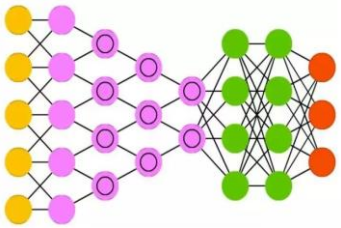
智能计算系统

第四章 编程框架使用

中国科学院计算技术研究所
陈云霁 研究员
cyj@ict.ac.cn

Driving Example


输入 ▶ 建模 ▶ 实现 ▶ 运行 ▶ 输出


编程框架


Bang

第四章将学习到实现深度学习算法所使用的编程框架的简单用法

提纲

- 深度学习编程框架的概念
- TensorFlow概述
- TensorFlow编程模型及基本用法
- 基于TensorFlow的训练及预测实现

为什么需要编程框架?

深度学习算法得到广泛关注，越来越多的公司、程序员需要使用深度学习算法

为什么需要编程框架?

计算误差对权重 w_{ij} 的偏导数是两次使用链式法则得到的：

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net_j}}\frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

在右边的最后一项中，只有加权和 $\mathrm{net_j}$ 取决于 w_{ij}，因此

$$\frac{\partial \mathrm{net_j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj}o_k\right) = o_i.$$

神经元 j 的输出对其输入的导数就是激活函数的偏导数（这里假定使用逻辑函数）：

$$\frac{\partial o_j}{\partial \mathrm{net_j}} = \frac{\partial}{\partial \mathrm{net_j}}\varphi(\mathrm{net_j}) = \varphi(\mathrm{net_j})(1-\varphi(\mathrm{net_j}))$$

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from matplotlib import style

def SGD(samples, y, step_size=2, max_iter_count=1000):
  m, var = samples.shape
  theta = np.zeros(2)
  y = y.flatten()
  #进入循环内
  loss = 1
  iter_count = 0
  iter_list=[]
  loss_list=[]
  theta1=[]
```

有必要将算法中的常用操作封装成组件提供给程序员，以提高深度学习算法开发效率

但如果 j 是网络中任一内层，求 E 关于 o_j 的导数就不太简单了。

考虑 E 为接受来自神经元 j 的输入的所有神经元 $L = u, v, \ldots, w$ 的输入的函数，

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(\mathrm{net}_u, \mathrm{net}_v, \ldots, \mathrm{net}_w)}{\partial o_j}$$

并关于 o_j 取全微分，可以得到该导数的一个递归表达式：

$$\frac{\partial E}{\partial o_j} = \sum_{l\in L}\left(\frac{\partial E}{\partial \mathrm{net}_l}\frac{\partial \mathrm{net}_l}{\partial o_j}\right) = \sum_{l\in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial \mathrm{net}_l}\frac{\partial \mathrm{net}_l}{\partial o_j}\right) = \sum_{l\in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial \mathrm{net}_l}w_{jl}\right)$$

```
    rand1 = np.random.randint(0,m,1)
    h = np.dot(theta,samples[rand1].T)
    #关键点，只需要一个样本点来更新权值
    for i in range(len(theta)):
      theta[i] =theta[i] - step_size*(1/m)*(h - y[rand1])*samples[rand1,i]
    #计算总体的损失精度，等于各个样本损失精度之和
    for i in range(m):
      h = np.dot(theta.T, samples[i])
      #每组样本点损失的精度
      every_loss = (1/(var*m))*np.power((h - y[i]), 2)
      loss = loss + every_loss

    print("iter_count: ", iter_count, "the loss:", loss)
```

算法理论复杂

代码实现工作量大

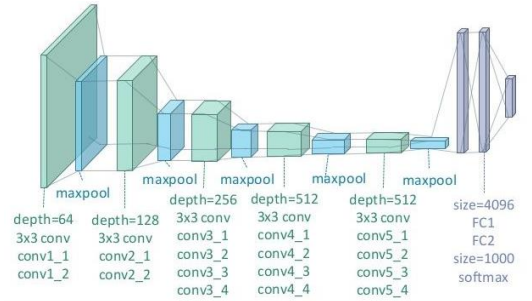
为什么需要编程框架?

- 深度学习算法具有多层结构，每层的运算由一些基本操作构成
- 这些基本操作中存在大量共性运算，如卷积、池化、激活等。将这些共性运算操作封装起来，可以提高编程实现效率
- 面向这些封装起来的操作，硬件程序员可以基于硬件特征，有针对性的进行充分优化，使其能充分发挥硬件的效率

定义

- 随着深度学习研究的深入，深度学习算法变得愈加复杂，研究人员需要花更多的时间和精力在算法的实现上
- 深度学习编程框架：将深度学习算法中的基本操作封装成一系列组件，这一系列深度学习组件，即构成一套深度学习框架
- 编程框架能够帮助算法开发人员更简单的实现已有算法，或设计新的算法。也有助于硬件程序员更有针对性的对关键操作进行优化，使其能充分发挥硬件效率

Driving example-VGGNET19



神经网络

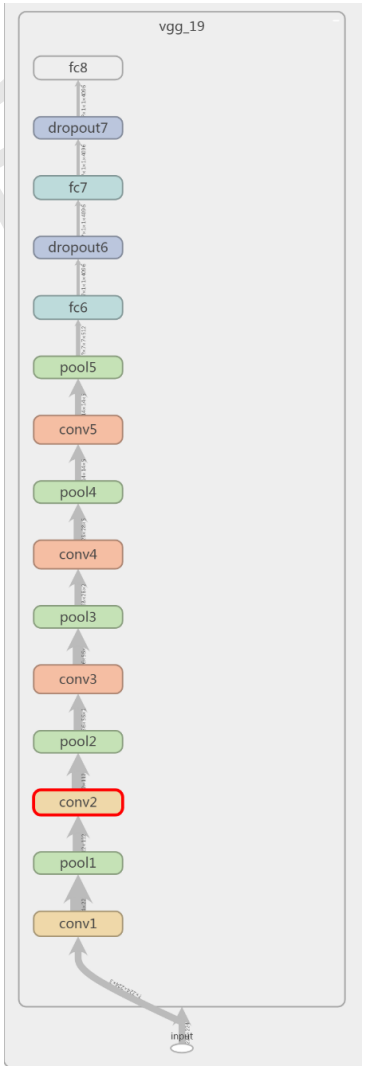
```
def build_vgg19(path):
    net = {}
    vgg_rawnet = scipy.io.loadmat(path)
    vgg_layers = vgg_rawnet['layers'][0]
    net['input'] = tf.Variable(np.zeros((1, IMAGE_HEIGHT, IMAGE_WIDTH, 3)).astype('float32'))
    net['conv1_1'] = build_net('conv', net['input'], get_weight_bias(vgg_layers, 0))
    net['conv1_2'] = build_net('conv', net['conv1_1'], get_weight_bias(vgg_layers, 2))
    net['pool1'] = build_net('pool', net['conv1_2'])
    net['conv2_1'] = build_net('conv', net['pool1'], get_weight_bias(vgg_layers, 5))
    net['conv2_2'] = build_net('conv', net['conv2_1'], get_weight_bias(vgg_layers, 7))
    net['pool2'] = build_net('pool', net['conv2_2'])
    net['conv3_1'] = build_net('conv', net['pool2'], get_weight_bias(vgg_layers, 10))
    net['conv3_2'] = build_net('conv', net['conv3_1'], get_weight_bias(vgg_layers, 12))
    net['conv3_3'] = build_net('conv', net['conv3_2'], get_weight_bias(vgg_layers, 14))
    net['conv3_4'] = build_net('conv', net['conv3_3'], get_weight_bias(vgg_layers, 16))
    net['pool3'] = build_net('pool', net['conv3_4'])
    net['conv4_1'] = build_net('conv', net['pool3'], get_weight_bias(vgg_layers, 19))
    net['conv4_2'] = build_net('conv', net['conv4_1'], get_weight_bias(vgg_layers, 21))
    net['conv4_3'] = build_net('conv', net['conv4_2'], get_weight_bias(vgg_layers, 23))
    net['conv4_4'] = build_net('conv', net['conv4_3'], get_weight_bias(vgg_layers, 25))
    net['pool4'] = build_net('pool', net['conv4_4'])
    net['conv5_1'] = build_net('conv', net['pool4'], get_weight_bias(vgg_layers, 28))
    net['conv5_2'] = build_net('conv', net['conv5_1'], get_weight_bias(vgg_layers, 30))
    net['conv5_3'] = build_net('conv', net['conv5_2'], get_weight_bias(vgg_layers, 32))
    net['conv5_4'] = build_net('conv', net['conv5_3'], get_weight_bias(vgg_layers, 34))
    net['pool5'] = build_net('pool', net['conv5_4'])
    return net
```

TensorFlow实现

在计算设备运行

