

改了第二步，不是直接选择 target DQN 上最大的 Q 值，而是在我们的主 DQN 上通过其最大 Q 值选择 Action，再去获取这个 Action 在 target DQN 上的 Q 值。这样我们的主网络负责选择 Action，而这个被选定的 Action 的 Q 值则由 target DQN 生成。被选择的 Q 值，不一定总是最大的 Q 值，这样就避免了被高估的次优 Action 总是超过最优的 Action，导致我们发现不了真正最好的 Action。我们的学习目标因此可以写成下面的式子。

$$Target = r_{t+1} + \gamma \cdot Q_{target}(s_{t+1}, \text{argmax}_a(Q_{main}(s_{t+1}, a)))$$

第 5 个 Trick 是 Dueling DQN，也是 DQN 的一个重大改进，在 Google 的论文 *Dueling Network Architectures for Deep Reinforcement Learning* 中被首次提出。Dueling DQN 将 Q 值的函数 $Q(s_t, a_t)$ 拆分为两部分，一部分是静态的环境状态本身具有的价值 $V(s_t)$ ，称为 Value；另一部分是动态的通过选择某个 Action 额外带来的价值 $A(a_t)$ ，称为 Advantage。我们的 Q 值将由这两部分组合而成，可以写成下面这个公式。

$$Q(s_t, a_t) = V(s_t) + A(a_t)$$

Dueling 的目标就是让网络可以分别计算环境本身的 Value 和选择 Action 带来的 Advantage，这里的 Advantage 是某个 Action 与其他 Action 的比较，因此我们将它设计为零均值的。如图 8-10 所示，上面那部分是传统的 DQN 网络，下面的就是 Dueling DQN 了，在网络的最后部分，不再是直接输出 Action 数量（假定为 n ）的 Q 值，而是输出一个 Value 值及 n 个 Advantage 值，然后将 V 值分别加到每一个 Advantage 值上，得到最后的结果。这样做的目的是让 DQN 的学习目标更明确，如果当前的期望价值主要是由环境状态决定的，那么 Value 值很大，而所有 Advantage 的波动都不大；如果期望价值主要由 Action 决定，那么 Value 值很小，而 Advantage 波动会很大，分解这两个部分会让我们的学习目标更稳定、更精确，让 DQN 对环境状态的估计能力更强。

下面我们就实现带有前面几个 Trick 的 DQN。使用的任务环境是叫作 GridWorld 的导航类游戏，如图 8-11 所示。GridWorld 中包含一个 hero（实际为蓝色，这里以白色显示）4 个 goal（实际为绿色，这里以浅灰表示）和 2 个 fire（实际为红色，这里以深灰色表示）。我们的目标就是控制 hero 移动，每次向上、下、左、右等方向移动一步，尽可能多地触碰 goal（奖励值为 1），同时避开 fire（奖励值为 -1）。游戏的目标是在限定步数内拿到最多的分数。我们的 Agent 将直接通过 GridWorld 的图像学习控制 hero 移动的最优策略。

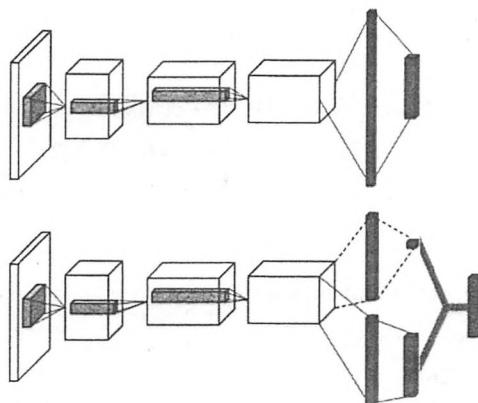


图 8-10 Dueling DQN 拆分了 value function 和 advantage function

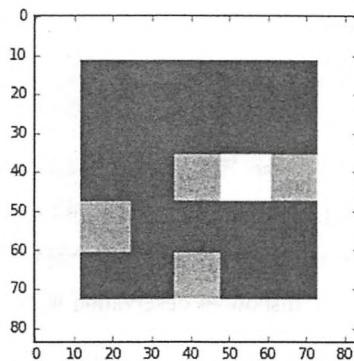


图 8-11 GridWorld 游戏环境示例

下面开始创建 GridWorld 任务的环境。首先是载入各种依赖的库，这次需要载入的库相对较多，其中 `itertools` 可以方便地进行迭代操作，`scipy.misc` 和 `matplotlib.pyplot` 可以绘图。同时因为训练时间较长，我们也载入 `os` 用来定期储存模型文件。本节代码主要来自 DeepRL-Agents 的开源实现⁶⁶。

```
import numpy as np
import random
import itertools
import scipy.misc
import matplotlib.pyplot as plt
import tensorflow as tf
```

```
import os
%matplotlib inline
```

先是创建环境内物体对象的 class，环境物体包括以下几个属性：coordinates (x,y 坐标)、size (尺寸)、intensity (亮度值)、channel (RGB 颜色通道)、reward (奖励值)，以及 name (名称)。

```
class gameOb():
    def __init__(self, coordinates, size, intensity, channel, reward, name):
        self.x = coordinates[0]
        self.y = coordinates[1]
        self.size = size
        self.intensity = intensity
        self.channel = channel
        self.reward = reward
        self.name = name
```

然后创建 GridWorld 环境的 class，其初始化方法只需要传入一个参数，即环境的 size。我们将环境的长和宽都设为输入的 size，同时将环境的 Action Space 设为 4，并初始化环境的物体对象的列表。调用 self.reset() 方法重置整个环境，得到初始的 observation (即 GridWorld 的图像)，并使用 plt.imshow 将 observation 展示出来。

```
class gameEnv():
    def __init__(self, size):
        self.sizeX = size
        self.sizeY = size
        self.actions = 4
        self.objects = []
        a = self.reset()
        plt.imshow(a, interpolation="nearest")
```

接下来定义环境的 reset 方法。我们将创建所有 GridWorld 中的物体，包括 1 个 hero (用户控制的对象)、4 个 goal (reward 为 1)、2 个 fire (reward 为 -1)，并把他们添加到物体对象的列表 self.objects。创建物体的位置时使用 self.newPosition()，该方法会随机选择一个没有被占用的新位置。所有物体的 size 和 intensity 均为 1，其中 hero 的 channel 为 2

(蓝色),goal的channel为1(绿色),fire的channel为0(红色)。最后我们使用self.renderEnv()将GridWorld的图像绘制出来,即state。

```
def reset(self):
    self.objects = []
    hero = gameOb(self newPosition(), 1, 1, 2, None, 'hero')
    self.objects.append(hero)
    goal = gameOb(self newPosition(), 1, 1, 1, 1, 'goal')
    self.objects.append(goal)
    hole = gameOb(self newPosition(), 1, 1, 0, -1, 'fire')
    self.objects.append(hole)
    goal2 = gameOb(self newPosition(), 1, 1, 1, 1, 'goal')
    self.objects.append(goal2)
    hole2 = gameOb(self newPosition(), 1, 1, 0, -1, 'fire')
    self.objects.append(hole2)
    goal3 = gameOb(self newPosition(), 1, 1, 1, 1, 'goal')
    self.objects.append(goal3)
    goal4 = gameOb(self newPosition(), 1, 1, 1, 1, 'goal')
    self.objects.append(goal4)
    state = self.renderEnv()
    self.state = state
    return state
```

这里我们实现移动英雄角色的方法,我们传入的值为0、1、2、3这四个数字,分别代表上、下、左、右。函数根据输入来操作英雄的移动,但如果移动该方向会导致英雄出界,则不会进行任何移动。

```
def moveChar(self,direction):
    hero = self.objects[0]
    heroX = hero.x
    heroY = hero.y
    if direction == 0 and hero.y >= 1:
        hero.y -= 1
    if direction == 1 and hero.y <= self.sizeY-2:
        hero.y += 1
```

```

if direction == 2 and hero.x >= 1:
    hero.x -= 1
if direction == 3 and hero.x <= self.sizeX-2:
    hero.x += 1
self.objects[0] = hero

```

然后定义刚才提到的 newPosition 方法，它可以选择一个跟现有物体不冲突的位置。itertools.product 方法可以得到几个变量的所有组合，使用这个方法创建环境 size 允许的所有位置的集合 points，并获取目前所有物体位置的集合 currentPositions，再从 points 中去掉 currentPositions，剩下的就是可用的位置。最后使用 np.random.choice 随机抽取一个可用位置并返回。

```

def newPosition(self):
    iterables = [ range(self.sizeX), range(self.sizeY) ]
    points = []
    for t in itertools.product(*iterables):
        points.append(t)
    currentPositions = []
    for objectA in self.objects:
        if (objectA.x,objectA.y) not in currentPositions:
            currentPositions.append((objectA.x,objectA.y))
    for pos in currentPositions:
        points.remove(pos)
    location = np.random.choice(range(len(points)),replace=False)
    return points[location]

```

下面定义 checkGoal 函数，用来检查 hero 是否触碰了 goal 或者 fire。我们先从 objects 中获取 hero，并将其他物体对象放到 others 列表中。然后遍历 others 列表，如果有物体和坐标与 hero 完全一致，那么可判定为触碰。接下来根据触碰到的是什么物体，我们销毁该物体，并调用 self.newPosition() 方法在随机位置重新生成一个该物体，并返回这个物体的 reward 值 (goal 为 1, fire 为 -1)。

```

def checkGoal(self):
    others = []
    for obj in self.objects:

```

```

if obj.name == 'hero':
    hero = obj
else:
    others.append(obj)
for other in others:
    if hero.x == other.x and hero.y == other.y:
        self.objects.remove(other)
        if other.reward == 1:
            self.objects.append(gameOb(self newPosition(),1,1,1,1,
                                         'goal'))
        else:
            self.objects.append(gameOb(self newPosition(),1,1,0,-1,
                                         'fire'))
    return other.reward, False
return 0.0, False

```

先创建一个长宽为 size+2，颜色通道数为 3 的图片，初始值全部为 1，代表全为白色。然后把最外边一圈内部的像素的颜色值全部赋为 0，代表黑色。遍历物体对象的列表 self.objects，并设置这些物体的亮度值。同时，使用 scipy.misc.imresize 将图像从原始大小 resize 为 84×84×3 的尺寸，即一个正常的游戏图像尺寸。

```

def renderEnv(self):
    a = np.ones([self.sizeY+2,self.sizeX+2,3])
    a[1:-1,1:-1,:] = 0
    hero = None
    for item in self.objects:
        a[item.y+1:item.y+item.size+1,item.x+1:item.x+item.size+1,
          item.channel] = item.intensity
    b = scipy.misc.imresize(a[:, :, 0], [84, 84, 1], interp='nearest')
    c = scipy.misc.imresize(a[:, :, 1], [84, 84, 1], interp='nearest')
    d = scipy.misc.imresize(a[:, :, 2], [84, 84, 1], interp='nearest')
    a = np.stack([b,c,d],axis=2)
    return a

