:.3f}".format(loss(model, training_inputs, training_outputs

# 訓練迴圈， 反向傳播優化
for i in range(300):
    grads = grad(model, training_inputs, training_outputs)
    optimizer.apply_gradients(zip(grads, [model.W, model.B]))
    if i % 20 == 0:
        print("Loss at step {:03d}: {:.3f}".format(i, loss(model, training_inputs

print("Final loss: {:.3f}".format(loss(model, training_inputs, training_outputs))
print("W = {}, B = {}".format(model.W.numpy(), model.B.numpy()))
```

```
Initial loss: 69.425
Loss at step 000: 66.713
Loss at step 020: 30.274
Loss at step 040: 14.048
Loss at step 060: 6.822
Loss at step 080: 3.604
Loss at step 100: 2.171
Loss at step 120: 1.533
Loss at step 140: 1.249
Loss at step 160: 1.122
Loss at step 180: 1.066
Loss at step 200: 1.040
Loss at step 220: 1.029
Loss at step 240: 1.024
```

```
Loss at step 260: 1.022
Loss at step 280: 1.021
Final loss: 1.021
W = 2.9795801639556885, B = 2.0041041374206543
```

5.Eager Execution中的物件

使用 Graph Execution 時，程式狀態（如變數）存儲在全域集合中，它們的生命週期由 tf.Session 物件管理。相反，在 Eager Execution 期間，狀態物件的生命週期由其對應的 Python 物件的生命週期決定。

變數物件

變數將持續存在，直到刪除物件的最後一個引用，然後變數被刪除。

In [8]:
```
if tf.test.is_gpu_available():
    with tf.device("gpu:0"):
        v = tf.Variable(tf.random.normal([1000, 1000]))
        v = None  # v no longer takes up GPU memory
```

基於對象的保存

tf.train.Checkpoint 可以將 tf.Variable 保存到檢查點並從中恢復：

In [9]:
```
# 使用檢測點保存變數
x = tf.Variable(6.0)
checkpoint = tf.train.Checkpoint(x=x)
# 變數的改變會同步到檢測點
x.assign(1.0)
checkpoint.save('./ckpt/')
# 檢測點保存後，變數的改變對檢測點無影響
x.assign(8.0)
checkpoint.restore(tf.train.latest_checkpoint('./ckpt/'))
print(x)#
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=1.0>
```

要保存和載入模型，tf.train.Checkpoint 會存儲物件的內部狀態，而不需要隱藏變數。要記錄 model、optimizer 和全域步的狀態，可以將它們傳遞到 tf.train.Checkpoint：

In [10]:
```
# 模型保持
import os
model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(16,[3,3], activation='relu'),
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(10)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
checkpoint_dir = './ck_model_dir'
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
# 將優化器和模型記錄至檢測點
root = tf.train.Checkpoint(optimizer=optimizer,
                           model=model)
# 保存檢測點
root.save(checkpoint_prefix)
# 讀取檢測點
root.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

Out[10]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x1496dcd9588
>
```

物件導向的指標

tf.keras.metrics存儲為物件。通過將新資料傳遞給callable來更新度量標準，並使用tf.keras.metrics.result方法檢索結果，例如：

In [11]:
```
m = tf.keras.metrics.Mean('loss')
m(0)
m(5)
print(m.result())  # => 2.5
m([8, 9])
print(m.result())  # => 5.5
```