```
__mlu_entry__ void Proposal(...) {
  ...
  __nram__ half scores[...];
  __nramset_half(scores, ...);
  ...
  __bang_maxpool(...);
  ...
}
```

Bang实现算子功能，
嵌入编程框架中



生成计算图

提纲

- 深度学习编程框架的概念
- TensorFlow概述
- TensorFlow编程模型及基本用法
- 基于TensorFlow的训练及预测实现

主流深度学习编程框架

	支持语言
TensorFlow	Python/C/C++/Java/Go
Caffe2	Python/C++
Pytorch	Python
Cognitive Toolkit	Python/C++/C#
MXNet	Python/C++/R/Scala/Julia
Torch7	Lua
Theano	Python
DL4J	Java/Scala
Lasagne	Python

TensorFlow是目前使用人数最多、影响最大的编程框架

TensorFlow概述

- TensorFlow---AlphaGo背后的力量

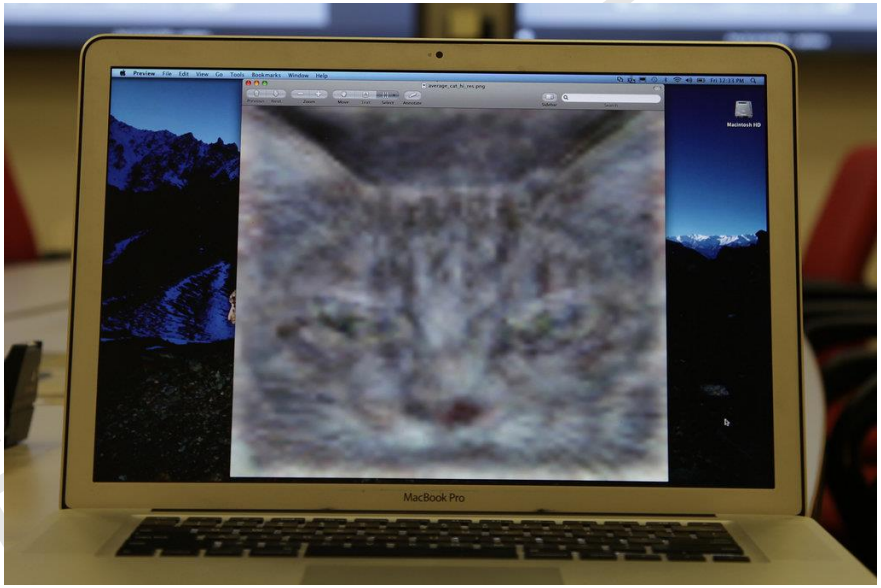
- 谷歌团队于2015年开发并开源，Github上活跃度最高的深度学习编程框架
- TensorFlow及其变种可以工作于各种类型的异构系统，包括手机、平板电脑等移动设备，数百台机器和数千种计算设备（如GPU）的大规模分布式系统

谷歌第一代分布式深度学习平台--- DistBelief

- 谷歌大脑项目的早期成果，2011年开发完成，是第一代可扩展的分布式深度学习平台，可用于深度神经网络的训练和预测。
- 谷歌团队使用DistBelief进行了各种各样的研究，包括无监督学习、图像分类、视频分类、语音识别、序列预测、行人检测、强化学习等。
- 该平台将Google语音识别率提高了25%，在Google Photos 中建立了图片搜索，并驱动了 Google 的图片字幕匹配实验

谷歌第一代分布式深度学习平台--- DistBelief

- 谷歌基于DistBelief建立了大规模的神经网络，开展了著名的“猫脸识别”实验
- 在YouTube视频中提取出的1000万张图像中，在没有外界干涉的条件下，通过自我学习，识别出了猫脸
- 仅可用于神经网络，且和Google内部基础产品联系紧密，在实际配置和应用中存在局限性

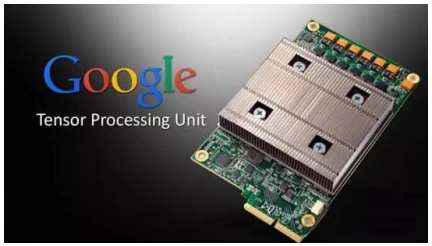


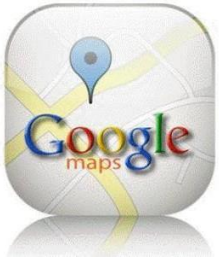
谷歌第二代大规模机器学习系统---TensorFlow

- 作为DistBelief的继任者，由谷歌团队开发并于2015年11月开源的深度学习框架，用于实施和部署大规模机器学习模型
- 相比DistBelief，具有以下优点：
  - 支持多种高级语言作为输入：Python、C、C++、Java、Go
  - 更灵活的编程模型
  - 更高的性能
  - 支持在更广泛的异构硬件平台上进行训练和使用更大规模的神经网络模型

TensorFlow的应用

- Google 多个产品（Gmail，Google Play Recommendation，Search，Translate，Map等）
- AlphaGo
- TPU2.0/3.0
- IBM PowerAI
- Intel Movidius Myriad
- 寒武纪MLU100/270





安装与技术资料获取

- www.tensorflow.org

  TensorFlow官网

- https://github.com/tensorflow/tensorflow

  官方Github仓库

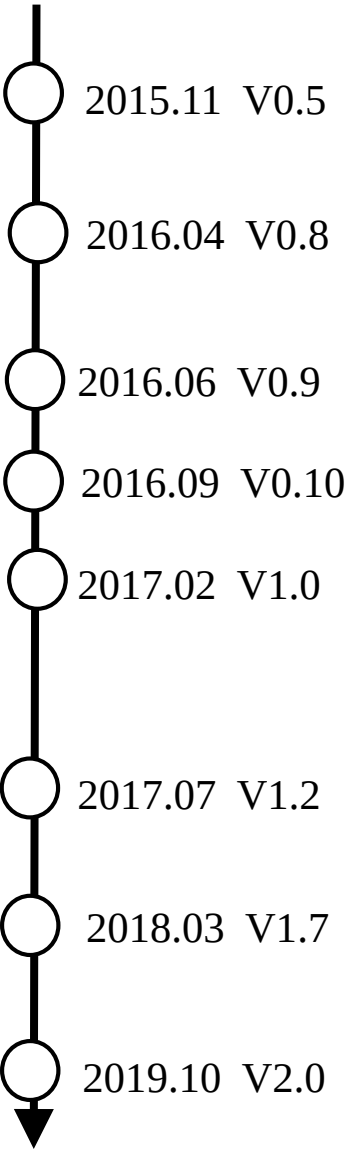
- http://web.stanford.edu/class/cs20si/

  斯坦福大学TensorFlow系列课程

- https://developers.google.cn/machine-learning/crash-course/

  谷歌官方视频教程

版本历史



时间	版本号	更新内容
2015.11	0.5	第一个发布版本
2016.04	0.8	支持分布式计算，支持大规模智能应用的部署。使用16块GPU性能可达单GPU的15倍，50块GPU时性能可达单GPU的40倍
2016.06	0.9	增加了对多平台的支持，包括iOS、Raspberry Pi等
2016.09	0.10	提供深度学习高级抽象库TensorFlow-Slim，方便用户快速定义和训练模型，有效扩大了用户群体
2017.02	1.0	第一个正式版。增加了多个提升性能及易用性的更新，包括线性代数编译器XLA、调试工具TensorFlow Debugger、对Android的支持等。
2017.07	1.2	增加了面向InfiniBand等高性能网络的RDMA通信方案，解决了分布式训练时通信开销过大的问题
2018.03	1.7	推出TensorBoard可视化工具，有效辅助用户理解算法模型及工作流程
2020.01	2.1	目前的最新版本

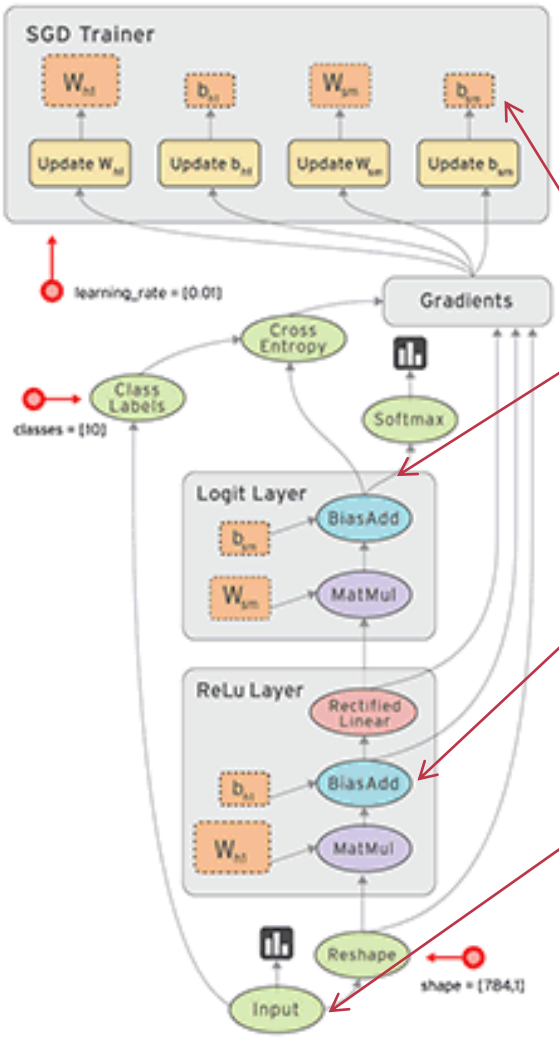
提纲

- 深度学习编程框架的概念
- TensorFlow概述
- TensorFlow编程模型及基本用法
- 基于TensorFlow的训练及预测实现

TensorFlow编程模型及基本概念

- 命令式编程与声明式编程
  - 命令式编程：关注程序执行的具体步骤，计算机按照代码中的顺序一步一步执行具体的运算。整体优化困难。
  - 示例：交互式UI程序、操作系统
  - 声明式编程：告诉计算机想要达到的目标，不指定具体的实现步骤，而是通过函数、推论规则等描述数据之间的关系，优化较容易。
  - 示例：人工智能、深度学习

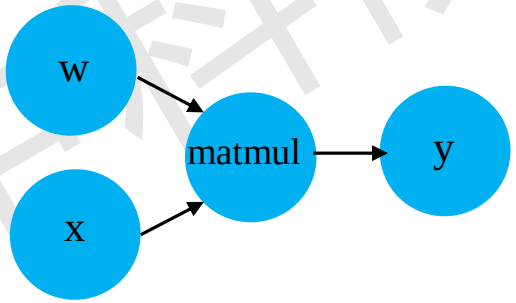
TensorFlow中的几个基本概念



- 1、使用计算图来表示机器学习算法中所有的计算和状态
- 2、将所有的数据建模成张量（tensor）
- 3、具体计算操作运行在会话（session）环境中
- 4、将多种类型的计算定义为操作（operation）
- 5、通过变量（variable）存储计算图中的有状态参数，如模型参数
- 6、通过占位符（placeholder）将张量传递到会话中
- 7、通过队列（Queue）处理数据读取和计算图的异步执行

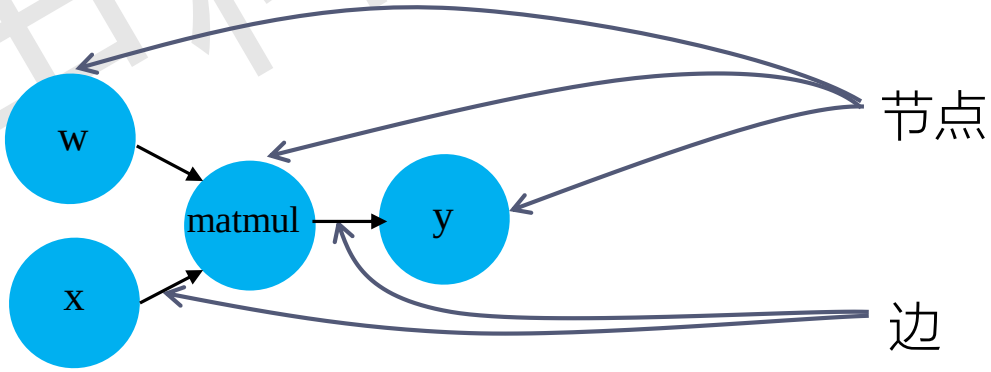
1、计算图

- TensorFlow中使用有向图来描述计算过程。有向图中包含一组节点和边
- 支持通过多种高级语言来构建计算图（C++/Python）
- 计算图对应了神经网络的结构
- 示例：y=w*x



节点和边

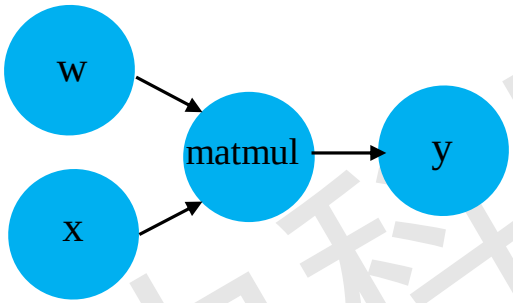
- 节点一般用来表示各类操作，包括数学运算、变量读写、数据填充等，也可以表示输入数据、模型参数、输出数据
- 边表示"节点"之间的输入输出关系。分为两类：
  - 一类是传递具体数据的边。传递的数据即为张量（tensor）。
  - 一类是表示节点之间控制依赖关系的边。这类边不传递数据，只表示节点执行的顺序：必须前序节点计算完成，后序节点才开始计算。



TensorFlow程序简单示例

- TensorFlow程序一般分为两部分：构建计算图（Line3-6）、执行计算图（Line8-10）

构造计算图



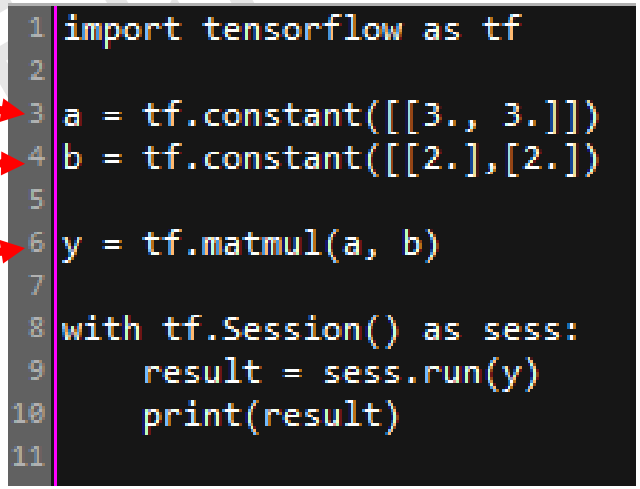