```

最后定义在 GridWorld 环境中执行一步 Action 的方法。输入的参数为 Action，先使用 self.moveChar(action) 移动 hero 的位置，再使用 self.checkGoal() 检测 hero 是否有触碰物体，并得到 reward 和 done 标记。然后使用 self.renderEnv 获取环境的图像 state，最后返回 state、reward 和 done。

```
def step(self,action):
    self.moveChar(action)
    reward,done = self.checkGoal()
    state = self.renderEnv()
    return state,reward,done
```

接下来调用刚才写好的 gameEnv 类的初始化方法，并设置 size 为 5，创建一个 5×5 大小的 GridWorld 环境，每一次创建的 GridWorld 环境都是随机生成的。读者可以尝试使用不同尺寸的 GridWorld，小尺寸的环境会相对容易学习，大尺寸的则较难，训练时间也更长。

```
env = gameEnv(size=5)
```

下面便是我们创建好的 5×5 的 GridWorld 环境图像，如图 8-12 所示，因为黑白印刷的原因，其中白色代表 hero，浅灰色代表 goal（reward 为 1），深灰色代表 fire（reward 为 -1）。我们的任务目标是在指定步数（每一步可以选择向上、下、左、右移动）内获得尽可能多的分数，我们每触碰一个物体，将会销毁该物体并在其他位置重建。因此，Agent 的目标就是避开 fire，同时多触碰 goal。我们还需要规划最优路线，在有限步数内收集尽可能多的 goal。当然，这些策略都是 DQN 需要自己通过试验来学习的。

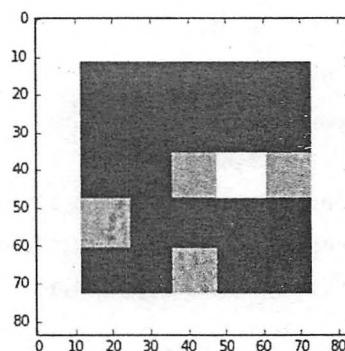


图 8-12 5×5 的 GridWorld 环境，白色为 hero，浅灰色为 goal，深灰色为 fire

下面我们就开始设计 DQN (Deep Q-Network) 网络, 相对上一节的简单例子, 本节的网络更复杂一些, 并且使用了卷积层, 可以直接从环境的原始像素中学习策略。输入 scalarInput 是被扁平化的长为 $84 \times 84 \times 3 = 21168$ 的向量, 需要先将其恢复成 $[-1, 84, 84, 3]$ 尺寸的图片 ImageIn。我们使用 tf.contrib.layers.convolution2d 创建第 1 个卷积层, 卷积核尺寸为 8×8 , 步长为 4×4 , 输出通道数 (filter 的数量) 为 32, padding 模式为 VALID (以下所有层 padding 模式均为 VALID), bias 初始化器为空。因为使用了 4×4 的步长和 VALID 模式的 padding, 所以第一层卷积的输出维度为 $20 \times 20 \times 32$ 。第 2 个卷积层尺寸为 4×4 , 步长为 2×2 , 输出通道数为 64, 这一层的输出维度为 $9 \times 9 \times 64$ 。第 3 层卷积层尺寸为 3×3 , 步长为 1×1 , 输出通道数为 64, 这一层输出维度为 $7 \times 7 \times 64$ 。第 4 层卷积尺寸为 7×7 , 步长为 1×1 , 输出通道数一下涨到了 512, 这一层的空间尺寸只允许在一个位置进行卷积, 因此最后的输出维度变为 $1 \times 1 \times 512$ 。

```
class Qnetwork():
    def __init__(self,h_size):
        self.scalarInput = tf.placeholder(shape=[None,21168],
                                         dtype=tf.float32)
        self.imageIn = tf.reshape(self.scalarInput,shape=[-1,84,84,3])
        self.conv1 = tf.contrib.layers.convolution2d(
            inputs=self.imageIn,num_outputs=32,
            kernel_size=[8,8],stride=[4,4],
            padding='VALID', biases_initializer=None)
        self.conv2 = tf.contrib.layers.convolution2d(
            inputs=self.conv1,num_outputs=64,kernel_size=[4,4],stride=[2,2],
            padding='VALID', biases_initializer=None)
        self.conv3 = tf.contrib.layers.convolution2d(
            inputs=self.conv2,num_outputs=64,kernel_size=[3,3],stride=[1,1],
            padding='VALID', biases_initializer=None)
        self.conv4 = tf.contrib.layers.convolution2d(
            inputs=self.conv3,num_outputs=512,
            kernel_size=[7,7],stride=[1,1],
            padding='VALID', biases_initializer=None)
```

接下来, 使用 tf.split() 将第 4 个卷积层的输出 conv4 平均拆分成两段, streamAC 和

streamVC，即 Dueling DQN 中的 Advantage Function（Action 带来的价值）和 Value Function（环境本身的价值）。这里注意 `tf.split` 函数的第 2 个参数代表要拆分成几段，第 3 个参数代表要拆分的是第几个维度。然后分别使用 `tf.contrib.layers.flatten` 将 streamAC 和 streamVC 转为扁平的 streamA 和 streamV。下面创建 streamA 和 streamV 的线性全连接层参数 AW 和 VW，我们直接使用 `tf.random_normal` 初始化它们的权重，再使用 `tf.matmul` 做全连接层的矩阵乘法，得到 `self.Advantage` 和 `self.Value`。因为 Advantage 是针对 Action 的，因此输出数量为 Action 的数量，而 Value 则是针对环境统一的，输出数量为 1。我们的 Q 值则由 Value 和 Advantage 复合而成，即 Value 加上减去均值的 Advantage。Advantage 减去均值的操作使用的是 `tf.subtract`，均值计算使用的是 `tf.reduce_mean` 函数（`reduce_indices` 为 1，即代表 Action 数量的维度）。最后输出的 Action 即为 Q 值最大的 Action，这里使用 `tf.argmax` 求出这个 Action。

```

self.streamAC, self.streamVC = tf.split(self.conv4, 2, 3)
self.streamA = tf.contrib.layers.flatten(self.streamAC)
self.streamV = tf.contrib.layers.flatten(self.streamVC)
self.AW = tf.Variable(tf.random_normal([h_size//2, env.actions]))
self.VW = tf.Variable(tf.random_normal([h_size//2, 1]))
self.Advantage = tf.matmul(self.streamA, self.AW)
self.Value = tf.matmul(self.streamV, self.VW)

self.Qout = self.Value + tf.subtract(self.Advantage, tf.reduce_mean(
    self.Advantage, reduction_indices=1, keep_dims=True))
self.predict = tf.argmax(self.Qout, 1)

```

我们定义 Double DQN 中的目标 Q 值 `targetQ` 的输入 `placeholder`，以及 Agent 的动作 `actions` 的输入 `placeholder`。在计算目标 Q 值时，`action` 由主 DQN 选择，Q 值则由辅助的 target DQN 生成。在计算预测 Q 值时，我们将 scalar 形式的 `actions` 转为 onehot 编码的形式，然后将主 DQN 生成的 `Qout` 乘以 `actions_onehot`，得到预测 Q 值（`Qout` 和 `actions` 都来自主 DQN）。

```

self.targetQ = tf.placeholder(shape=[None], dtype=tf.float32)
self.actions = tf.placeholder(shape=[None], dtype=tf.int32)
self.actions_onehot = tf.one_hot(self.actions, env.actions,
                                dtype=tf.float32)

```

```
self.Q = tf.reduce_sum(tf.multiply(self.Qout, self.actions_onehot),
                      reduction_indices=1)
```

接下来定义 loss，使用 tf.square 和 tf.reduce_mean 计算 targetQ 和 Q 的均方误差，并使用学习速率为 1e-4 的 Adam 优化器优化预测 Q 值和目标 Q 值的偏差。

```
self.td_error = tf.square(self.targetQ - self.Q)
self.loss = tf.reduce_mean(self.td_error)
self.trainer = tf.train.AdamOptimizer(learning_rate= 0.0001)
self.updateModel = self.trainer.minimize(self.loss)
```

接下来实现前面提到的 Experience Replay 策略。我们定义 experience_buffer 的 class，其初始化需要定义 buffer_size 即存储样本的最大容量，并创建 buffer 的列表。然后定义向 buffer 中添加元素的方法，如果超过了 buffer 的最大容量，就清空前面最早的一些样本，并在列表末尾添加新元素。然后在定义对样本进行抽样的方法，这里直接使用 random.sample() 函数随机抽取一定数量的样本。

```
class experience_buffer():
    def __init__(self, buffer_size = 50000):
        self.buffer = []
        self.buffer_size = buffer_size

    def add(self,experience):
        if len(self.buffer) + len(experience) >= self.buffer_size:
            self.buffer[0:(len(experience)+len(self.buffer)) - \
                       self.buffer_size] = []
        self.buffer.extend(experience)

    def sample(self,size):
        return np.reshape(np.array(random.sample(self.buffer,size)),
                        [size,5])
```

下面定义将 $84 \times 84 \times 3$ 的 states 扁平化为 1 维向量的函数 processState，这样做的主要目的是后面堆叠样本时会比较方便。

```
def processState(states):
```

```
return np.reshape(states,[21168])
```

这里的 updateTargetGraph 函数是更新 target DQN 模型参数的方法（主 DQN 则是直接使用 DQN class 中的 self.updateModel 方法更新模型参数）。我们的输入变量 tfVars 是 TensorFlow Graph 中的全部参数，tau 是 target DQN 向主 DQN 学习的速率。函数 updateTargetGraph 会取 tfVars 中前一半参数，即主 DQN 的模型参数，再令辅助的 target DQN 的参数朝向主 DQN 的参数前进一个很小的比例（即 tau，一般设为 0.001），这样做是让 target DQN 缓慢地学习主 DQN。我们在训练时，目标 Q 值不能在几次迭代间波动太大，否则训练会非常不稳定并且失控，陷入目标 Q 值和预测 Q 值之间的反馈循环中。因此，需要使用稳定的目标 Q 值训练主网络，所以我们使用一个缓慢学习的 target DQN 网络输出目标 Q 值，并让主网络来优化目标 Q 值和预测 Q 值间的 loss，再让 target DQN 跟随主 DQN 并缓慢学习。函数 updateTargetGraph 会创建更新 target DQN 模型参数的操作，而函数 updateTarget 则直接执行这些操作。

```
def updateTargetGraph(tfVars,tau):
    total_vars = len(tfVars)
    op_holder = []
    for idx,var in enumerate(tfVars[0:total_vars//2]):
        op_holder.append(tfVars[idx+total_vars//2].assign((var.value() * \
            tau) + ((1-tau)*tfVars[idx+total_vars//2].value())))
    return op_holder

def updateTarget(op_holder,sess):
    for op in op_holder:
        sess.run(op)
```