```
tf.Tensor(2.5, shape=(), dtype=float32)
tf.Tensor(5.5, shape=(), dtype=float32)
```

6.自動微分高級內容

動態模型

tf.GradientTape 也可用於動態模型。這個回溯線搜索演算法示例看起來像普通的 NumPy 代碼，除了存在梯度並且可微分，儘管控制流比較複雜：

In [12]:
```
def line_search_step(fn, init_x, rate=1.0):
    with tf.GradientTape() as tape:
        # 變數會自動記錄，但需要手動觀察張量
        tape.watch(init_x)
        value = fn(init_x)
    grad = tape.gradient(value, init_x)
    grad_norm = tf.reduce_sum(grad * grad)
    init_value = value
    while value > init_value - rate * grad_norm:
        x = init_x - rate * grad
        value = fn(x)
        rate /= 2.0
    return x, value
```

自訂梯度

自訂梯度是在 Eager Execution 和 Graph Execution 中覆蓋梯度的一種簡單方式。在正向函數中，定義相對於輸入、輸出或中間結果的梯度。例如，下面是在反向傳播中截斷梯度範數的一種簡單方式：

In [13]:
```
@tf.custom_gradient
def clip_gradient_by_norm(x, norm):
    y = tf.identity(x)
    def grad_fn(dresult):
        return [tf.clip_by_norm(dresult, norm), None]
    return y, grad_fn
```

自訂梯度可以提供數值穩定的梯度

In [14]:
```
def log1pexp(x):
    return tf.math.log(1 + tf.exp(x))

def grad_log1pexp(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        value = log1pexp(x)
    return tape.gradient(value, x)
# 梯度計算在x = 0時工作正常。
print(grad_log1pexp(tf.constant(0.)).numpy())
# 但是，由於數值不穩定，x = 100失敗。
print(grad_log1pexp(tf.constant(100.)).numpy())
```

```
0.5
nan
```

這裡，log1pexp函數可以使用自訂梯度求導進行分析簡化。 下面的實現重用了在前向傳遞期間計算的tf.exp（x）的值 - 通過消除冗餘計算使其更有效：

In [15]:
```
@tf.custom_gradient
def log1pexp(x):
    e = tf.exp(x)
    def grad(dy):
        return dy * (1 - 1 / (1 + e))
    return tf.math.log(1 + e), grad

def grad_log1pexp(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        value = log1pexp(x)
    return tape.gradient(value, x)
# 和以前一樣，梯度計算在x = 0時工作正常。
print(grad_log1pexp(tf.constant(0.)).numpy())
# 並且梯度計算也適用於x = 100
print(grad_log1pexp(tf.constant(100.)).numpy())
```

```
0.5
1.0
```

7.使用gpu提升性能

在 Eager Execution 期間，計算會自動分流到 GPU。如果要控制計算運行的位置，可以將其放在 tf.device('/gpu:0') 塊（或 CPU 等效塊）中：

In [16]:
```
import time

def measure(x, steps):
    # TensorFlow在第一次使用時初始化GPU，其不計入時間。
    tf.matmul(x, x)
    start = time.time()
    for i in range(steps):
        x = tf.matmul(x, x)
    # tf.matmul可以在完成矩陣乘法之前返回（例如，
    # 可以在對CUDA流進行操作之後返回）。
    # 下麵的x.numpy（）調用將確保所有已排隊
    # 的操作都已完成（並且還將結果複製到主機記憶體，
    # 因此我們只包括一些matmul操作時間）
    _ = x.numpy()
    end = time.time()
    return end - start

shape = (1000, 1000)
steps = 200
print("Time to multiply a {} matrix by itself {} times:".format(shape, steps))

# 在CPU上運行:
with tf.device("/cpu:0"):
    print("CPU: {} secs".format(measure(tf.random.normal(shape), steps)))

# 在GPU上運行,如果可以的話:
if tf.test.is_gpu_available():
    with tf.device("/gpu:0"):
        print("GPU: {} secs".format(measure(tf.random.normal(shape), steps)))
else:
    print("GPU: not found")
```

```
Time to multiply a (1000, 1000) matrix by itself 200 times:
CPU: 1.698425054550171 secs
GPU: 0.13264727592468262 secs
```

tf.Tensor物件可以被複製到不同的設備來執行其操作

In [17]:
```
if tf.test.is_gpu_available():
    x = tf.random.normal([10, 10])
    # 將tensor對象複製到gpu上
    x_gpu0 = x.gpu()
    x_cpu = x.cpu()

    _ = tf.matmul(x_cpu, x_cpu)    # Runs on CPU
    _ = tf.matmul(x_gpu0, x_gpu0)  # Runs on GPU:0
```

In [ ]:
```
```