```
1  import tensorflow as tf
2
3  x = tf.constant([[3., 3.]])
4  w = tf.constant([[2.],[2.]])
5
6  y = tf.matmul(x, w)
7
8  with tf.Session() as sess:
9    result = sess.run(y)
10   print(result)
```

创建会话，执行运算

- TensorFlow 1.x
  - 静态图，方便对整个计算图做全局优化，性能较高；但调试困难，影响开发效率
- TensorFlow 2.x
  - 动态图，调试简单，更适合快速开发；但运行效率低于静态图方法

2、操作

- 计算图中的每个计算节点即代表一个操作（operation），其接收0个或多个tensor作为输入，产生0个或多个tensor作为输出
- 操作对应了神经网络中的具体计算
- 右图中包含了三个操作：
  1）给a赋常数值
  2）给b赋常数值
  3）a、b相乘得到y



```
1  import tensorflow as tf
2
3  a = tf.constant([[3., 3.]])
4  b = tf.constant([[2.],[2.]])
5
6  y = tf.matmul(a, b)
7
8  with tf.Session() as sess:
9      result = sess.run(y)
10     print(result)
11
```

操作的主要属性

属性名	功能说明
tf.operation.name	操作的名称
tf.operation.type	操作的类型，如add
tf.operation.inputs	操作的输入
tf.operation.outputs	操作的输出
tf.operation.control_inputs	该操作的控制依赖列表
tf.operation.device	执行该操作所使用的设备
tf.operation.graph	操作所属的计算图
tf.operation.traceback	实例化该操作时的调用栈

TensorFlow中的常用操作

操作类型	常用算子
标量运算	add、subtract、multiply、div、greater、less、equal、abs、sign、square、pow、log、sin、cos
矩阵运算	matmul、matrix_inverse、matrix_determinant、matrix_transpose
逻辑操作	logical_and、is_finite
神经网络运算	convolution、max_pool、bias_add、softmax、dropout、sigmoid、relu
储存、恢复	save、restore
初始化操作	zeros_initializer、random_normal_initializer、orthogonal_initializer
随机运算	random_gamma、multinomial、random_normal、random_shuffle

3、张量（tensor）

- TensorFlow中，张量是计算图上的数据载体，用张量统一表示所有的数据，张量在计算图的节点之间传递
- 张量中并没有实际保存数据，而仅是对计算结果的引用，对应了神经网络中各个节点之间流动的数据
- 张量可以看做是*n*维的数组，数组的维数即为张量的阶数

阶数	对应数据形式
0	标量
1	向量
2	矩阵
n	*n*维数组

- 例如，一张RGB图片可以表示成三阶张量，多张图片组成的数据集可以表示成四阶张量

tensor的常用属性

属性名	含义
dtype	tensor存储的数据类型
shape	tensor各阶的长度
name	tensor在计算图中的名称
op	计算出此tensor的操作
device	计算出此tensor所用的设备名
graph	包含此tensor的计算图

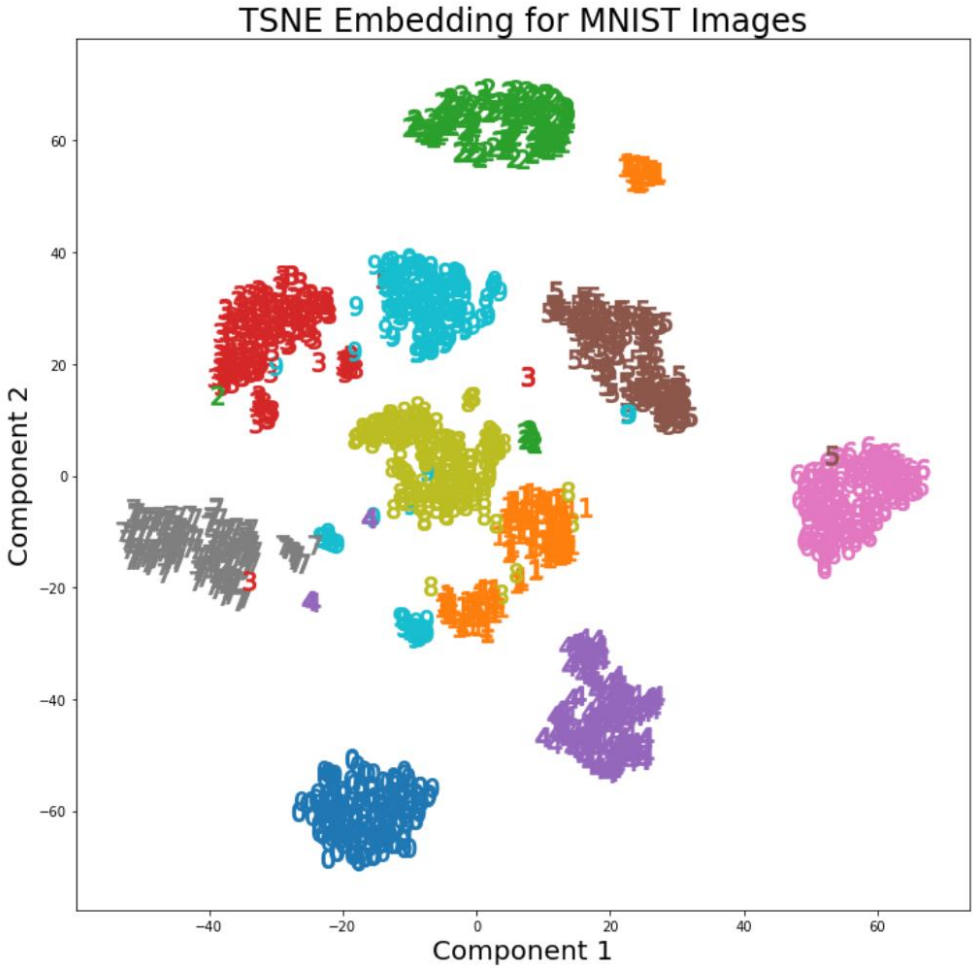
tensor中支持的dtype类型

TensorFlow数据类型	说明
int8/int16/int32/int64	8位/16位/32位/64位有符号整数
float16/float32/float64	半精度/单精度/双精度浮点数
bfloat16	裁短浮点数
uint8/uint16/uint32/uint64	8位/16位无符号整数
bool	布尔值
string	字符串
complex64/complex128	单精度/双精度复数
qint8/qint16/qint32	量化的8位/16位/32位有符号整数
quint8/quint16	量化的8位/16位无符号整数

- 示例： a=tf.constant([2,3],dtype=tf.float32)

深度学习为什么不需要全部float32

- 深度学习算法特性

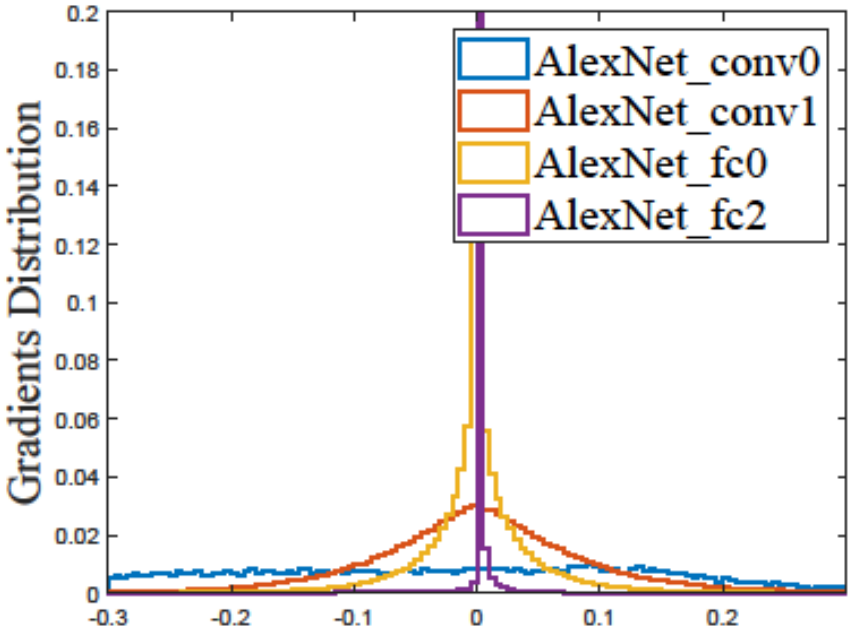


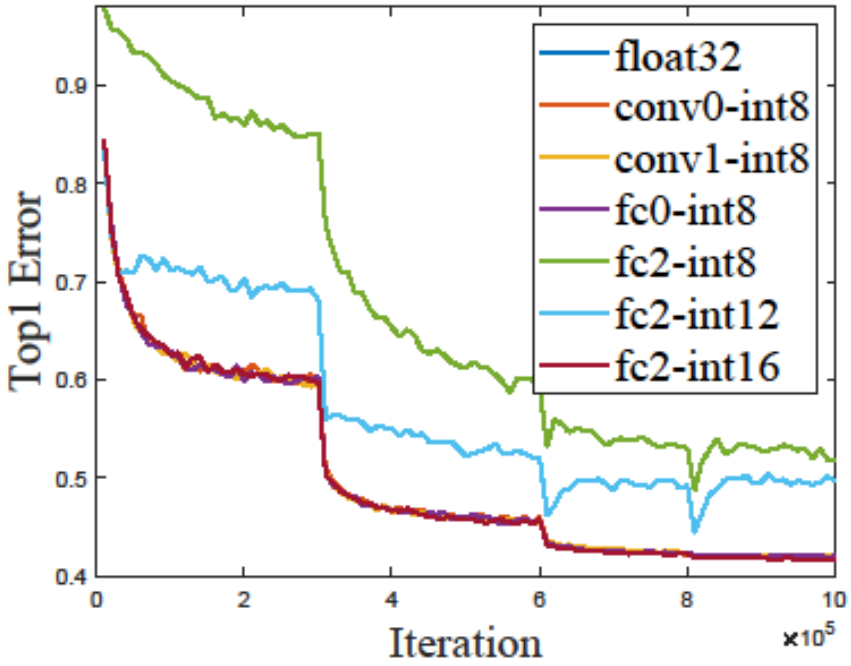
t-SNE可视化深度学习数据

- 这是一个典型的手写体识别任务的网络特征（神经元）二维可视化图
- 不同类别（颜色）的数据间距
- “大间距”意味着 容忍非精确计算

数据位宽与算法精度

- 不同数据的位宽需求是不同的





每层数据都有其保持网络收敛的最低位宽需求，
每层数据的位宽需求与数据分布之间存在关系

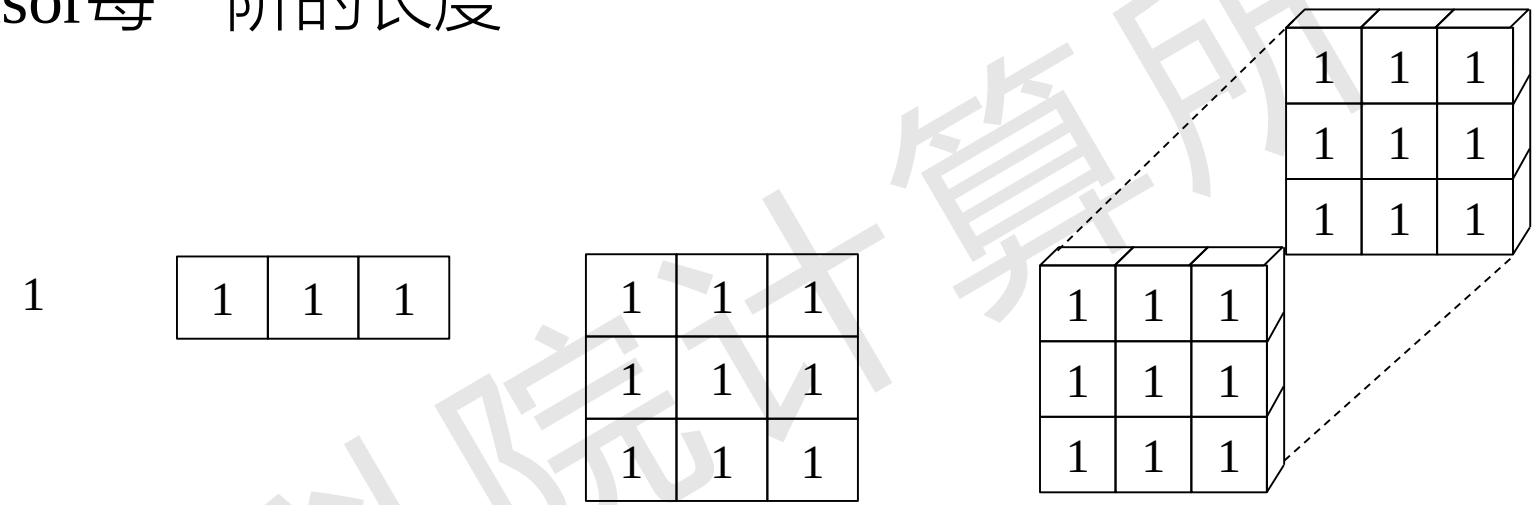
训练时不需要高位宽

- CNN网络，分类、检测、分割任务下的低位宽训练
  - 神经元和权值自适应8bit，梯度8-16bit——精度无损

Classification Network	float32 Acc	Adaptive Acc	Activation int8	Weight int8	Activation Gradient int8	Activation Gradient int16
AlexNet	58.0	58.22	100%	100%	22.5%	77.5%
VGG16	71.0	70.6	100%	100%	31.3%	68.7%
Inception_BN	73.0	72.8	100%	100%	4.5%	95.5%
ResNet50	76.4	76.2	100%	100%	0.8%	99.2%
ResNet152	78.8	78.2	100%	100%	1.7%	98.3%
MobileNet v2	72.0	70.5	100%	100%	0.7%	99.2%
SSD Detection Network	**float32 mAP**	**Adaptive mAP**	**Activation** int8	**Weight** int8	**Activation Gradient** int8	**Activation Gradient** int16
COCO_VGG	43.1	42.4	100%	100%	31.4%	68.6%
VOC_VGG	77.3	77.2	100%	100%	34.3%	65.7%
VOC_ResNet101	73.4	73.1	100%	100%	7.4%	83.6%
IMGDET_ResNet101	44.1	44.4	100%	100%	28.6%	71.4%
Segmentation Network	**float32 meanIoU**	**Adaptive meanIoU**	**Activation** int8	**Weight** int8	**Activation Gradient** int8	**Activation Gradient** int16
deeplab-v1	70.1	69.9	100%	100%	1.0%	99.0%

tensor的shape属性

- 表示tensor每一阶的长度



阶数	0阶	1阶	2阶	3阶
对应数据形式	标量	向量	矩阵	三维数组
shape	()	(3)	(3,3)	(2,3,3)

tensor的device属性

- tf.device(device_name)指定计算出此tensor所用的设备名
- TensorFlow不区分CPU，所有CPU均使用/cpu:0作为设备名称
- 用/gpu:n表示第n个GPU设备，用/mlu:n表示第n个深度学习处理器
- 示例程序：