下面是 DQN 网络及其训练过程的一些参数。batch_size 即每次从 experience buffer 中获取多少样本，设为 32；更新频率 update_freq，即每隔多少 step 执行一次模型参数更新，设为 4；Q 值的衰减系数（discount factor）y 设为 0.99；startE 为起始的执行随机 Action 的概率；endE 为最终的执行随机 Action 的概率（在训练时，我们始终需要一些随机 Action 进行探索，实际预测时则没有必要）；annealing_steps 是从初始随机概率降到最终随机概率所需要的步数；num_episodes 指总共进行多少次 GridWorld 环境的试验；pre_train_steps 代表正式使用 DQN 选择 Action 前进行多少步随机 Action 的测试；max_epLength 是每个 episode 进行多少步 Action；load_model 代表是否读取之前训练的模型；path 是模型储存

的路径；`h_size` 是 DQN 网络最后的全连接层的隐含节点数；`tau` 是 target DQN 向主 DQN 学习的速率。

```
batch_size = 32
update_freq = 4
y = .99
startE = 1
endE = 0.1
annealing_steps = 10000.
num_episodes = 10000
pre_train_steps = 10000
max_epLength = 50
load_model = False
path = "./dqn"
h_size = 512
tau = 0.001
```

我们使用前面写好的 `Qnetwork` 类初始化 `mainQN` 和辅助的 `targetQN`，并初始化所有模型参数。同时，使用 `trainables` 获取所有可训练的参数，并使用 `updateTargetGraph` 创建更新 target DQN 模型参数的操作。

```
mainQN = Qnetwork(h_size)
targetQN = Qnetwork(h_size)
init = tf.global_variables_initializer()

trainables = tf.trainable_variables()
targetOps = updateTargetGraph(trainables, tau)
```

我们使用前面定义的 `experience_buffer` 创建 `experience replay` 的 class，设置当前随机 Action 的概率 `e`，并计算 `e` 在每一步应该衰减的值 `stepDrop`。接着初始化储存每一个 episode 的 reward 的列表 `rList`，总步数为 `total_steps`。然后创建模型训练的保存器（`Saver`），并检查保存目录是否存在。

```
myBuffer = experience_buffer()

e = startE
```

```

stepDrop = (startE - endE)/annealing_steps

rList = []
total_steps = 0

saver = tf.train.Saver()
if not os.path.exists(path):
    os.makedirs(path)

```

接下来创建默认的 Session，如果 load_model 标志为 True，那么检查模型文件路径的 checkpoint，读取并载入之前已保存的模型。接着，我们执行参数初始化的操作，并执行更新 targetQN 模型参数的操作。然后创建进行 GridWorld 试验的循环，并创建每个 episode 内部的 experience_buffer，这些内部的 buffer 不会参与当前迭代的训练，训练只会使用之前 episode 的样本。同时，初始化环境得到第一个环境信息 s，并使用 processState() 函数将其扁平化。我们初始化默认的 done 标记 d、episode 内总 reward 值 rAll，以及 episode 内的步数 j。

```

with tf.Session() as sess:
    if load_model == True:
        print('Loading Model...')
        ckpt = tf.train.get_checkpoint_state(path)
        saver.restore(sess, ckpt.model_checkpoint_path)
    sess.run(init)
    updateTarget(targetOps, sess)
    for i in range(num_episodes+1):
        episodeBuffer = experience_buffer()
        s = env.reset()
        s = processState(s)
        d = False
        rAll = 0
        j = 0

```

接着创建一个内层循环，每一次迭代执行一次 Action。当总步数小于 pre_train_steps 时，强制使用随机 Action，相当于只从随机 Action 学习，但不去强化其过程。达到

pre_train_steps 后，我们会保留一个较小的概率去随机选择 Action。若不随机选择 Action，则传入当前状态 s 给主 DQN，预测得到应该执行的 Action。然后使用 env.step() 执行一步 Action，并得到接下来的状态 s1、reward 和 done 标记。我们使用 processState 对 s1 进行扁平化处理，然后将 s、a、r、s1、d 等结果传入 episodeBuffer 中存储。

```
while j < max_epLength:
    j+=1
    if np.random.rand(1) < e or total_steps < pre_train_steps:
        a = np.random.randint(0,4)
    else:
        a = sess.run(mainQN.predict,
                    feed_dict={mainQN.scalarInput:[s]})[0]
    s1,r,d = env.step(a)
    s1 = processState(s1)
    total_steps += 1
    episodeBuffer.add(np.reshape(np.array([s,a,r,s1,d]),[1,5]))
```

当总步数超过 pre_train_steps 时，我们持续降低随机选择 Action 的概率 e，直到达到其最低值 endE。并且每当总步数达到 update_freq 的整数倍时，我们进行一次训练，即模型参数的更新。首先是从 myBuffer 中 sample 出一个 batch_size 的样本，然后将训练样本中第 3 列信息，即下一个状态 s1，传入 mainQN 并执行 main.predict，得到主模型选择的 Action。再将 s1 传入辅助的 targetQN，并得到 s1 状态下所有 Action 的 Q 值。接下来，使用 mainQN 的输出 Action，选择 targetQN 输出的 Q，得到 doubleQ。这里使用两个 DQN 网络把选择 Action 和输出 Q 值两个操作分隔开来的做法，正是 Double DQN 的方法。然后使用训练样本的第 2 列信息，即当前的 reward，加上 doubleQ 乘以衰减系数 y，得到我们的学习目标 targetQ。接着，传入当前的状态 s，学习目标 targetQ 和这一步实际采取的 Action，执行 updateModel 操作更新一次主模型 mainQN 的参数（即执行一次训练操作）。同时也调用 updateTarget 函数，执行一次 targetQN 模型参数的更新（缓慢地向 mainQN 学习），这样就完整地完成了一次训练过程。同时，在每个 step 结束时，累计当前这步获取的 reward，并更新当前状态为下一步试验做准备。如果 done 标记为 True，我们直接中断这个 episode 的试验。

```
if total_steps > pre_train_steps:
    if e > endE:
```

```

        e -= stepDrop

    if total_steps % (update_freq) == 0:
        trainBatch = myBuffer.sample(batch_size)
        A = sess.run(mainQN.predict, feed_dict={
            mainQN.scalarInput: np.vstack(trainBatch[:, 3])})
        Q = sess.run(targetQN.Qout, feed_dict={
            targetQN.scalarInput: np.vstack(trainBatch[:, 3])})
        doubleQ = Q[range(batch_size), A]
        targetQ = trainBatch[:, 2] + y * doubleQ
        _ = sess.run(mainQN.updateModel, feed_dict={
            mainQN.scalarInput: np.vstack(trainBatch[:, 0]),
            mainQN.targetQ: targetQ,
            mainQN.actions: trainBatch[:, 1]})

        updateTarget(targetOps, sess)

    rAll += r
    s = s1

    if d == True:
        break

```

我们将 episode 内部的 episodeBuffer 添加到 myBuffer 中，用作以后训练抽样的数据集，并将当前 episode 的 reward 添加到 rList 中。然后，每 25 个 episode 就展示一次它们平均的 reward 值，同时每 1000 个 episode 或全部训练完成后，保存当前模型。

```

myBuffer.add(episodeBuffer.buffer)
rList.append(rAll)
if i > 0 and i % 25 == 0:
    print('episode', i, ', average reward of last 25 episode',
          np.mean(rList[-25:]))
if i > 0 and i % 1000 == 0:
    saver.save(sess, path+'model-'+str(i)+'.cptk')
    print("Saved Model")
saver.save(sess, path+'model-'+str(i)+'.cptk')

```

在初始的 200 个 episode 内，即完全随机 Action 的前 10000 步内，平均可以获得 reward 在 2 附近，这是基础的 baseline。

```
episode 25 , average reward of last 25 episode 2.52
episode 50 , average reward of last 25 episode 2.32
episode 75 , average reward of last 25 episode 1.68
episode 100 , average reward of last 25 episode 1.92
episode 125 , average reward of last 25 episode 2.16
episode 150 , average reward of last 25 episode 2.28
episode 175 , average reward of last 25 episode 1.6
episode 200 , average reward of last 25 episode 2.36
```

这是训练到最后一些 episode 的输出，平均 reward 已经涨到了 22 左右，相比之前的 baseline 是非常大的提升。

```
episode 9750 , average reward of last 25 episode 23.36
episode 9775 , average reward of last 25 episode 22.8
episode 9800 , average reward of last 25 episode 22.36
episode 9825 , average reward of last 25 episode 22.68
episode 9850 , average reward of last 25 episode 22.0
episode 9875 , average reward of last 25 episode 22.96
episode 9900 , average reward of last 25 episode 22.08
episode 9925 , average reward of last 25 episode 21.88
episode 9950 , average reward of last 25 episode 22.08
episode 9975 , average reward of last 25 episode 22.2
```

计算每 100 个 episodes 的平均 reward，并使用 plt.plot 展示 reward 变化的趋势。

```
rMat = np.resize(np.array(rList),[len(rList)//100,100])
rMean = np.average(rMat,1)
plt.plot(rMean)
```

如图 8-13 所示，我们可以看到从第 1000 个 episode 开始，reward 快速提升，到第 4000 个 episode 时基本达到了高峰，后面进入平台期，没有太大提升。

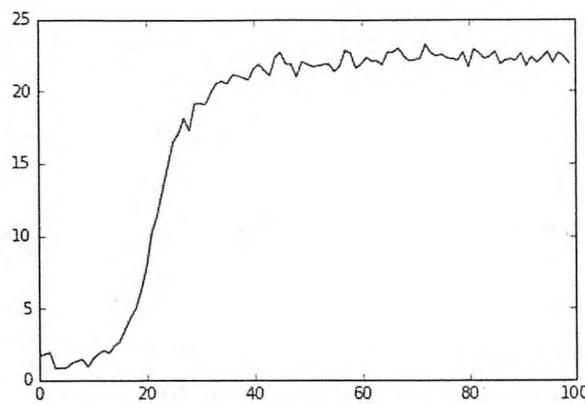
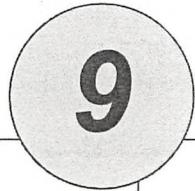