```
1  import tensorflow as tf
2
3  with tf.device('/cpu:0'):
4    a = tf.constant([[3.,3.]])
5    b = tf.constant([[2.],[2.]])
6
7  with tf.device('/mlu:0'):
8    y = tf.matmul(a,b)
9
10 with tf.Session() as sess:
11   result = sess.run(y)
12   print(result)
```

- Line 3：指定使用CPU来执行Line 4-5
- Line 7：指定使用第0号MLU来执行Line 8

Tensor的常用操作(op)命令

函数名称	功能
tf.shape(tensor)	返回tensor的shape值
tf.to_double(x,name='ToDouble') tf.to_float(x,name='ToFloat') tf.to_int32(x,name='ToInt32') tf.to_int64(x,name='ToInt64')	将x转为64位浮点类型 将x转为32位浮点类型 将x转为32位整型 将x转为64位整型
tf.cast(x,dtype)	将x转换为dtype类型数据
tf.reshape(tensor,shape)	修改tensor各阶的长度为shape 如：假设a=[1,2,3,4]，则tf.reshape(a,[2,2])命令将输出：[[1,2],[3,4]]
tf.slice(input,begin,size)	从由begin指定位置开始的input中提取一个尺寸为size的切片
tf.split(value,num_or_size_splits,axis)	沿着第axis阶对value进行切割，切割成num_or_size_splits份
tf.concat(values,axis)	沿着第axis阶对value进行连接操作

tensor属性简单示例

```
import tensorflow as tf

t0=tf.constant(9,dtype=tf.int32)
#创建一个0阶整型常量

t1=tf.constant([3.,4.1,5.2],dytpe=tf.float32)
#创建一个1阶浮点数数组

t2=tf.constant([['Apple','Pear'],['Potato','Tomato']],dtype=tf.string)
#创建一个2x2的字符串数组

t3=tf.constant([[[2],[2],[1]],[[7],[3],[3]]])
#创建一个2x3x1的3阶整型数组

print(t0)
print(t1)
print(t2)
print(t3)
```

- 输出

```
>>> print(t0)
Tensor("Const:0", shape=(), dtype=int32)
>>> print(t1)
Tensor("Const_1:0", shape=(3,), dtype=float32)
>>> print(t2)
Tensor("Const_2:0", shape=(2,2), dtype=string)
>>> print(t3)
Tensor("Const_3:0", shape=(2,3,1), dtype=int32)
```

tensor属性简单示例

- print(tensor)：打印tensor的属性，而不是打印tensor的值
- 如果需要查看tensor的值，需要运行会话session

```
>>>sess=tf.Session()
>>>print(sess.run(t0))
9
>>>print(sess.run(t1))
[3.        4.0999999       5.19999981]
>>>print(sess.run(t2))
[[b'Apple' b'Pear']
 [b'Potato' b'Tomato']]
>>>print(sess.run(t3))
[[[2]
  [2]
  [1]]

 [[7]
  [3]
  [3]]]
>>>
```

4、会话（session）

- TensorFlow中的计算图描述了计算执行的过程，但并没有真正给输入赋值并执行计算
- 真正的神经网络计算过程需要在TensorFlow程序的session部分中定义并执行
- session为程序提供求解张量、执行操作的运行环境。将计算图转化为不同设备上的执行步骤

```
1  import tensorflow as tf
2
3  with tf.device('/cpu:0'):
4    a = tf.constant([[3., 3.]])
5    b = tf.constant([[2.],[2.])
6
7  with tf.device('/gpu:0'):
8    y = tf.matmul(a, b)
9
10 with tf.Session() as sess:
11     result = sess.run(y)
12     print(result)
13
```

session

- session的典型使用流程：

```
#创建会话
sess = tf.Session()

#执行会话
sess.run()

#关闭会话
sess.close()

```

创建会话（tf.Session）

- Session的输入参数：

参数	功能说明
target	会话连接的执行引擎，默认为进程内引擎
graph	执行计算时加载的计算图。默认值为当前代码中惟一的计算图。当代码中定义了多幅计算图时，使用graph指定待加载的计算图
config	指定相关配置项，如设备数量、并行线程数、GPU配置参数等

- 示例：

```
sess=tf.Session(target=‘ ’,graph=None,config=None)
```

执行会话

- 基于计算图和输入数据，求解张量或执行计算
- 输入参数：

参数	功能说明
fetches	本会话需计算的张量或操作
feed_dict	指定会话执行时需填充的张量或操作，及对应的填充数据
options	设置会话运行时的控制选项
run_metadata	设置会话运行时的非张量信息输出

简单程序示例

```
import tensorflow as tf
a = tf.placeholder(tf.int32)
b = tf.placeholder(tf.int32)
c= tf.multiply(a,b)
with tf.Session() as sess:
    print(sess.run(c,feed_dict = {a:100,b:200}))


x1 = tf.placeholder(tf.float32,[2,3])
x2 = tf.placeholder(tf.float32,[3,2])
x3 = tf.matmul(x1,x2)
with tf.Session() as sess:
    print(sess.run(x3,feed_dict = {x1:[[1,2,3],[4,5,6]],x2:[[1,2],[3,4],[5,6]]}))

```

- 输出

```
20000
[[22.   28.]
 [49.   61.]]

```

关闭会话

- 会话的执行会占用大量硬件资源，因此会话结束时需要关闭会话，以释放这些资源
- 关闭会话的两种方式

1）使用close语句显式关闭会话

```
sess.close()
```

2）使用with语句隐式关闭会话

```
import tensorflow as tf

a = tf.constant([[3., 3.]])
b = tf.constant([[2.],[2.]])

y = tf.matmul(a, b)

with tf.Session() as sess:
    result = sess.run(y)
    print(result)
```

求解张量值的方法

- 方法1：会话中使用run()函数

```
import tensorflow as tf
a = tf.placeholder(tf.int32)
b = tf.placeholder(tf.int32)
c= tf.multiply(a,b)
with tf.Session() as sess:
    print(sess.run(c,feed_dict = {a:100,b:200}))

```

- 方法2：tensor.eval()

```
import tensorflow as tf

a = tf.constant([[1.0,2.0]])
b = tf.constant([[3.0],[4.0]])
c = tf.matmul(a,b)

with tf.Session():
    print(a.eval())
    print(c.eval())
```

session.run与tensor.eval的区别

- tensor.eval()输入参数：

参数名	说明
feed_dict	指定需填充的张量或操作，及对应的填充数据
session	指定求解此tensor或操作的会话

- tensor.eval()函数使用前，均需要显式指定求解张量或操作的会话，如用with语句定义会话
- 处理单个tensor时， tensor.eval()与session.run()等价
- session.run可以一次进行多个tensor或操作的计算

简单程序示例

- tensor.eval()：每次仅能计算一个tensor
- sess.run()：每次可计算多个tensor

```
1  import tensorflow as tf
2
3  a = tf.constant([[1.0,2.0]])
4  b = tf.constant([[3.0],[4.0]])
5  c = tf.constant([[5.0],[6.0]])
6
7  y1 = tf.matmul(a,b)
8  y2 = tf.matmul(a,c)
9
10 with tf.Session() as sess:
11     y1.eval()
12     y2.eval()
13     sess.run([y1,y2])
```

5、变量

- 大多数计算中计算图被执行多次，每次执行后其中的值即被释放
- 变量（variable）是计算图中的一种有状态节点，用于在多次执行同一计算图时存储并更新指定参数，对应了机器学习或深度学习算法中的模型参数
- 作为有状态节点，其输出由输入、节点操作、节点内部已保存的状态值共同作用

变量的常用属性

属性名	含义
dtype	变量的数据类型
shape	变量的长度
name	变量在计算图中的名称
op	变量操作
device	存储此变量所用的设备名
graph	包含此变量的计算图
initialized_value	变量的初值
initializer	为变量赋值的初始化操作
trainable	是否在训练时被优化器更新

创建变量

- 将一个tensor传递给Variable()构造函数，创建时需指定变量的形状与数据类型
- 方法1：使用tf.Variable()函数直接定义

```
a=tf.Variable(2,tf.int16)
b=tf.Variable([1,2])

```

- 方法2：使用TensorFlow内置的函数来定义变量初值，可以是常量或随机值

```
#以标准差0.35的正态分布初始化一个形状为[20，40]的变量
r = tf.Variable(tf.random_normal([20, 40], stddev=0.35))
#初始化一个形状为[2,3]的变量，  里面的元素值全部为0
z = tf.Variable(tf.zeros([2,3]))

```

创建变量

- 方法3：用其他变量的初始值来定义新变量

```
#创建一个随机变量
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")

#创建一个与weights值相同的变量
w2 = tf.Variable(weights.initialized_value(), name="w2")

#创建一个值为weights2倍的变量
w_twice = tf.Variable(weights.initialized_value() * 2, name="w_twice")

```

常用于构建变量的TensorFlow操作

TensorFlow操作	说明
tf.zeros()	产生一个全为0的张量
tf.ones()	产生一个全为1的张量
tf.random_normal()	产生正态分布的随机数
tf.truncated_normal()	从截断的正态分布中输出随机数
tf.random_uniform()	产生满足平均分布的随机数
tf.random_gamma()	产生满足Gamma分布的随机数
tf.fill()	产生一个全为给定值的张量
tf.constant()	产生常量
variable.initialized_value()	产生一个变量的初值

初始化变量

- 创建变量后还需要进行变量初始化
- 最简单的变量初始化方法：使用tf.global_variables_initializer()对所有变量初始化

```
import tensorflow as tf

a = tf.Variable(tf.constant(0.0),dtype=tf.float32)

#在会话中用tf.global_variables_initializer()函数对所有变量初始化
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(a))
```

更新变量

- 变量是计算图中的有状态节点，其输出受到输入、操作、节点内部已保存状态的共同影响
- 变量更新可以通过优化器自动更新完成，也可以通过自定义方法强制赋值更新

更新函数	说明
强制赋值更新	
tf.assign()	更新变量的值
tf. assign_add()	加法赋值
tf. assign_sub()	减法赋值
自动更新	
tf.train.**Optimizer	使用多种优化方法自动更新参数

示例

```
1  f = tf.Variable(1.0)
2  f2 = tf.assign(f,f+2.0)
3  f3 = tf.assign_add(f,3.0)
4  f4 = tf.assign_sub(f,1.5)
5  with tf.Session() as sess:
6    sess.run(tf.global_variables_initializer())
7    print(sess.run(f))
8    print(sess.run(f2))
9    print(sess.run(f3))
10   print(sess.run(f4))
```

6、占位符(placeholder)

- 训练神经网络模型时需要大量的样本输入，如果每个输入都用常量表示，则每个常量都需要在计算图中增加一个节点，最终的计算图会非常大
- 计算图表达的是计算的拓扑结构，在向计算图填充数据前，计算图并没有真正执行运算
- TensorFlow使用占位符来构建计算图中的样本输入节点，而不需要实际分配数据
- 占位符本身没有初始值，只是在程序中分配了内存
- 使用占位符则只会在计算图中增加一个节点，并只在执行时向其填充数据

- tf.placeholder()的输入参数

输入参数	说明
name	在计算图中的名字
dtype	填充数据的数据类型
shape	填充数据的shape值

- 使用时需与feed_dict参数配合，用feed_dict提交数据，进行参数传递

向占位符填充数据

```
import tensorflow as tf
import numpy as np

w1=tf.Variable(tf.random_normal([1,2],stddev=1,seed=1))
#因为需要重复输入x，而每建一个x就会生成一个节点，计算图的效率会低。所以使用占位符
x=tf.placeholder(tf.float32,shape=(1,2))
x1=tf.constant([[0.7,0.9]])
a=x+w1
b=x1+w1

sess=tf.Session()
sess.run(tf.global_variables_initializer())
#运行y时将占位符填上，feed_dict为字典，变量名不可变
y_1=sess.run(a,feed_dict={x:[[0.7,0.9]]})
y_2=sess.run(b)
print(sess.run(w1))
print(y_1)
print(y_2)
sess.close()
```

7、队列(Queue)

- TensorFlow提供了队列（queue）机制，通过多线程将读取数据与计算数据分开
- 队列是一种有状态的操作机制，用来处理数据读取
- 为了加快训练速度，我们可以采用多个线程读取数据，一个线程消耗数据
- 队列操作包含了入队、出队操作
- TensorFlow 提供多种队列机制，如 FIFOQueue 和 RandomShuffleQueue

FIFOQueue

- 先入先出队列，支持入队、出队操作
- 入队操作将输入输出放到FIFOQueue队尾，出队操作将FIFOQueue队列中首个元素取出
- 当队列满时入队操作会被阻塞，当队列为空时出队操作会被阻塞

示例

```
1  import tensorflow as tf
2
3  # 创建一个先进先出队列，指定队列中可以保存三个元素，并指定类型为float
4  q = tf.FIFOQueue(3, 'float')
5  # 使用enqueue_many函数来初始化队列中的元素
6  init = q.enqueue_many(([0., 0., 0.],))
7  # 使用Dequeue函数将队列中的第一个元素出队列。这个元素值，将被存在变量x中
8  x = q.dequeue()
9  y = x + 1
10 # 将加1后的值在重新加入队列中
11 q_inc = q.enqueue([y])
12
13 init.run()
14 q_inc.run()
15 q_inc.run()
16 q_inc.run()
17 q_inc.run()
```

Client

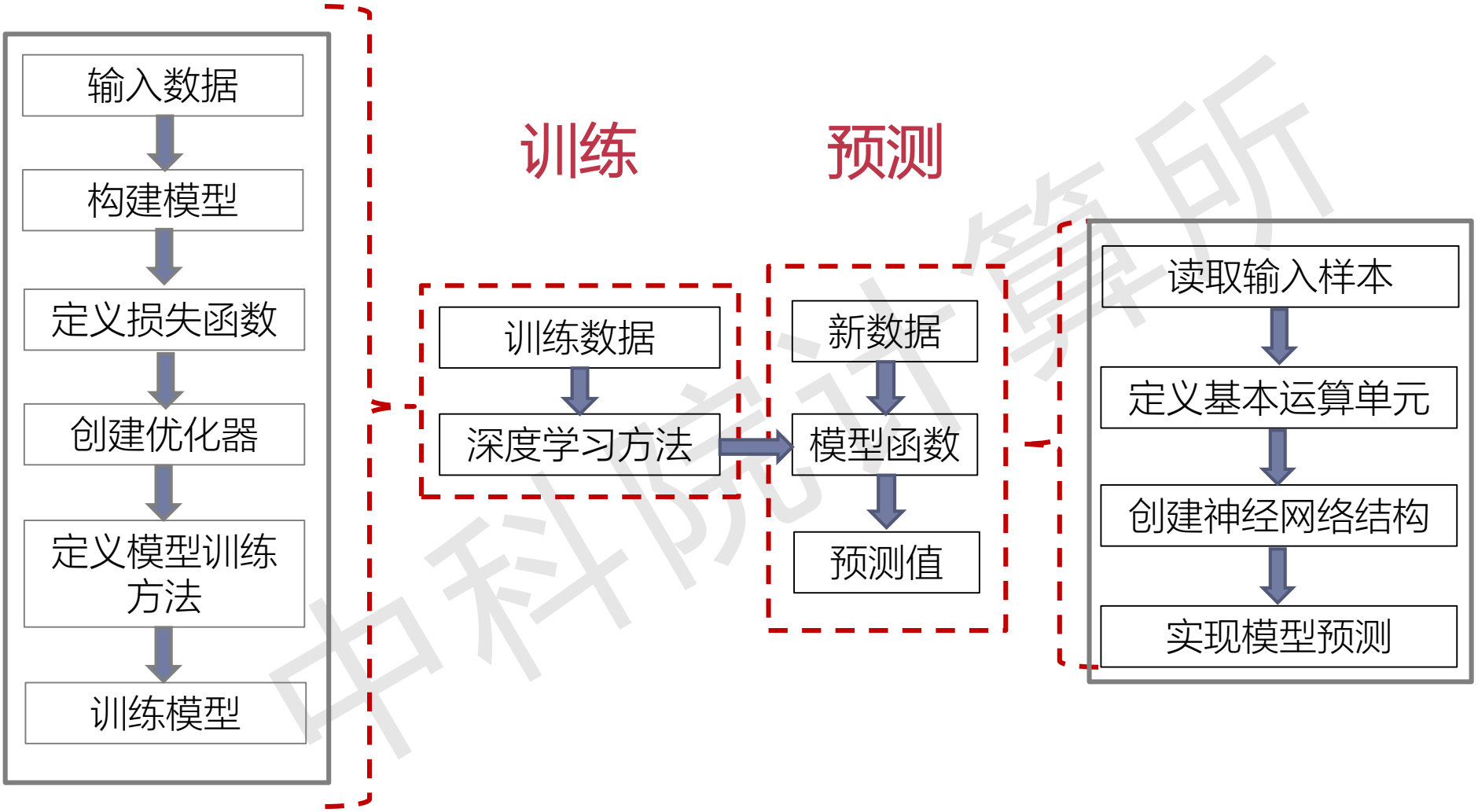
```
q = tf.FIFOQueue(3, "float")
init = q.enqueue_many(([0.,0.,0.],))

x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()
```

总结

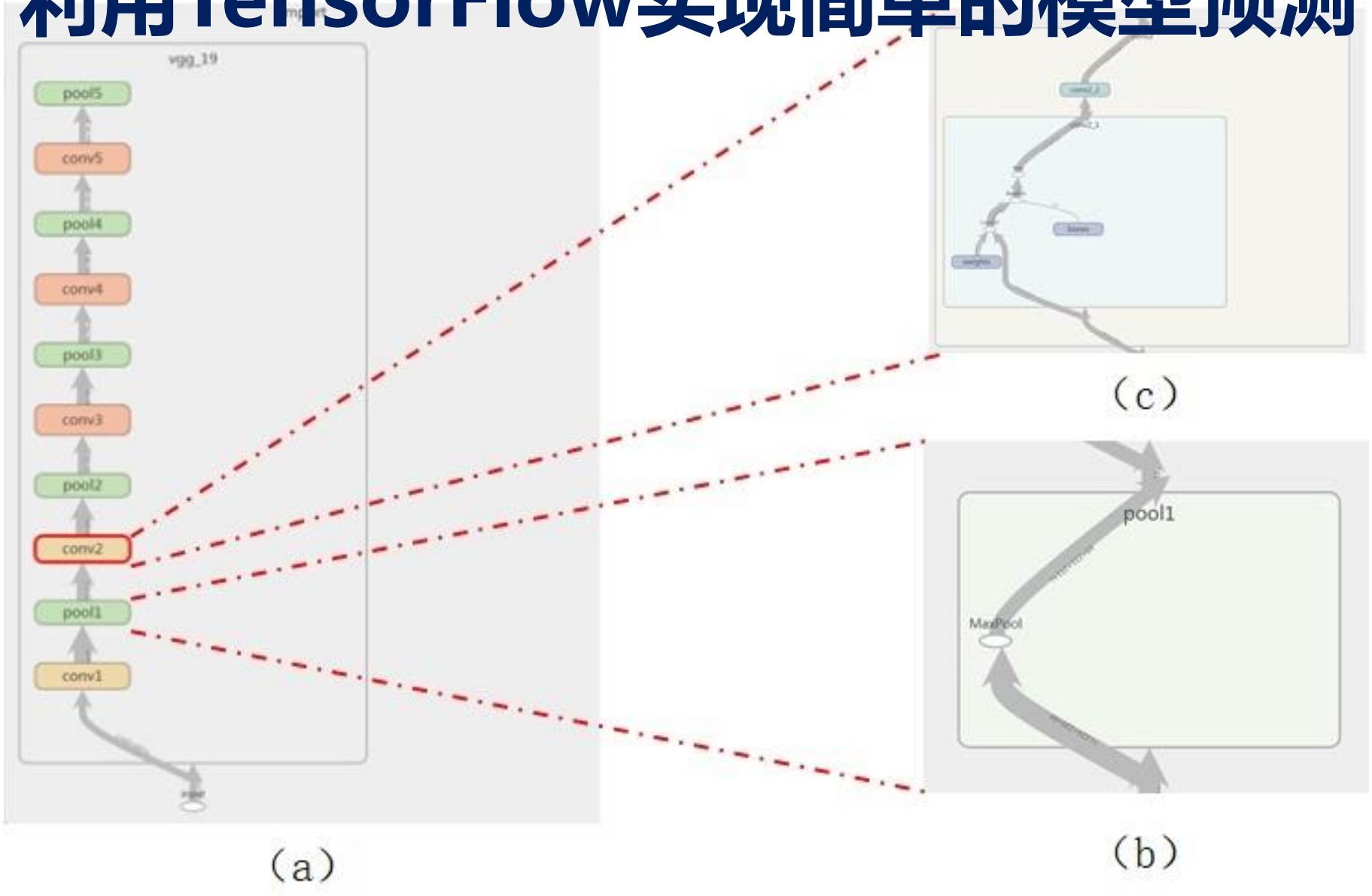
- 计算图：对应神经网络结构
- 操作：对应神经网络具体计算
- 张量：对应神经网络中的数据
- 会话：执行神经网络真正的训练和预测
- 变量：对应神经网络参数
- 占位符：对应神经网络的训练或预测输入
- 队列：对应神经网络训练样本的多线程并行处理

提纲

- 深度学习编程框架的概念
- TensorFlow概述
- TensorFlow编程模型及基本用法
- 基于TensorFlow的训练及预测实现


输入数据
构建模型
定义损失函数
创建优化器
定义模型训练方法
训练模型
训练
预测
训练数据
深度学习方法
新数据
模型函数
预测值
读取输入样本
定义基本运算单元
创建神经网络结构
实现模型预测

利用TensorFlow实现简单的模型预测



1、读取输入样本