图 8-13 训练过程中 reward 的变化趋势

本节中讲述了 DQN 的基本原理，和使用 DQN 的几个非常重要的 Trick。目前 DQN 的研究仍在快速发展中，已经有越来越多新的技术被应用到 DQN 中。DQN 首次被提出了，在 Atari 2600 游戏中展示出了惊人的表现，并直接引发了深度强化学习的热潮。相信在未来，DQN 或 Value Network 会继续在更多地方发挥出强大的作用。



9

TensorBoard、多 GPU 并行及分布式并行

9.1 TensorBoard

TensorBoard 是 TensorFlow 官方推出的可视化工具，如图 9-1 所示，它可以将模型训练过程中的各种汇总数据展示出来，包括标量（Scalars）、图片（Images）、音频（Audio）、计算图（Graphs）、数据分布（Distributions）、直方图（Histograms）和嵌入向量（Embeddings）。我们在使用 TensorFlow 训练大型深度学习神经网络时，中间的计算过程可能非常复杂，因此为了理解、调试和优化我们设计的网络，可以使用 TensorBoard 观察训练过程中的各种可视化数据。如果要使用 TensorBoard 展示数据，我们需要在执行 TensorFlow 计算图的过程中，将各种类型的数据汇总并记录到日志文件中。然后使用 TensorBoard 读取这些日志文件，解析数据并生成数据可视化的 Web 页面，让我们可以在浏览器中观察各种汇总数据。下面我们将通过一个简单的 MNIST 手写数字识别的例子，讲解各种类型数据的汇总和展示的方法。

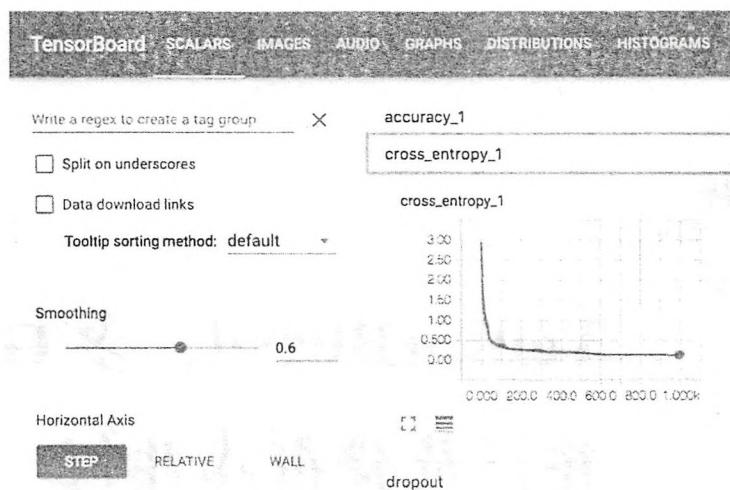


图 9-1 TensorBoard——基于 Web 的 TensorFlow 数据可视化工具

我们首先载入 TensorFlow，并设置训练的最大步数为 1000，学习速率为 0.001，dropout 的保留比率为 0.9。同时，设置 MNIST 数据的下载地址 `data_dir` 和汇总数据的日志存放路径 `log_dir`。这里的日志路径 `log_dir` 非常重要，会存放所有汇总数据供 TensorBoard 展示。本节代码主要来自 TensorFlow 的开源实现⁶⁷。

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
max_steps=1000
learning_rate=0.001
dropout=0.9
data_dir='/tmp/tensorflow/mnist/input_data'
log_dir='/tmp/tensorflow/mnist/logs/mnist_with_summaries'
```

我们使用 `input_data.read_data_sets` 下载 MNIST 数据，并创建 TensorFlow 的默认 Session。

```
mnist = input_data.read_data_sets(data_dir, one_hot=True)
sess = tf.InteractiveSession()
```

为了在 TensorBoard 中展示节点名称，我们设计网络时会经常使用 `with tf.name_scope` 限定命名空间，在这个 `with` 下的所有节点都会被自动命名为 `input/xxx` 这样的格式。下面定义输入 `x` 和 `y` 的 `placeholder`，并将输入的一维数据变形为 28×28 的图片储存到另一个

tensor，这样就可以使用 `tf.summary.image` 将图片数据汇总给 TensorBoard 展示了。

```
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None, 784], name='x-input')
    y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')

with tf.name_scope('input_reshape'):
    image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])
    tf.summary.image('input', image_shaped_input, 10)
```

同时，定义神经网络模型参数的初始化方法，权重依然使用我们常用的 `truncated_normal` 进行初始化，偏置则赋值为 0.1。

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

再定义对 `Variable` 变量的数据汇总函数，我们计算出 `Variable` 的 `mean`、`stddev`、`max` 和 `min`，对这些标量数据使用 `tf.summary.scalar` 进行记录和汇总。同时，使用 `tf.summary.histogram` 直接记录变量 `var` 的直方图数据。

```
def variable_summaries(var):
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
            tf.summary.scalar('stddev', stddev)
            tf.summary.scalar('max', tf.reduce_max(var))
            tf.summary.scalar('min', tf.reduce_min(var))
            tf.summary.histogram('histogram', var)
```

然后我们设计一个 MLP 多层神经网络来训练数据，在每一层中都会对模型参数进行数据汇总。因此，我们定义创建一层神经网络并进行数据汇总的函数 nn_layer。这个函数的输入参数有输入数据 input_tensor、输入的维度 input_dim、输出的维度 output_dim 和层名称 layer_name，激活函数 act 则默认使用 ReLU。在函数内，先是初始化这层神经网络的权重和偏重，并使用前面定义的 variable_summaries 对 variable 进行数据汇总。然后对输入做矩阵乘法并加偏置，再将未进行激活的结果使用 tf.summary.histogram 统计直方图。同时，在使用激活函数后，再使用 tf.summary.histogram 统计一次。

```
def nn_layer(input_tensor, input_dim, output_dim, layer_name,
             act=tf.nn.relu):
    with tf.name_scope(layer_name):
        with tf.name_scope('weights'):
            weights = weight_variable([input_dim, output_dim])
            variable_summaries(weights)
        with tf.name_scope('biases'):
            biases = bias_variable([output_dim])
            variable_summaries(biases)
        with tf.name_scope('Wx_plus_b'):
            preactivate = tf.matmul(input_tensor, weights) + biases
            tf.summary.histogram('pre_activations', preactivate)
            activations = act(preactivate, name='activation')
            tf.summary.histogram('activations', activations)
    return activations
```

我们使用刚刚定义好的 nn_layer 创建一层神经网络，输入维度是图片的尺寸 ($784=28 \times 28$)，输出的维度是隐藏节点数 500。再创建一个 Dropout 层，并使用 tf.summary.scalar 记录 keep_prob。然后再使用 nn_layer 定义神经网络的输出层，其输入维度为上一层的隐含节点数 500，输出维度为类别数 10，同时激活函数为全等映射 identity，即暂不使用 Softmax，在后面会处理。

```
hidden1 = nn_layer(x, 784, 500, 'layer1')

with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
```

```

tf.summary.scalar('dropout_keep_probability', keep_prob)
dropped = tf.nn.dropout(hidden1, keep_prob)

y = nn_layer(dropped, 500, 10, 'layer2', act=tf.identity)

```

这里使用 `tf.nn.softmax_cross_entropy_with_logits()` 对前面输出层的结果进行 Softmax 处理并计算交叉熵损失 `cross_entropy`。我们计算平均的损失，并使用 `tf.summary.scalar` 进行统计汇总。

```

with tf.name_scope('cross_entropy'):
    diff = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)
    with tf.name_scope('total'):
        cross_entropy = tf.reduce_mean(diff)
tf.summary.scalar('cross_entropy', cross_entropy)

```

下面使用 `Adma` 优化器对损失进行优化，同时统计预测正确的样本数并计算正确率 `accuracy`，再使用 `tf.summary.scalar` 对 `accuracy` 进行统计汇总。

```

with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    with tf.name_scope('accuracy'):
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', accuracy)

```

因为我们之前定义了非常多的 `tf.summary` 的汇总操作，逐一执行这些操作太麻烦，所以这里使用 `tf.summary.merge_all()` 直接获取所有汇总操作，以便后面执行。然后，定义两个 `tf.summary.FileWriter`（文件记录器）在不同的子目录，分别用来存放训练和测试的日志数据。同时，将 Session 的计算图 `sess.graph` 加入训练过程的记录器，这样在 TensorBoard 的 GRAPHS 窗口中就能展示整个计算图的可视化效果。最后使用 `tf.global_variables_initializer().run()` 初始化全部变量。

```

merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter(log_dir + '/train', sess.graph)

```

```
test_writer = tf.summary.FileWriter(log_dir + '/test')
tf.global_variables_initializer().run()
```

接下来定义 feed_dict 的损失函数。该函数先判断训练标记，如果训练标记为 True，则从 mnist.train 中获取一个 batch 的样本，并设置 dropout 值；如果训练标记为 False，则获取测试数据，并设置 keep_prob 为 1，即等于没有 dropout 效果。

```
def feed_dict(train):
    if train:
        xs, ys = mnist.train.next_batch(100)
        k = dropout
    else:
        xs, ys = mnist.test.images, mnist.test.labels
        k = 1.0
    return {x: xs, y_: ys, keep_prob: k}
```

最后一步，实际执行具体的训练、测试及日志记录的操作。首先使用 tf.train.Saver() 创建模型的保存器。然后进入训练的循环中，每隔 10 步执行一次 merged（数据汇总）、accuracy（求测试集上的预测准确率）操作，并使用 test_writer.add_summary 将汇总结果 summary 和循环步数 i 写入日志文件；同时每隔 100 步，使用 tf.RunOptions 定义 TensorFlow 运行选项，其中设置 trace_level 为 FULL_TRACE，并使用 tf.RunMetadata() 定义 TensorFlow 运行的元信息，这样可以记录训练时运算时间和内存占用等方面的信息。再执行 merged 数据汇总操作和 train_step 训练操作，将汇总结果 summary 和训练元信息 run_metadata 添加到 train_writer。平时，则只执行 merged 操作和 train_step 操作，并添加 summary 到 train_writer。所有训练全部结束后，关闭 train_writer 和 test_writer。

```
saver = tf.train.Saver()
for i in range(max_steps):
    if i % 10 == 0:
        summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
        test_writer.add_summary(summary, i)
        print('Accuracy at step %s: %s' % (i, acc))
    else:
        if i % 100 == 99:
            run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
```