```python
import cv2
import numpy as np

def load_image(path):
    img = cv2.imread(path, cv2.IMREAD_COLOR)
    resize_img = cv2.resize(img, (224, 224))
    norm_img = resize_img / 255.0
    return np.reshape(norm_img, (1, 224, 224, 3))
```

2、定义基本运算单元

```
def basic_calc(caltype, nin,inwb=None):
    if caltype=='conv':
        # nin: 本层输入; inwb: inwb[0],inwb[1] == weights,bias
        return tf.nn.relu(tf.nn.conv2d(nin,inwb[0],\
                          strides=[1,1,1,1], padding='SAME')+ inwb[1])
    elif caltype=='pool':
        return tf.nn.max_pool(nin, ksize=[1,2,2,1],\
                              strides=[1,2,2,1], padding='SAME')

def read_wb(vgg19_npy_path,name):
  #从vgg_npy_path路径中读取模型参数到数组data_dict
  data_dict = np.load(vgg19_npy_path,encoding='latin1').item()
  weights = data_dict[name][0]
  weights = tf.constant(weights)
  bias = data_dict[name][1]
  bias = tf.constant(bias)
  return weights,bias

```

使用tf.nn模块定义基本运算单元

tf.nn模块

- TensorFlow中用于深度学习计算的核心模块，提供神经网络相关操作的支持，包括卷积、池化、损失、分类等
- 卷积函数

操作	说明
tf.nn.conv2d(input, filter, strides, padding)	在给定的 input与 filter下计算卷积
tf.nn.depthwise_conv2d(input, filter, strides, padding)	卷积核能相互独立的在自己的通道上面进行卷积操作。
tf.nn.separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding)	在纵深卷积 depthwise filter 之后进行逐点卷积 separable filter
tf.nn,bias_add(value,bias)	对输入加上偏置

激活函数

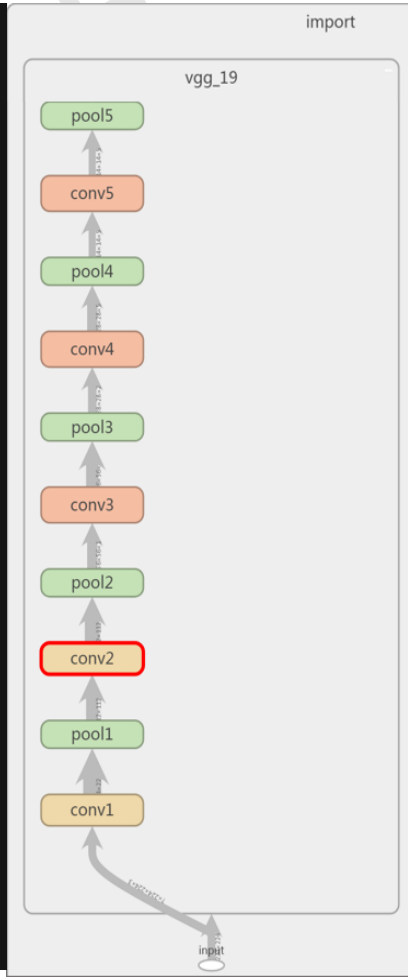
操作	说明
tf.nn.relu(features)	计算relu函数
tf.nn.elu(features)	计算elu函数
tf.nn.dropout(x, keep_prob)	计算dropout，keep_prob为keep概率,
tf.sigmoid(x)	计算sigmoid函数
tf.tanh(x)	计算tanh函数

池化函数与损失函数

操作	说明
tf.nn.avg_pool(value, ksize, strides, padding)	平均方式池化
tf.nn.max_pool(value, ksize, strides, padding)	最大值方法池化
tf.nn.max_pool_with_argmax(input, ksize, strides,padding)	返回一个二维元组(output,argmax),最大值pooling，返回最大值及其相应的索引
tf.nn.l2_loss(t)	output = sum(t ** 2) / 2

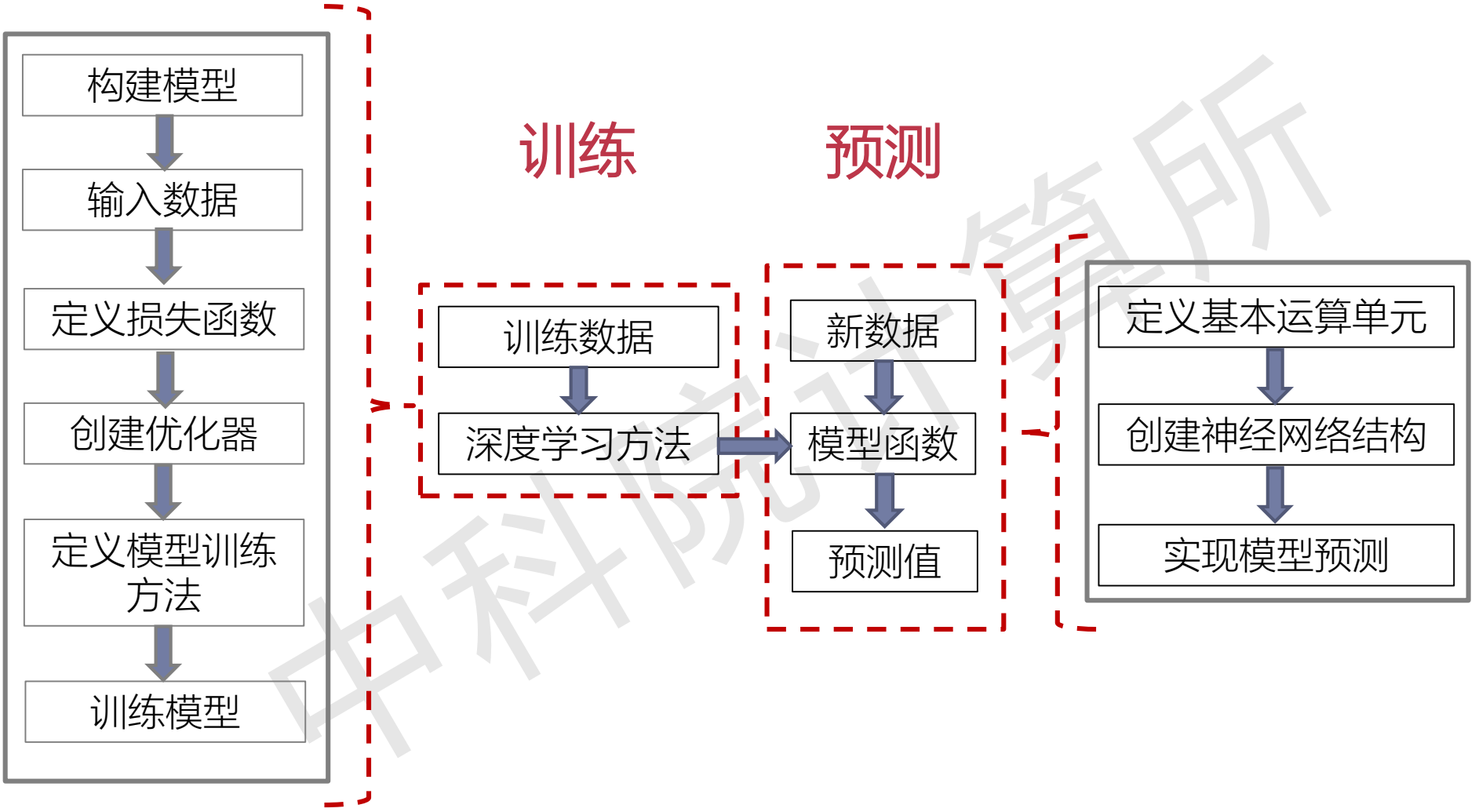
3、创建神经网络结构

```
def build_vggnet(vgg19_npy_path):
  models = {}
  models['input'] = tf.Variable(np.zeros((1, 224, 224, 3)).astype('float32'))
  models['conv1_1'] = basic_calc('conv',models['input'],read_wb(vgg19_npy_path,'conv1_1'))
  models['conv1_2'] = basic_calc('conv',models['conv1_1'],read_wb(vgg19_npy_path,'conv1_2'))
  models['pool1']   = basic_calc('pool',models['conv1_2'])
  models['conv2_1'] = basic_calc('conv',models['pool1'],read_wb(vgg19_npy_path,'conv2_1'))
  models['conv2_2'] = basic_calc('conv',models['conv2_1'],read_wb(vgg19_npy_path,'conv2_2'))
  models['pool2']   = basic_calc('pool',models['conv2_2'])
  models['conv3_1'] = basic_calc('conv',models['pool2'],read_wb(vgg19_npy_path,'conv3_1'))
  models['conv3_2'] = basic_calc('conv',models['conv3_1'],read_wb(vgg19_npy_path,'conv3_2'))
  models['conv3_3'] = basic_calc('conv',models['conv3_2'],read_wb(vgg19_npy_path,'conv3_3'))
  models['conv3_4'] = basic_calc('conv',models['conv3_3'],read_wb(vgg19_npy_path,'conv3_4'))
  models['pool3']   = basic_calc('pool',models['conv3_4'])
  models['conv4_1'] = basic_calc('conv',models['pool3'],read_wb(vgg19_npy_path,'conv4_1'))
  models['conv4_2'] = basic_calc('conv',models['conv4_1'],read_wb(vgg19_npy_path,'conv4_2'))
  models['conv4_3'] = basic_calc('conv',models['conv4_2'],read_wb(vgg19_npy_path,'conv4_3'))
  models['conv4_4'] = basic_calc('conv',models['conv4_3'],read_wb(vgg19_npy_path,'conv4_4'))
  models['pool4']   = basic_calc('pool',models['conv4_4'])
  models['conv5_1'] = basic_calc('conv',models['pool4'],read_wb(vgg19_npy_path,'conv5_1'))
  models['conv5_2'] = basic_calc('conv',models['conv5_1'],read_wb(vgg19_npy_path,'conv5_2'))
  models['conv5_3'] = basic_calc('conv',models['conv5_2'],read_wb(vgg19_npy_path,'conv5_3'))
  models['conv5_4'] = basic_calc('conv',models['conv5_3'],read_wb(vgg19_npy_path,'conv5_4'))
  models['pool5']   = basic_calc('pool',models['conv5_4'])
  return models
```

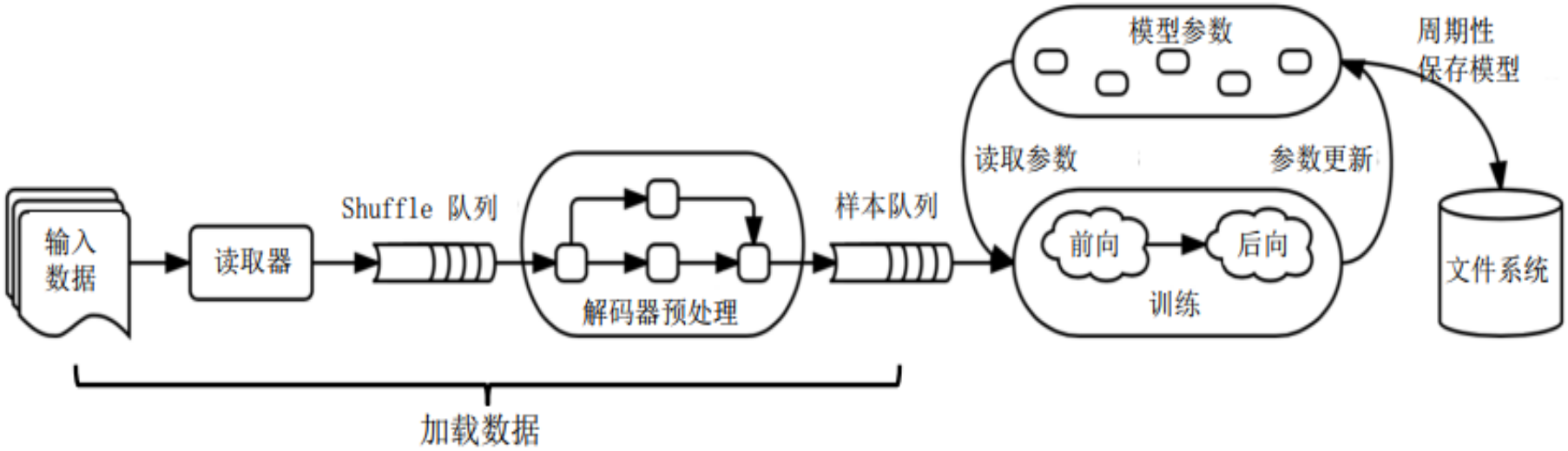


4、计算模型输出

```
#模型文件路径，需要加载
vgg19_npy_path = './vgg_models.npy'
#获取输入的内容图像
img_content = load_image('./content.jpg')

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  models = build_vggnet(vgg19_npy_path)

  sess.run(models['input'].assign(img_content))
  res = sess.run(models['pool5'])

  # other process on res
  ...
```


构建模型
输入数据
定义损失函数
创建优化器
定义模型训练方法
训练模型
训练
预测
训练数据
深度学习方法
新数据
模型函数
预测值
定义基本运算单元
创建神经网络结构
实现模型预测

利用TensorFlow实现简单的模型训练



```python
def train_vgg():
    sess = tf.Session()
    #构建模型，使用与模型推理时相同的网络结构
    models = build_vggnet(vgg19_npy_path)
    #获取输入的内容图像、风格图像
    img_content = load_image('./content.jpg')
    img_style = load_image('./style.jpg')
    #生成噪声图像img_random
    img_random = get_random_img(img_content)
    sess.run(tf.global_variables_initializer())
    #定义损失函数
    total_loss = loss(sess, models,img_content,img_style)
    #创建优化器
    optimizer = tf.train.AdamOptimizer(2.0)
    #定义模型训练方法
    train_op = optimizer.minimize(total_loss)
    sess.run(tf.global_variables_initializer())
    #使用噪声图像img_random进行训练
    sess.run(models['input'].assign(img_random))

    for i in range(3000):
        #完成一次反向传播
        sess.run(train_op)
        if i % 100 == 0:
            #每完成100次训练即打印中间结果，从而监测训练效果
            img_transfer = sess.run(models['input'])
            print('Iteration %d' % (i))
            print('cost: ', sess.run(total_loss))

    #训练结束，保存训练结果，显示图像
    ...

if __name__ == '__main__':
    train_vgg()
```

构建模型

1、加载数据

2、定义损失函数

3、创建优化器

4、定义模型训练方法

训练模型

1、加载数据

- 注入（feeding）：利用 feed_dict 直接传递输入数据
- 预取（pre_load）：利用 Const和 Variable直接读取输入数据
- 基于队列API：基于队列相关的API来构建输入流水线（Pipeline）
- tf.data API：利用tf.data API来构建输入流水线

注入

```
with tf.Session():
  input = tf.placeholder(tf.float32)
  classifier = ...
  print(classifier.eval(feed_dict={input: one_numpy_ndarray}))
```

预取

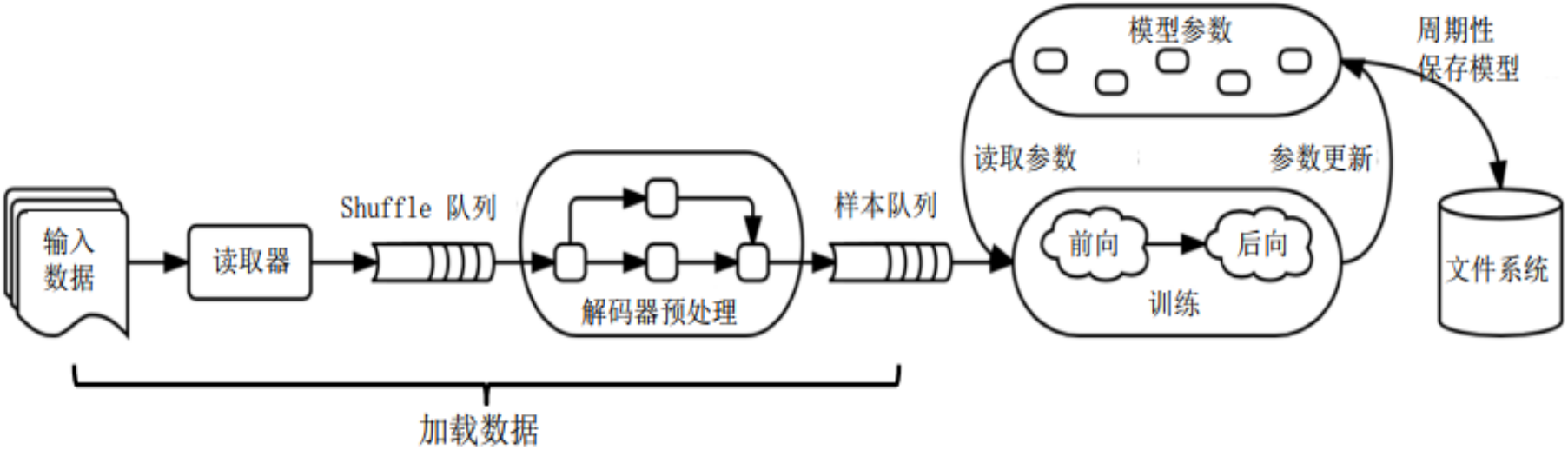
- 利用常量进行数据预取

```
training_data = ...  #some data like numpy array
training_labels = ...
with tf.Session():
  input_data = tf.constant(training_data)
  input_labels = tf.constant(training_labels)
  ...
```

- 利用变量进行数据预取

```
training_data = ...  #some data like numpy array
training_labels = ...
with tf.Session() as sess:
  data_initializer = tf.placeholder(dtype=training_data.dtype,
                                    shape=training_data.shape)
  label_initializer = tf.placeholder(dtype=training_labels.dtype,
                                     shape=training_labels.shape)
  input_data = tf.Variable(data_initializer, trainable=False, collections=[])
  input_labels = tf.Variable(label_initializer, trainable=False, collections=[])
  ...
  sess.run(input_data.initializer,
           feed_dict={data_initializer: training_data})
  sess.run(input_labels.initializer,
           feed_dict={label_initializer: training_labels})
  ...
```

基于队列API构建输入流水线



- 基于队列的输入流水线具有如下要素：
  - 文件名称组成的列表
  - 保存文件名称的FIFO队列
  - 相应文件格式的读取器
  - 解码器
  - 生成的样本队列（ShuffleQueue）

- 一个典型的输入结构：使用一个队列作为模型训练的输入，多个线程准备训练样本，一个训练线程执行一个训练操作
- TensorFlow提供了两个类来帮助多线程的实现：
  - Coordinator类：同时停止多个工作线程
  - QueueRunner类：协调多个工作线程同时将多个张量推入同一个队列中

```
#用来保存文件名称的FIFO队列
filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])

#根据csv文件选用的读取器，不同种类文件读取器种类也不同
reader = tf.TextLineReader()
key, value = reader.read(filename_queue)

#为处理出现空值情况设置的默认值，以及给定解码器输出的类型
record_defaults = [[1], [1], [1], [1], [1]]
col1, col2, col3, col4, col5 = tf.decode_csv(
    value, record_defaults=record_defaults)
features = tf.stack([col1, col2, col3, col4])

with tf.Session() as sess:
  #创建协调器
  coord = tf.train.Coordinator()
  #在调用run或eval执行读取之前，必须用tf.train.start_queue_runners来填充队列
  threads = tf.train.start_queue_runners(coord=coord)

  for i in range(10):
    #拿到每次读取的结果
    example, label = sess.run([features, col5])
    print "example = ", example, ", label = ", label

  coord.request_stop()
  coord.join(threads)
```

- file0.csv和file1.csv中的内容

```
#'file0.csv'
111, 222, 333, 444, 555
222, 333, 444, 555, 666
333, 444, 555, 666, 777
444, 555, 666, 777, 888

#'file1.csv'
555, 444, 333, 222, 111
666, 555, 444, 333, 222
777, 666, 555, 444, 333
888, 777, 666, 555, 444
```

- 随机生成的样本队列

```
#'file0.csv'
example = [111  222  333  444] , label = 555
example = [222  333  444  555] , label = 666
example = [333  444  555  666] , label = 777
example = [444  555  666  777] , label = 888
example = [555  444  333  222] , label = 111
example = [666  555  444  333] , label = 222
example = [777  666  555  444] , label = 333
example = [888  777  666  555] , label = 444
example = [555  444  333  222] , label = 111
example = [666  555  444  333] , label = 222
```

- 数据的批量处理

```
def read_my_file_format(filename_queue):
  reader = tf.SomeReader()
  key, record_string = reader.read(filename_queue)
  example, label = tf.some_decoder(record_string)
  processed_example = some_processing(example)
  return processed_example, label

def input_pipeline(filenames, batch_size, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example, label = read_my_file_format(filename_queue)
  #出队操作后的所剩数据的最小值
  min_after_dequeue = 10000
  #队列的容量
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch(
      [example, label], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch, label_batch
```

利用tf.data API构建输入流水线

- 包含两个基础类：Dataset和Iterator
- Dataset是一类相同类型元素的序列，每个元素由一个或多个张量组成
- 创建一个Dataset有两种方法：
  - 通过不同的API来读取不同类型的源数据，返回一个Dataset
  - 在已有Dataset基础上通过变换得到新的Dataset，包括map、shuffle、batch和repeat等等

- 利用Iterator类来读取创建好的Dataset中的数据
- 常用的Iterator：
  - one-shot iterator：单次迭代器。一次遍历所有元素
  - initializable iterator：可初始化迭代器。需要显式运行初始化操作
  - reinitializable iterator：可重新初始化迭代器。可以通过多个不同的 Dataset 对象进行初始化
  - feedable iterator：可馈送迭代器。与feed_dict配合使用

2、定义损失函数

- 自定义损失函数示例

```
def loss(sess, models,img_content,img_style):
    #计算内容损失函数
    sess.run(models['input'].assign(img_content))
    #内容图像在conv4_2层的特征矩阵
    p = sess.run(models['conv4_2'])
    #输入图像在conv4_2层的特征矩阵
    x = models['conv4_2']
    M = p.shape[1] * p.shape[2]
    N = p.shape[3]
    content_loss = (1.0 / (4 * M * N)) * tf.reduce_sum(tf.pow(p - x, 2))