```

run_metadata = tf.RunMetadata()
summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True),
                     options=run_options, run_metadata=run_metadata)
train_writer.add_run_metadata(run_metadata, 'step%03d' % i)
train_writer.add_summary(summary, i)
saver.save(sess, log_dir+"/model.ckpt", i)
print('Adding run metadata for', i)
else:
    summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
    train_writer.add_summary(summary, i)
train_writer.close()
test_writer.close()

```

之后切换到 Linux 命令行下，执行 TensorBoard 程序，并通过--logdir 指定 TensorFlow 日志路径，然后 TensorBoard 就可以自动生成所有汇总数据可视化的结果了。

```
tensorboard --logdir=/tmp/tensorflow/mnist/logs/mnist_with_summaries
```

执行上面的命令后，出现一条提示信息，复制其中的网址到浏览器，就可以看到数据可视化的图表了。

```

Starting TensorBoard b'39' on port 6006
(You can navigate to http://192.168.233.101:6006)

```

首先打开标量 SCALARS 的窗口，并单击打开 accuracy 的图表，如图 9-2 所示。其中可以看到两条曲线，分别是 train 和 test 中 accuracy 随训练步数变化的趋势。我们可以调整 Smoothing 参数，控制对曲线的平滑处理，数值越小越接近实际值，但波动较大；数值越大则曲线越平缓。单击图表左下方的按钮，可以放大这个图片，单击它右边的按钮则可以调整坐标轴的范围，以便更清楚地展示。

切换到图像 IMAGES 窗口，如图 9-3 所示，可以看到 MNIST 数据集中的图片。不只是原始数据，所有在 `tf.summary.image()` 中汇总的图片数据都可以在这里看到，包括进行了各种光学畸变后的图片，或是神经网络的中间节点的输出。



图 9-2 TensorBoard SCALARS 变量展示效果

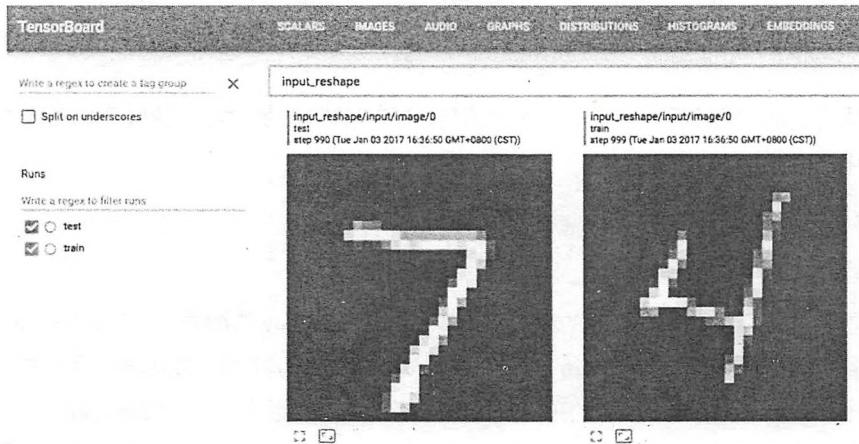


图 9-3 TensorBoard IMAGES 图片展示效果

进入计算图 GRAPHS 窗口,可以看到整个 TensorFlow 计算图的结构,如图 9-4 所示。这里展示了网络 forward 的 inference 的流程,以及 backward 训练更新参数的流程。我们在代码中创建的只有 forward 正向过程: input → layer1 → dropout → layer2 → cross_entropy、accuracy 的,而训练中 backward 的求解梯度、更新参数等操作是 TensorFlow 帮我们自动创建的。图中实线代表数据上的依赖关系,虚线代表控制条件上的依赖关系。单击某个节点的窗口,可以查看它的属性、输入及输出,并且可以看到输出 tensor 的尺寸。

我们也可以单击节点右上角的“+”号按钮，展开这个 node 的内部细节。例如，单击 layer2 可以看到内部的 weights、biases、矩阵乘法操作、向量加法操作，以及激活函数计算的操作，这些操作都归属于 `tf.name_scope('layer2')` 这个命名空间（name scope）。所有在一个命名空间中的节点都会被折叠在一起，在设计网络时，我们要尽可能精细地使用命名空间对节点名称进行规范，这样会展示出更清晰的结构。同时，在 TensorBoard 中，我们可以右键单击一个节点并选择删除它，这不会真的在计算图中中删除它，但是可以简化我们的视图，以便更好地观察网络结构。我们也可以切换配色风格，一种是基于结构的，相同的结构的节点有一样的颜色；另一种是基于运算硬件的，在同一个运算硬件上的节点有一样的颜色。同时，我们可以单击左边面板的 Session runs，选择我们之前记录过 run_metadata 的训练元信息，这样可以查看某轮迭代计算的时间消耗、内存占用等情况。

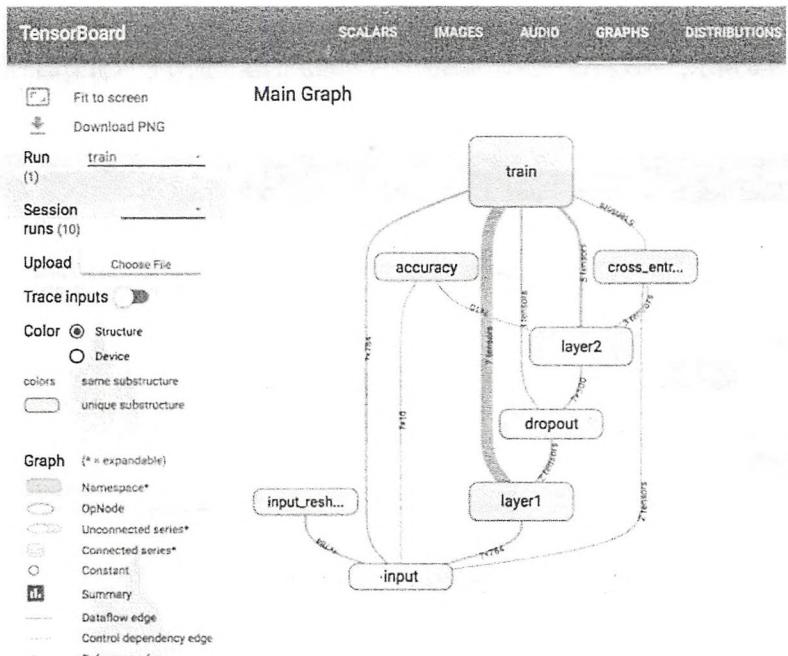


图 9-4 TensorBoard GRAPHS 计算图展示效果

切换到 DISTRIBUTIONS 窗口，如图 9-5 所示，可以查看之前记录的各个神经网络层输出的分布，包括在激活函数前的结果及在激活函数后的结果。这样能观察到神经网络节点的输出是否有效，会不会存在过多的被屏蔽的节点（dead neurons）。

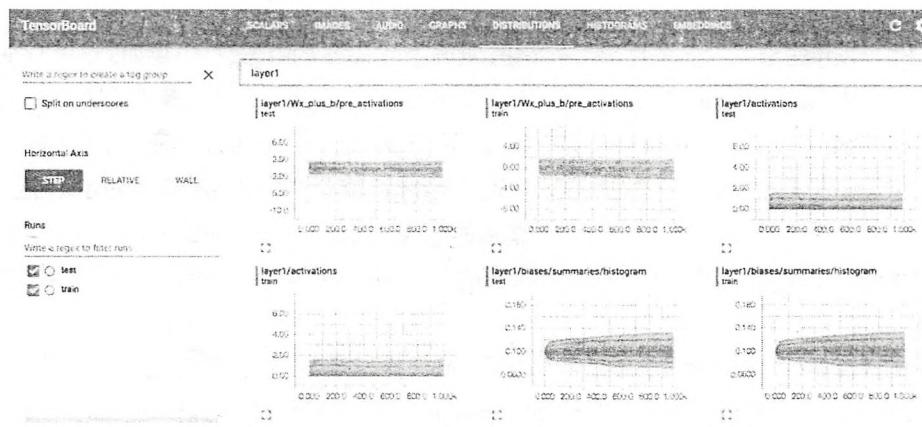


图 9-5 TensorBoard DISTRIBUTIONS 变量分布展示效果

也可以将 DISTRIBUTIONS 的图示结构转为直方图的形式。单击 HISTOGRAMS 窗口，如图 9-6 所示，可以将每一步训练后的神经网络层的输出的分布以直方图的形式展示出来。



图 9-6 TensorBoard HISTOGRAMS 直方图的展示效果

单击 EMBEDDINGS 窗口，如图 9-7 所示，可以看到降维后的嵌入向量的可视化效果，这是 TensorBoard 中的 Embedding Projector 功能。虽然在 MNIST 数据的训练中是没有嵌入向量的，但是只要我们使用 `tf.train.Saver` 保存了整个模型，就可以让 TensorBoard 自动

对模型中所有二维的 Variable 进行可视化 (TensorFlow 中只有 Variable 可以被保存, 而 Tensor 不可以, 因此我们需要把想可视化的 Tensor 转为 Variable)。我们可以选择 T-SNE 或者 PCA 等算法对数据的列 (特征) 进行降维, 并在 3D 或者 2D 的坐标中进行可视化展示。如果我们的模型是 Word2Vec 计算或 Language Model, 那么 TensorBoard 的 EMBEDDINGS 可视化功能会变得非常有用。



图 9-7 TensorBoard EMBEDDINGS 向量嵌入展示效果

9.2 多 GPU 并行

TensorFlow 中的并行主要分为模型并行和数据并行。模型并行需要根据不同模型设计不同的并行方式, 其主要原理是将模型中不同计算节点放在不同硬件资源上运算。比较通用的且能简便地实现大规模并行的方式是数据并行, 其思路我们在第 1 章讲解过, 是同时使用多个硬件资源来计算不同 batch 的数据的梯度, 然后汇总梯度进行全局的参数更新。

数据并行几乎适用于所有深度学习模型, 我们总是可以利用多块 GPU 同时训练多个 batch 数据, 运行在每块 GPU 上的模型都基于同一个神经网络, 网络结构完全一样, 并且共享模型参数。本节我们主要讲解同步的数据并行, 即等待所有 GPU 都计算完一个 batch 数据的梯度后, 再统一将多个梯度合在一起, 并更新共享的模型参数, 这种方法类似于使用了一个较大的 batch。使用数据并行时, GPU 的型号、速度最好一致, 这样效率最高。

而异步的数据并行，则不等待所有 GPU 都完成一次训练，而是哪个 GPU 完成了训练，就立即将梯度更新到共享的模型参数中。通常来说，同步的数据并行比异步的模式收敛速度更快，模型的精度更高。

下面就讲解使用多 GPU 的同步数据并行来训练卷积神经网络的例子，使用的数据集为 CIFAR-10。首先载入各种依赖的库，其中包括 TensorFlow Models 中 cifar10 的类（我们在第 5 章下载了这个库，现在只要确保 Python 执行路径在 models/tutorials/image/cifar10 下即可），它可以下载 CIFAR-10 数据并进行一些数据预处理。本节我们不再重头设计一个 CNN，而是直接使用一个现成的 CNN，并侧重于讲解如何使用数据并行训练这个 CNN。本节代码主要来自 TensorFlow 的开源实现⁶⁸。

```
import os.path
import re
import time
import numpy as np
import tensorflow as tf
import cifar10
```

我们设置 batch 大小为 128，最大步数为 100 万步（中间可以随时停止，模型定期保存），使用的 GPU 数量为 4（取决于当前机器上有多少可用显卡）。

```
batch_size=128
max_steps=1000000
num_gpus=4
```

然后定义计算损失的函数 tower_loss。我们先使用 cifar10.distorted_inputs 产生数据增强后的 images 和 labels，并调用 cifar10.inference 生成卷积网络（注意，我们需要为每个 GPU 生成单独的网络，这些网络的结构完全一致，并且共享模型参数）。通过 cifar10.inference 生成的卷积网络和 5.3 节中的卷积网络一致，读者若想了解网络结构的具体细节，可参考 5.3 节中的内容。然后，根据卷积网络和 labels，调用 cifar10.loss 计算损失函数（这里不直接返回 loss，而是储存到 collection 中），并用 tf.get_collection('losses', scope) 获取当前这个 GPU 上的 loss（通过 scope 限定了范围），再使用 tf.add_n 将所有损失叠加到一起得到 total_loss。最后返回 total_loss 作为函数结果。

```
def tower_loss(scope):
    images, labels = cifar10.distorted_inputs()
```

```

logits = cifar10.inference(images)
_ = cifar10.loss(logits, labels)
losses = tf.get_collection('losses', scope)
total_loss = tf.add_n(losses, name='total_loss')
return total_loss

```

下面定义函数 `average_gradients`，它负责将不同 GPU 计算出的梯度进行合成。函数的输入参数 `tower_grads` 是梯度的双层列表，外层列表是不同 GPU 计算得到的梯度，内层列表是某个 GPU 内计算的不同 `Variable` 对应的梯度，最内层元素为(`grads, variable`)，即 `tower_grads` 的基本元素为二元组(梯度，变量)。其具体形式为`[[grad0_gpu0, var0_gpu0), (grad1_gpu0, var1_gpu0), ...], [(grad0_gpu1, var0_gpu1), (grad1_gpu1, var1_gpu1), ...], ...]`。我们先创建平均梯度的列表 `average_grads`，它负责将梯度在不同 GPU 间进行平均。然后使用 `zip(*tower_grads)` 将这个双层列表转置，变成`[(grad0_gpu0, var0_gpu0), (grad0_gpu1, var0_gpu1), ...], [(grad1_gpu0, var1_gpu0), (grad1_gpu1, var1_gpu1), ...], ...]`的形式，然后使用循环遍历其元素。每个循环中获取的元素 `grad_and_vars`，是同一个 `Variable` 的梯度在不同 GPU 上的计算结果，即`[(grad0_gpu0, var0_gpu0), (grad0_gpu1, var0_gpu1), ...]`。对同一个 `Variable` 的梯度在不同 GPU 计算出的副本，需要计算其梯度的均值，如果这个梯度是一个 N 维的向量，需要在每个维度上都进行平均。我们先使用 `tf.expand_dims` 给这些梯度添加一个冗余的维度 0，然后把这些梯度放到列表 `grad` 中，接着使用 `tf.concat` 将它们在维度 0 上合并，最后使用 `tf.reduce_mean` 针对维度 0 上求平均，即将其他维度全部平均。最后将平均后的梯度跟 `Variable` 组合得到原有的二元组(梯度，变量)格式，并添加到列表 `average_grads` 中。当所有梯度都求完均值后，我们返回 `average_grads`。

```

def average_gradients(tower_grads):
    average_grads = []
    for grad_and_vars in zip(*tower_grads):
        grads = []
        for g, _ in grad_and_vars:
            expanded_g = tf.expand_dims(g, 0)
            grads.append(expanded_g)

        grad = tf.concat(grads, 0)
        grad = tf.reduce_mean(grad, 0)
        average_grads.append((grad, grad_and_vars[0].var))
    return average_grads

```

```
v = grad_and_vars[0][1]
grad_and_var = (grad, v)
average_grads.append(grad_and_var)

return average_grads
```

下面定义训练的函数。先设置默认的计算设备为 CPU，用来进行一些简单的计算。然后使用 global_step 记录全局训练的步数，并计算一个 epoch 对应的 batch 数，以及学习速率衰减需要的步数 decay_steps。我们使用 tf.train.exponential_decay 创建随训练步数衰减的学习速率，这里第 1 个参数为初始学习速率，第 2 个参数为全局训练的步数，第 3 个参数为每次衰减需要的步数，第 4 个参数为衰减率，staircase 设为 True 代表是阶梯式的衰减。然后设置优化算法为 GradientDescent，并传入随步数衰减的学习速率。

```
def train():
    with tf.Graph().as_default(), tf.device('/cpu:0'):
        global_step = tf.get_variable('global_step', [],
                                      initializer=tf.constant_initializer(0),
                                      trainable=False)

        num_batches_per_epoch = cifar10.NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN / \
                               batch_size
        decay_steps = int(num_batches_per_epoch * cifar10.NUM_EPOCHS_PER_DECAY)

        lr = tf.train.exponential_decay(cifar10.INITIAL_LEARNING_RATE,
                                       global_step,
                                       decay_steps,
                                       cifar10.LEARNING_RATE_DECAY_FACTOR,
                                       staircase=True)

    opt = tf.train.GradientDescentOptimizer(lr)
```

我们定义储存各 GPU 计算结果的列表 tower_grads。然后创建一个循环，循环次数为 GPU 数量，在每一个循环内，使用 tf.device 限定使用第几个 GPU，如 gpu0、gpu1，然后使用 tf.name_scope 将命名空间定义为 tower_0、tower_1 的形式。对每一个 GPU，使用前

面定义好的函数 `tower_loss` 获取其损失，然后调用 `tf.get_variable_scope().reuse_variables()` 重用参数，让所有 GPU 共用一个模型及完全相同的参数。再使用 `opt.compute_gradients(loss)` 计算单个 GPU 的梯度，并将求得的梯度添加到梯度列表 `tower_grads`。最后使用前面写好的函数 `average_gradients` 计算平均梯度，并使用 `opt.apply_gradients` 更新模型参数。这样就完成了多 GPU 的同步训练和参数更新。

```

tower_grads = []
for i in range(num_gpus):
    with tf.device('/gpu:%d' % i):
        with tf.name_scope('%s_%d' % (cifar10.TOWER_NAME, i)) as scope:
            loss = tower_loss(scope)
            tf.get_variable_scope().reuse_variables()
            grads = opt.compute_gradients(loss)
            tower_grads.append(grads)

grads = average_gradients(tower_grads)
apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)

```

我们创建模型的保存器 `saver`，将 `Session` 的 `allow_soft_placement` 参数设置为 `True`（有些操作只能在 CPU 进行，不使用 `soft_placement` 可能导致运行出错），初始化全部参数，并调用 `tf.train.start_queue_runners()` 准备好大量的数据增强后的训练样本，防止后面的训练被阻塞在生成样本上。

```

saver = tf.train.Saver(tf.all_variables())
init = tf.global_variables_initializer()
sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True))
sess.run(init)
tf.train.start_queue_runners(sess=sess)

```

下面进入训练的循环，最大迭代次数为 `max_steps`。在每一步中执行一次更新梯度的操作 `apply_gradient_op`（即一次训练操作）和计算损失的操作 `loss`，同时使用 `time.time()` 记录耗时。每隔 10 步，展示一次当前 `batch` 的 `loss`，以及每秒钟可训练的样本数和每个 `batch` 训练所需要花费的时间。每隔 1000 步，使用 `Saver` 保存整个模型文件。

```
for step in range(max_steps):
```

```

start_time = time.time()
_, loss_value = sess.run([apply_gradient_op, loss])
duration = time.time() - start_time

if step % 10 == 0:
    num_examples_per_step = batch_size * num_gpus
    examples_per_sec = num_examples_per_step / duration
    sec_per_batch = duration / num_gpus

    format_str = ('step %d, loss = %.2f (%.1f examples/sec; %.3f '
                  'sec/batch)')
    print (format_str % (step, loss_value, examples_per_sec,
                         sec_per_batch))

if step % 1000 == 0 or (step + 1) == max_steps:
    saver.save(sess, '/tmp/cifar10_train/model.ckpt', global_step=step)

```

我们将主函数后全部定义完后，使用 `cifar10.maybe_download_and_extract()` 下载完整的 CIFAR-10 数据，并调用 `train()` 函数开始训练。

```

cifar10.maybe_download_and_extract()
train()

```

下面展示的结果即为训练过程中显示的日志，`loss` 从最开始的 4 点几，到第 70 万步时，大致降到了 0.07。我们的训练速度很快，平均每个 `batch` 的耗时仅为 0.021s，平均每秒可以训练 6000 个样本，差不多正好是单 GPU 的 4 倍。因此在单机多 GPU 的情况下，使用 TensorFlow 实现的数据并行效率是非常高的。

```

step 729470, loss = 0.07 (6043.4 examples/sec; 0.021 sec/batch)
step 729480, loss = 0.07 (6200.1 examples/sec; 0.021 sec/batch)
step 729490, loss = 0.08 (6055.5 examples/sec; 0.021 sec/batch)
step 729500, loss = 0.09 (5986.7 examples/sec; 0.021 sec/batch)
step 729510, loss = 0.07 (6075.3 examples/sec; 0.021 sec/batch)
step 729520, loss = 0.06 (6630.1 examples/sec; 0.019 sec/batch)
step 729530, loss = 0.09 (6788.4 examples/sec; 0.019 sec/batch)

```