    #计算风格损失函数
    sess.run(models['input'].assign(img_style))
    style_loss = 0.0
    for layer_name, w in STYLE_LAYERS:
        #风格图像在layer_name各层的特征矩阵
        a = sess.run(models[layer_name])
        #输入图像在layer_name各层的特征矩阵
        x = models[layer_name]
        M = a.shape[1] * a.shape[2]
        N = a.shape[3]
        A = gram_matrix(a, M, N)
        G = gram_matrix(x, M, N)
        style_loss += (1.0 / (4 * N ** 2 * M ** 2)) * tf.reduce_sum(tf.pow(G - A, 2)) * w

    total_loss = ALPHA * content_loss + BETA * style_loss
    return total_loss

def gram_matrix(x, M, N):
    x = tf.reshape(x, (M, N))
    return tf.matmul(tf.transpose(x), x)
```

自定义损失函数

- 基本函数

类别	基本函数及功能说明
四则运算	tf.add(), tf.sub(),tf.mul()
科学计算	tf.abs(), tf.square(), tf.sin()
比较操作	tf.greater()（返回True或False）
条件判断	tf.where(condition, tensor_x, tensor_y) （condition为True时返回tensor_x，否则返回tensor_y）
降维操作	tf.reduce_sum()，tf.reduce_mean()（将高维矩阵元素以求和或均值的方式变为一维）

- 示例

```
loss = tf.reduce_mean(tf.square(y-y_data))
#实际值和预测值的差值平方再求平均值，训练的目的就是要让这个loss越来越小
```

TensorFlow内置的4个损失函数

- 1）softmax交叉熵：

tf.nn.softmax_cross_entropy_with_logits(lables,logits)

- 2）加了稀疏的softmax交叉熵

tf.nn.sparse_softmax_cross_entropy_with_logits(lables,logits)

参数	含义
logits	网络最后一层的输出
labels	标签，分类或分割等问题中的标准答案

- 3）sigmoid交叉熵

tf.nn.sigmoid_cross_entropy_with_logits(lables,logits)

- 计算公式

loss=labels*-log(sigmoid(logits))+(1-labels)*-log(1-sigmoid(logits))

- 令x=logits，z=labels，则有：

$$loss = z * -\log\big(sigmoid(x)\big) + (1-z) * -\log\big(1-sigmoid(x)\big)$$

$$= z * -\log\left(\frac{1}{1+e^{-x}}\right) + (1-z) * -log\frac{e^{-x}}{1+e^{-x}}$$

$$= z * \log(1+e^{-x}) + (1-z) * log\frac{1+e^{-x}}{e^{-x}}$$

$$= \mathrm{z} * \log(1+e^{-x}) + (1-\mathrm{z}) * \big(\log(1+e^{-x})\big) - \log(e^{-x}))$$

$$= \log(1+e^{-x}) + x(1-z) = x - xz + \log(1+e^{-x})$$

- 当 x<0时，为防止e^{-x} 项溢出，此时应为

$$loss = x - xz + \log(1 + e^{-x}) = -xz + loge^{x} + \log(1 + e^{-x})$$

$$= -xz + \log(1 + e^{x})$$

- 即：

x>0时，$loss = x - xz + \log(1 + e^{-x})$

x<0时，$loss = -xz + \log(1 + e^{x})$

- 因此，实际计算时，为了保证稳定性并防止溢出，使用：

$$loss = \max(x, 0) - xz + \log(1 + e^{-|x|})$$

- 4）带权重的sigmoid交叉熵：
  tf.nn.weighted_cross_entropy_with_logits(targets, logits, pos_weight)
- 该函数功能及计算方法与tf.nn.sigmoid_cross_entropy_with_logits类似，但加了权重功能，是计算具有权重的sigmoid交叉熵函数
- 目的：增加或减小正样本在计算交叉熵时的loss
- 计算方法：loss=-pos_weight*targets∗log(sigmoid(logits))-(1−targets)∗log(1−sigmoid(logits))

- 令x=logits，z=targets，q=pos_weight
- 则有：

$$loss = qz * -\log\big(sigmoid(x)\big) + (1-z) * -\log\big(1-sigmoid(x)\big)$$

$$= qz * -\log\left(\frac{1}{1+e^{-x}}\right) + (1-z) * -log\frac{e^{-x}}{1+e^{-x}}$$

$$= \mathrm{qz} * \log(1+e^{-x}) + (1-\mathrm{z}) * \big(\log(1+e^{-x})\big) - \log(e^{-x}))$$

$$= qz * log(1+e^{-x}) + (1-z) * (x + \log(1+e^{-x}))$$

$$= (1-z) * x + (qz + 1 - z) * \log(1+e^{-x})$$

$$= (1-z) * x + (1 + (q-1) * z) * log(1+e^{-x})$$

- 令l= (1 + (q - 1) * z)，为保证稳定性并避免溢出，实现中采用：

$$loss = \max(x, 0) - xz + l * log\big(1 + e^{-|x|}\big)$$

2、创建优化器

- 优化器的功能是实现优化算法，可以自动为用户计算模型参数的梯度值
- TensorFlow中支持的优化器函数
  - tf.train.Optimizer
  - tf.train.GradientDescentOptimizer：梯度下降优化器
  - tf.train.AdadeltaOptimizer
  - tf.train.AdagradOptimizer
  - tf.train.AdagradDAOptimizer
  - tf.train.MomentumOptimizer：动量梯度下降优化器
  - tf.train.AdamOptimizer：Adam算法优化器
  - tf.train.FtrlOptimizer
  - tf.train.ProximalGradientDescentOptimizer
  - tf.train.ProximalAdagradOptimizer
  - tf.train.RMSPropOptimizer

- tf.train.GradientDescentOptimizer：梯度下降优化器
  - 返回一个优化器，参数为learningRate
  - 用法：

```
train=tf.train.GradientDescentOptimizer(learningRate)
```

- tf.train.AdamOptimizer：Adam算法优化器
  - 返回一个使用Adam算法的优化器，参数为learningRate
  - Adam算法：综合了 Momentum 和 RMSProp 方法，根据损失函数对每个参数的梯度的一阶矩估计和二阶矩估计动态调整针对于每个参数的学习率
  - 用法：

```
train=tf.train.AdamOptimizer(learningRate)
```

3、定义模型训练方法

- 一般采用最小化损失函数（minimize）的方法

```
train_op = tf.train.Optimizer.minimize(loss,global_step=None,var_list=None)
```

- 常用训练操作：

操作	功能
tf.train.Optimizer.minimize(loss, global_step=None, var_list=None)	使用最小化损失函数的方法来训练模型。执行该操作时会内部依次调用compute_gradients和apply_gradients操作
tf.train.Optimizer.compute_gradients(loss,var_list=None)	对var_list中列出的模型参数计算梯度，返回（梯度，模型参数）组成的列表
tf.train.Optimizer.apply_gradients(grads_and_vars)	将计算出的梯度更新到模型参数上，返回更新参数的操作

- minimize方法可以直接用于模型训练，简单有效
- 对于需要对梯度进行其他处理的优化器，其设计流程为：
  - 1、使用compute_gradients()方法计算出梯度
  - 2、按需求处理梯度，如进行裁剪、加权、平均等
  - 3、使用apply_gradients()方法将处理后的梯度值更新到模型参数中

对梯度的处理

- 对于模型层次较多的网络，由于输入数据不合法、求导精度限制等原因，可能出现梯度爆炸或梯度消失的问题，使得模型训练无法快速收敛
- 解决方法：
  - 1、减小学习率
  - 2、梯度裁剪（Gradient Clipping）。

示例：对梯度的L2范式进行裁剪

梯度的L2范式为：$\|t\|_2 = \sqrt{grad(w_1)^2 + grad(w_2)^2 + \cdots}$

设置裁剪阈值c，则当$\|t\|_2 >$c时，$\mathrm{grad}(w_i) = \frac{c}{\|t\|_2} \cdot \mathrm{grad}(w_i)$

当$\|t\|_2 \leq$c时，$\mathrm{grad}(w_i)$不变

TensorFlow中内置的梯度处理功能

方法	功能
tf.clip_by_value(t,clip_value_min,clip_value_max)	将梯度t裁剪到[clip_value_min,clip_value_max]区间
tf.clip_by_norm(t,clip_norm)	对梯度t的L2范式进行裁剪，clip_norm为裁剪阈值
tf.clip_by_average_norm(t, clip_norm)	对梯度t的平均L2范式进行裁剪，clip_norm为裁剪阈值
tf.clip_by_global_norm(t_list, clip_norm)	对梯度t_list进行全局规范化加和裁剪，clip_norm为裁剪阈值
tf.global_norm(t_list)	计算t_list中所有梯度的全局范式

```python
#创建Adam优化器
optimizer = tf.train.AdamOptimizer(learning_rate)

#计算梯度
grads = optimizer.compute_gradients(loss)

#对梯度的L2范数进行裁剪
grads = tf.clip_by_norm(grads,clip_norm)

#更新模型参数
train_op = optimizer.apply_gradients(grads)

```

4、模型保存

- 在模型训练过程中，使用tf.train.Saver()来保存模型中的所有变量

```
import tensorflow as tf
weights = tf.Variable(tf.random_normal([30,60],stddev=0.35),name="weights")
w2 = tf.Variable(weights.initialized_value(),name="w2")

#实例化saver对象
saver = tf.train.Saver()

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for step in xrange(1000000):
    #执行模型训练
    sess.run(training_op)
    if step % 1000 == 0:
        # 将训练得到的变量值保存到检查点文件中
        saver.save(sess, './ckpt/my-model')
```

恢复模型

- 当需要基于某个checkpoint继续训练模型参数时，需要从.ckpt文件中恢复出已保存的变量
- 同样使用tf.train.Saver()来恢复变量，恢复变量时不需要先初始化变量

```
import tensorflow as tf

weights = tf.Variable(tf.random_normal([30,60],stddev=0.35),name="weights")
w2 = tf.Variable(weights.initialized_value(),name="w2")

#模型路径只需要给出文件夹名称
model_path = "./ckpt"

#实例化saver对象
saver = tf.train.Saver()

with tf.Session() as sess:
  #找到存储变量值的位置
  ckpt = tf.train.latest_checkpoint(model_path)
  #恢复变量
  saver.restore(sess,ckpt)
  print(sess.run(weights))
  print(sess.run(w2))
```

图像风格迁移训练的实现

```python
def train_vgg():
    sess = tf.Session()
    #构建模型，使用与模型推理时相同的网络结构
    models = build_vggnet(vgg19_npy_path)
    #获取输入的内容图像、风格图像
    img_content = load_image('./content.jpg')
    img_style = load_image('./style.jpg')
    #生成噪声图像img_random
    img_random = get_random_img(img_content)
    sess.run(tf.global_variables_initializer())
    #定义损失函数
    total_loss = loss(sess, models,img_content,img_style)
    #创建优化器
    optimizer = tf.train.AdamOptimizer(2.0)
    #定义模型训练方法
    train_op = optimizer.minimize(total_loss)
    sess.run(tf.global_variables_initializer())
    #使用噪声图像img_random进行训练
    sess.run(models['input'].assign(img_random))

    for i in range(3000):
        #完成一次反向传播
        sess.run(train_op)
        if i % 100 == 0:
            #每完成100次训练即打印中间结果，从而监测训练效果
            img_transfer = sess.run(models['input'])
            print('Iteration %d' % (i))
            print('cost: ', sess.run(total_loss))

    #训练结束，保存训练结果，显示图像
    ...

if __name__ == '__main__':
    train_vgg()
```

小结

- 深度学习编程框架的概念

  深度学习编程框架的概念及分类

- TensorFlow概述

  TensorFlow的历史、发展历程

- TensorFlow编程模型及基本用法

  TensorFlow中常用的计算图、张量、操作、会话、变量、占位符、队列等概念

- 基于TensorFlow的训练及预测实现

  TensorFlow模型训练、预测等

谢谢大家！