```
step 729540, loss = 0.08 (6464.4 examples/sec; 0.020 sec/batch)
step 729550, loss = 0.06 (6548.5 examples/sec; 0.020 sec/batch)
step 729560, loss = 0.08 (6900.3 examples/sec; 0.019 sec/batch)
step 729570, loss = 0.08 (6381.3 examples/sec; 0.020 sec/batch)
step 729580, loss = 0.07 (6101.0 examples/sec; 0.021 sec/batch)
```

9.3 分布式并行

TensorFlow 的分布式并行基于 gRPC 通信框架,其中包括一个 master 负责创建 Session,还有多个 worker 负责执行计算图中的任务。我们需要先创建一个 TensorFlow Cluster 对象,它包含了一组 task (每个 task 一般是一台单独的机器) 用来分布式地执行 TensorFlow 的计算图。一个 Cluster 可以切分为多个 job,一个 job 是指一类特定的任务,比如 parameter server (ps)、worker,每一个 job 里可以包含多个 task。我们需要为每一个 task 创建一个 server,然后连接到 Cluster 上,通常每个 task 会执行在不同的机器上,当然也可以一台机器上执行多个 task (控制不同的 GPU)。Cluster 对象通过 `tf.train.ClusterSpec` 来初始化,初始化信息是一个 Python 的 dict,例如 `tf.train.ClusterSpec({"ps": ["192.168.233.201:2222"], "worker":["192.168.233.202:2222","192.168.233.203:2222"]})`,这代表设置了一个 parameter server 和两个 worker,分别在三台不同机器上。对每个 task,我们需要给它定义自己的身份,比如对这个 ps 我们将设置 `server = tf.train.Server(cluster, job_name="ps", task_index=0)`,将这台机器的 job 定义为 ps,并且是 ps 中的第 0 台机器。此外,通过在程序中使用诸如 `with tf.device("/job:worker/task:7")`,可以限定 Variable 存放在哪个 task 或哪台机器上。

TensorFlow 的分布式有几种模式,比如 In-graph replication 模型并行,将模型的计算图的不同部分放在不同机器上执行;而 Between-graph replication 则是数据并行,每台机器使用完全相同的计算图,但是计算不同的 batch 数据。此外,我们还有异步并行和同步并行,异步并行指每机器独立计算梯度,一旦计算完就更新到 parameter server 中,不等其他机器;同步并行指所有机器都完成对梯度的计算后,将多个梯度合成并统一更新模型参数。一般来说,同步并行训练时,loss 下降的速度更快,可达到的最大精度更高,但是同步并行有木桶效应,速度取决于最慢的那个机器,所以当设备速度一致时,效率比较高。

下面我们就用 TensorFlow 实现包含 1 个 parameter server 和 2 个 worker 的分布式并行训练程序，并以 MNIST 手写数据识别任务作为示例。这里需要写一个完整的 Python 文件，并在不同机器上以不同的 task 执行。首先载入 TensorFlow 和所有依赖库。本节代码主要来自 TensorFlow 的开源实现⁶⁹。

```
import math
import tempfile
import time
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

这里使用 `tf.app.flags` 定义标记，用以在命令行执行 TensorFlow 程序时设置参数。在命令行中指定的参数会被 TensorFlow 读取，并直接转为 flags。设定数据储存目录 `data_dir` 默认为 `/tmp/mnist-data`，隐藏节点数默认为 100，训练最大步数 `train_steps` 默认为 1000000，batch size 默认为 100，学习速率为默认 0.01。

```
flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/mnist-data",
                    "Directory for storing mnist data")
flags.DEFINE_integer("hidden_units", 100,
                     "Number of units in the hidden layer of the NN")
flags.DEFINE_integer("train_steps", 1000000,
                     "Number of (global) training steps to perform")
flags.DEFINE_integer("batch_size", 100, "Training batch size")
flags.DEFINE_float("learning_rate", 0.01, "Learning rate")
```

然后设定是否使用同步并行的标记 `sync_replicas` 默认为 `False`，在命令行执行时可以设为 `True` 开启同步并行。同时，设定需要累计多少个梯度来更新模型的值默认为 `None`，这个参数代表进行同步并行时，一共积攒多少个 batch 的梯度才进行一次参数更新，设为 `None` 则使用 worker 的数量，即所有 worker 都完成一个 batch 的训练后再更新模型参数。

```
flags.DEFINE_boolean("sync_replicas", False,
                     "Use the sync_replicas (synchronized replicas) mode, "
                     "wherein the parameter updates from workers are "
                     "aggregated before applied to avoid stale gradients")
```

```
flags.DEFINE_integer("replicas_to_aggregate", None,
                    "Number of replicas to aggregate before parameter "
                    "update is applied (For sync_replicas mode only; "
                    "default: num_workers)")
```

再定义 ps 的地址，这里默认为 192.168.233.201:2222，读者应该根据集群的实际情况配置，下同。将 worker 的地址设置为 192.168.233.202:2222 和 192.168.233.203:2222。同时，设置 job_name 和 task_index 的 FLAG，这样在命令行执行时，可以输入这两个参数。

```
flags.DEFINE_string("ps_hosts", "192.168.233.201:2222",
                   "Comma-separated list of hostname:port pairs")
flags.DEFINE_string("worker_hosts",
                   "192.168.233.202:2222,192.168.233.203:2222",
                   "Comma-separated list of hostname:port pairs")
flags.DEFINE_string("job_name", None, "job name: worker or ps")
flags.DEFINE_integer("task_index", None,
                     "Worker task index, should be >= 0. task_index=0 is "
                     "the master worker task the performs the variable "
                     "initialization ")
```

将 flags.FLAGS 直接命名为 FLAGS，简化使用。同时，设置图片尺寸 IMAGE_PIXELS 为 28。

```
FLAGS = flags.FLAGS
IMAGE_PIXELS = 28
```

接下来编写程序的主函数 main，首先使用 input_data.read_data_sets 下载并读取 MNIST 数据集，并设置为 one_hot 编码格式。同时，检测命令行输入的参数，确保有 job_name 和 task_index 这两个必备的参数。显示出 job_name 和 task_index，并将 ps 和 worker 的所有地址解析成列表 ps_spec 和 worker_spec。

```
def main(unused_argv):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

    if FLAGS.job_name is None or FLAGS.job_name == "":
        raise ValueError("Must specify an explicit `job_name`")
```

```

if FLAGS.task_index is None or FLAGS.task_index == "":
    raise ValueError("Must specify an explicit `task_index`")

print("job name = %s" % FLAGS.job_name)
print("task index = %d" % FLAGS.task_index)

ps_spec = FLAGS.ps_hosts.split(",")
worker_spec = FLAGS.worker_hosts.split(",")

```

先计算总共的 worker 数量，然后使用 `tf.train.ClusterSpec` 生成一个 TensorFlow Cluster 的对象，传入的参数是 ps 的地址信息和 worker 的地址信息。再使用 `tf.train.Server` 创建当前机器的 server，用以连接到 Cluster。如果当前节点是 parameter server，则不再进行后续的操作，而是使用 `server.join` 等待 worker 工作。

```

num_workers = len(worker_spec)
cluster = tf.train.ClusterSpec({"ps": ps_spec, "worker": worker_spec})
server = tf.train.Server(
    cluster, job_name=FLAGS.job_name, task_index=FLAGS.task_index)
if FLAGS.job_name == "ps":
    server.join()

```

这里判断当前机器是否为主节点，即 `task_index` 是否为 0。然后定义当前机器的 `worker_device`，格式为"job:worker/task:0/cpu:0"。我们假定有两台机器，并且每台机器有 1 块 GPU，则总共需要两个 worker。如果一台机器有多块 GPU，可以通过一个 task 管理多个 GPU 或者使用多个 task 分别管理。下面使用 `tf.train.replica_device_setter()` 设置 worker 的资源，其中 `worker_device` 为计算资源，`ps_device` 为存储模型参数的资源。我们通过 `replica_device_setter` 将模型参数部署在独立的 ps 服务器 "/job:ps/cpu:0"，并将训练操作部署在"/job:worker/task:0/cpu:0"，即本机的 GPU。最后再创建记录全局训练步数的变量 `global_step`。

```

is_chief = (FLAGS.task_index == 0)
worker_device = "/job:worker/task:%d/cpu:0" % FLAGS.task_index
with tf.device(
    tf.train.replica_device_setter(
        worker_device=worker_device,

```

```

ps_device="/job:ps/cpu:0",
cluster=cluster)):

global_step = tf.Variable(0, name="global_step", trainable=False)

```

接下来，定义神经网络模型，本节的神经网络和 4.4 节的 MLP 全连接网络基本一致。下面使用 `tf.truncated_normal` 初始化权重，使用 `tf.zeros` 初始化偏置，创建输入的 `placeholder`，并使用 `tf.nn.xw_plus_b` 对输入 `x` 进行矩阵乘法和加偏置操作，再用 `ReLU` 激活函数处理，得到第一个隐层的输出 `hid`。然后使用 `tf.nn.xw_plus_b` 和 `tf.nn.softmax` 对第一层的输出 `hid` 进行处理，得到网络的最终输出 `y`。最后计算损失 `cross_entropy`，并定义优化器为 Adam。

```

hid_w = tf.Variable(
    tf.truncated_normal([IMAGE_PIXELS*IMAGE_PIXELS, FLAGS.hidden_units],
                        stddev=1.0 / IMAGE_PIXELS), name="hid_w")

hid_b = tf.Variable(tf.zeros([FLAGS.hidden_units]), name="hid_b")

sm_w = tf.Variable(
    tf.truncated_normal([FLAGS.hidden_units, 10],
                        stddev=1.0 / math.sqrt(FLAGS.hidden_units)), name="sm_w")

sm_b = tf.Variable(tf.zeros([10]), name="sm_b")

x = tf.placeholder(tf.float32, [None, IMAGE_PIXELS * IMAGE_PIXELS])
y_ = tf.placeholder(tf.float32, [None, 10])

hid_lin = tf.nn.xw_plus_b(x, hid_w, hid_b)
hid = tf.nn.relu(hid_lin)

y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b))
cross_entropy = -tf.reduce_sum(y_* tf.log(tf.clip_by_value(y, 1e-10,
                                                          1.0)))

opt = tf.train.AdamOptimizer(FLAGS.learning_rate)

```

我们判断是否设置了同步训练模式 `sync_replicas`，如果是同步模式，则先获取同步更新模型参数所需要的副本数 `replicas_to_aggregate`；如果没有单独设置，则使用 `worker` 数

作为默认值。然后使用 `tf.train.SyncReplicasOptimizer` 创建同步训练的优化器，它实质上是对原有优化器的一个扩展，我们传入原有优化器及其他参数（`replicas_to_aggregate`、`total_num_replicas`、`replica_id` 等），它就会将原有优化器改造为同步的分布式训练版本。最后，使用普通的（即异步的）或同步的优化器对损失 `cross_entropy` 进行优化。

```
if FLAGS.sync_replicas:
    if FLAGS.replicas_to_aggregate is None:
        replicas_to_aggregate = num_workers
    else:
        replicas_to_aggregate = FLAGS.replicas_to_aggregate

    opt = tf.train.SyncReplicasOptimizer(
        opt,
        replicas_to_aggregate=replicas_to_aggregate,
        total_num_replicas=num_workers,
        replica_id=FLAGS.task_index,
        name="mnist_sync_replicas")

train_step = opt.minimize(cross_entropy, global_step=global_step)
```

如果是同步训练模式，并且为主节点，则使用 `opt.get_chief_queue_runner` 创建队列执行器，并使用 `opt.get_init_tokens_op` 创建全局参数初始化器。

```
if FLAGS.sync_replicas and is_chief:
    chief_queue_runner = opt.get_chief_queue_runner()
    init_tokens_op = opt.get_init_tokens_op()
```

下面生成本地的参数初始化操作 `init_op`，创建临时的训练目录，并使用 `tf.train.Supervisor` 创建分布式训练的监督器，传入的参数包括 `is_chief`、`train_dir`、`init_op` 等。这个 `Supervisor` 会管理我们的 task 参与到分布式训练。

```
init_op = tf.global_variables_initializer()
train_dir = tempfile.mkdtemp()
sv = tf.train.Supervisor(is_chief=is_chief,
                        logdir=train_dir,
                        init_op=init_op,
```

```
recovery_wait_secs=1,
global_step=global_step)
```

然后设置 Session 的参数，其中 allow_soft_placement 设为 True 代表当某个操作在指定的 device 不能执行时，可以转到其他 device 执行。

```
sess_config = tf.ConfigProto(
    allow_soft_placement=True,
    log_device_placement=False,
    device_filters=["/job:ps",
                    "/job:worker/task:%d" % FLAGS.task_index])
```

如果为主节点，则显示初始化 Session，其他节点则显示等待主节点的初始化操作。然后执行 sv.prepare_or_wait_for_session()，若为主节点则会创建 Session，若为分支节点则会等待。

```
if is_chief:
    print("Worker %d: Initializing session..." % FLAGS.task_index)
else:
    print("Worker %d: Waiting for session to be initialized..." %
          FLAGS.task_index)

sess = sv.prepare_or_wait_for_session(server.target, config=sess_config)

print("Worker %d: Session initialization complete." % FLAGS.task_index)
```

接着，如果处于同步模式并且是主节点，则调用 sv.start_queue_runners 执行队列化执行器 chief_queue_runner，并执行全局的参数初始化器 init_tokens_op。

```
if FLAGS.sync_replicas and is_chief:
    print("Starting chief queue runner and running init_tokens_op")
    sv.start_queue_runners(sess, [chief_queue_runner])
    sess.run(init_tokens_op)
```

下面就正式到了训练过程。我们记录 worker 执行训练的启动时间，初始化本地训练的步数 local_step，然后进入训练循环。在每一步训练中，我们从 mnist.train.next_batch 读取一个 batch 的数据，并生成 feed_dict，再调用 train_step 执行一次训练。当全局训练步

数达到我们预设的最大值后，停止训练。

```
time_begin = time.time()
print("Training begins @ %f" % time_begin)

local_step = 0
while True:
    batch_xs, batch_ys = mnist.train.next_batch(FLAGS.batch_size)
    train_feed = {x: batch_xs, y_: batch_ys}

    _, step = sess.run([train_step, global_step], feed_dict=train_feed)
    local_step += 1

    now = time.time()
    print("%f: Worker %d: training step %d done (global step: %d)" %
          (now, FLAGS.task_index, local_step, step))

    if step >= FLAGS.train_steps:
        break
```

训练结束后，我们展示总训练时间，并在验证数据上计算预测结果的损失 cross_entropy，并展示出来。至此，我们的主函数 main 全部结束。

```
time_end = time.time()
print("Training ends @ %f" % time_end)
training_time = time_end - time_begin
print("Training elapsed time: %f s" % training_time)

val_feed = {x: mnist.validation.images, y_: mnist.validation.labels}
val_xent = sess.run(cross_entropy, feed_dict=val_feed)
print("After %d training step(s), validation cross entropy = %g" %
      (FLAGS.train_steps, val_xent))
```

这是代码的最后一部分，在主程序中执行 tf.app.run() 并启动 main() 函数，我们将全部代码保存为文件 distributed.py。我们需要在 3 台不同的机器上分别执行 distributed.py 启动

3 个 task，在每次执行 distributed.py 时我们需要传入 job_name 和 task_index 指定 worker 的身份。

```
if __name__ == "__main__":
    tf.app.run()
```

我们分别在三台机器 192.168.233.201、192.168.233.202 和 192.168.233.203 上执行下面三行代码。第一台机器执行第一行代码，第二台机器执行第二行代码，下同。这样我们就在三台机器上分别启动了一个 parameter server 及两个 worker。

```
python distributed.py --job_name=ps --task_index=0
python distributed.py --job_name=worker --task_index=0
python distributed.py --job_name=worker --task_index=1
```

如果我们想使用同步模式，只需要将上面的代码加上--sync_replicas=True，就可以自动开启同步训练。注意，此时 global_step 和异步不同，异步时，全局步数是所有 worker 训练步数之和，同步时则是指有多少轮并行训练。

```
python distributed.py --job_name=ps --task_index=0 --sync_replicas=True
python distributed.py --job_name=worker --task_index=0 --sync_replicas=True
python distributed.py --job_name=worker --task_index=1 --sync_replicas=True
```

下面是我们在 parameter server 上显示出的日志。我们在 192.168.233.201:2222 上顺利开启了 PS 的服务。

```
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:197] Initialize GrpcChannelCache for job ps -> {0 -> localhost:2222}
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:197] Initialize GrpcChannelCache for job worker -> {0 -> 192.168.233.202:2223, 1 -> 192.168.233.203:2224}
I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:206] Started server with target: grpc://localhost:2222
```

下面是 worker0 在 192.168.233.202 上的训练日志。

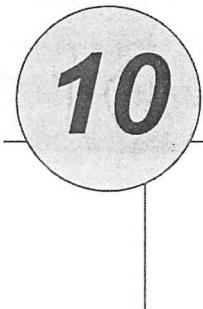
```
1484195706.167773: Worker 0: training step 5657 done (global step: 10285)
1484195706.178822: Worker 0: training step 5658 done (global step: 10287)
1484195706.189648: Worker 0: training step 5659 done (global step: 10289)
```

```
1484195706.200894: Worker 0: training step 5660 done (global step: 10291)
1484195706.212560: Worker 0: training step 5661 done (global step: 10293)
1484195706.224736: Worker 0: training step 5662 done (global step: 10295)
1484195706.237565: Worker 0: training step 5663 done (global step: 10297)
1484195706.252718: Worker 0: training step 5664 done (global step: 10299)
```

下面是 worker1 在 192.168.233.203 上的训练日志。

```
1484195714.332566: Worker 1: training step 5269 done (global step: 11569)
1484195714.345961: Worker 1: training step 5270 done (global step: 11571)
1484195714.359124: Worker 1: training step 5271 done (global step: 11573)
1484195714.372848: Worker 1: training step 5272 done (global step: 11575)
1484195714.386048: Worker 1: training step 5273 done (global step: 11577)
1484195714.398567: Worker 1: training step 5274 done (global step: 11579)
1484195714.411631: Worker 1: training step 5275 done (global step: 11581)
1484195714.424619: Worker 1: training step 5276 done (global step: 11583)
```

至此，我们在三台机器上的数据并行模式的分布式训练的示例就结束了，读者可以看到用 TensorFlow 实现分布式训练非常简单。我们可以复用单机版本的网络结构，只是在不同机器上训练不同 batch 的数据，并使用 parameter server 统一管理模型参数。另外，分布式 TensorFlow 的运行效率也非常高，在 16 台机器上可以获得 15 倍于单机的速度，非常适合大规模神经网络的训练。



TF.Learn 从入门到精通

TF.Learn 是 TensorFlow 中的一个很重要的模块，它包括各种类型的深度学习及流行的机器学习算法。这个模块是从之前比较热门的 TensorFlow 官方 Scikit Flow 项目迁移过来的，发起者是谷歌的员工 Illia Polosukhin 及本书作者之一唐源。代码的风格采用数据科学界比较热门的 Scikit-learn 风格，旨在帮助数据科学从业者更好、更快地适应和接受 TensorFlow 的代码。它囊括了许多 TensorFlow 的代码和设计模式，从而使用户能够更快地开始搭建自己的机器学习模型来实现不同的应用。同时，用户也能极大地避免代码重复，更好地把精力放在搭建更精确的模型上。自从 TensorFlow v0.9 版本发布之后，TF.Learn 能够无缝地和其他 contrib 模块结合起来使用，比如 contrib.losses、contrib.layer、contrib.metrics，等等（我们会在第 11 章系统地介绍 contrib 模块）。第 10 章和第 11 章将使用 TensorFlow 0.11.0-rc0 版本作为示例讲解，其他版本的代码可能会出现不兼容的现象。

10.1 分布式 Estimator

本节我们介绍 Estimator 的分布式特性、自定义模型的用法、Estimator 的架构，并介绍怎样建立自己的分布式机器学习 Estimator。

10.1.1 分布式 Estimator 自定义模型介绍

Estimator 包括各种各样的机器学习和深度学习的类，用户能直接使用这些高阶类，同时可以根据实际的应用需求快速创建自己的子类。有了 `graph_actions` 模块的帮助，Estimator 很大一部分在训练和评估模型时需要用到的复杂的分布式逻辑都被实现和浓缩，使用者就不再需要把精力放在很复杂的 Supervisor 和 Coordinator 分布式训练具体实现细节和逻辑上面。

Estimator 接受自定义模型，目前它接受以下几组不同的函数签名。

- (1) `(features, targets) -> (predictions, loss, train_op)`
- (2) `(features, targets, mode) -> (predictions, loss, train_op)`
- (3) `(features, targets, mode, params) -> (predictions, loss, train_op)`

我们以第一组函数签名举例说明，以下是一个简单的自定义的模型。

```
import tensorflow as tf
from tensorflow.contrib import layers
from tensorflow.contrib import learn

def my_model(features, target):
    target = tf.one_hot(target, 3, 1, 0)
    features = layers.stack(features, layers.fully_connected, [10, 20, 10])
    prediction, loss =
        tf.contrib.learn.models.logistic_regression_zero_init(features,
                                                               target)
    train_op = tf.contrib.layers.optimize_loss(
        loss, tf.contrib.framework.get_global_step(), optimizer='Adagrad',
        learning_rate=0.1)
    return {'class': tf.argmax(prediction, 1), 'prob': prediction}, loss,
           train_op
```

这个自定义模型接受两个参数：`features` 和 `targets`。`features` 是数据的特征，`targets` 是数据特征每一行的目标或者分类的标识，利用 `tf.one_hot` 对 `targets` 进行独热编码 (One-hot Encoding)，让接下来损失函数的计算更方便。接下来，用 `layers.stack` 叠加多层