

题 目 深度学习开源库与LabVIEW的集成实现与应用

软件学院 院（系） 软件工程 专业

学 号 71114343

学生姓名 包莹星

校内导师 曹玖新

企业导师 郭翘

起止日期 2018.01-2018.06

设计地点 美国国家仪器有限公司

摘 要

针对谷歌发布的TensorFlow及微软发布的CNTK深度学习框架进行研究，学习使用卷积神经网络进行图像分类。最终在LabVIEW上编写了一套APIs以调用TensorFlow和CNTK完成各项功能， 使LabVIEW用户可以直接在LabVIEW上完成图像分类的训练和预测等任务。

主要内容如下：

1. 基于Python toolkit 及Python node in LabVIEW 2018 方式设计一套LabVIEW APIs，调用 CNTK python编写的以Python函数为单位的接口（一个API调用一个python函数实现一项功能），分别实现图片集分类的训练，预测以及模型文件的保存及读取。
2. 基于Python node in LabVIEW 2018 方式设计一套LabVIEW APIs，调用 TensorFlow python编写的以Python函数为单位的接口（一个API调用一个python函数实现一项功能），分别实现图片集分类的训练，预测以及模型文件的保存及读取。
3. 基于call library node 方式编写一组LabVIEW APIs，调用 TensorFlow C编写的动态链接库，读取（加载，运行和关闭模型文件）TensorFlow模型文件，针对给定的图片进行预测类别。

**关键词：**

ABSTRACT

目录

[摘 要 I](#_Toc484621900)

[ABSTRACT II](#_Toc484621901)

[第一章 绪论 1](#_Toc484621902)

[1.1 研究背景及意义 1](#_Toc484621903)

[1.2 相关研究现状 1](#_Toc484621904)

[1.3 论文组织结构 2](#_Toc484621905)

[1.4 本章小结 2](#_Toc484621906)

[第二章 相关知识概述 3](#_Toc484621907)

[2.1 卷积神经网络（CNN） 3](#_Toc484621908)

[2.1.1 CNN结构 3](#_Toc484621909)

[2.1.2 Facebook介绍及相应爬取方法 3](#_Toc484621910)

[2.1.3 Instagram介绍及相应爬取方法 4](#_Toc484621911)

[2.1.4 Foursquare介绍及相应爬取方法 4](#_Toc484621912)

[2.1.5 跨社交网络重叠用户 4](#_Toc484621913)

[2.1.6 About.me介绍及相应爬取方法 4](#_Toc484621914)

[2.2 数据集 5](#_Toc484621915)

[2.3 用户兴趣标签 5](#_Toc484621916)

[2.4 本章小结 6](#_Toc484621917)

[第三章 文本分析相关介绍 7](#_Toc484621918)

[3.1 概述 7](#_Toc484621919)

[3.2 文本信息特点 7](#_Toc484621920)

[3.3 文本预处理 8](#_Toc484621921)

[3.3.1 噪声过滤 8](#_Toc484621922)

[3.3.2 文本分词 8](#_Toc484621923)

[3.3.3 词态整理 9](#_Toc484621924)

[3.3.4 停用词过滤 9](#_Toc484621925)

[3.3.5 整合 10](#_Toc484621926)

[3.4 主题建模与LDA 10](#_Toc484621927)

[3.5 用户兴趣模型 12](#_Toc484621928)

[3.5.1 LDA模型局限 12](#_Toc484621929)

[3.5.2 模型设计 13](#_Toc484621930)

[3.6 本章小结 13](#_Toc484621931)

[第四章 系统设计 15](#_Toc484621932)

[4.1 系统概述 15](#_Toc484621933)

[4.2 相关技术简介 15](#_Toc484621934)

[4.2.1 Python与pycharm 15](#_Toc484621935)

[4.2.2 网页爬取与scrapy 16](#_Toc484621936)

[4.2.3 OSN官方API 16](#_Toc484621937)

[4.2.4 MongoDB 17](#_Toc484621938)

[4.2.5 文本分析相关库 17](#_Toc484621939)

[4.2.6 LDA相关库 17](#_Toc484621940)

[4.2.7 可视化相关库 17](#_Toc484621941)

[4.3 数据爬取功能 18](#_Toc484621942)

[4.3.1 重叠用户发现模块 19](#_Toc484621943)

[4.3.2 Twitter爬取模块 19](#_Toc484621944)

[4.3.3 Foursquare爬取模块 21](#_Toc484621945)

[4.3.4 Facebook爬取模块 22](#_Toc484621946)

[4.3.5 Instagram爬取模块 24](#_Toc484621947)

[4.3.6 重试机制 24](#_Toc484621948)

[4.3.7 日志框架 25](#_Toc484621949)

[4.4 数据分析功能 26](#_Toc484621950)

[4.4.1 数据预处理模块 26](#_Toc484621951)

[4.4.2 数据建模模块 28](#_Toc484621952)

[4.5 展示功能 29](#_Toc484621953)

[4.5.1 数据展示模块 29](#_Toc484621954)

[4.5.2 结果展示模块及实时建模模块 29](#_Toc484621955)

[4.6 本章小结 29](#_Toc484621956)

[第五章 系统展示与分析 30](#_Toc484621957)

[5.1 系统展示 30](#_Toc484621958)

[5.2 实验分析 34](#_Toc484621959)

[5.3 本章小结 35](#_Toc484621960)

[第六章 总结与展望 36](#_Toc484621961)

[6.1 总结 36](#_Toc484621962)

[6.2 展望 36](#_Toc484621963)

[参考文献 37](#_Toc484621964)

[致 谢 38](#_Toc484621965)

第一章 绪论

1.1 研究背景及意义

随着大数据时代的到来，深度学习技术已经成为当前人工智能领域的一个研究热点，其已在图像识别、语音 识别、自然语言处理、搜索推荐等领域展现出了巨大的优势，并且仍在继续发展变化。 随着深度学习技术的兴起

无论是工业界、学术界还是广大群众都投身到了深度学习的洪流之中。在学术界，2012年,Krizhevsky等人训练出了AlexNet，以压倒性优势拿到了ILSVRC 2012的冠军，使得深度学习开始为人们所关注。2013年，Zeiler和Rob通过对CNN进行可视化，对AlexNet进行了改进，得到了ZFNet。2014年,VGGNet和GoogleNet通过增加更多的卷积层，将模型的最终性能又向前推进了一步。2015年，何恺明等人提出了残差结构，解决了在训练极深网络时遇到的降解(degradation)问题，并训练出了达到152层的CNN模型，一举拿下了多项大赛的冠军，在ILSVRC 2015上的top-5错误率只有3.57%，甚至远远超过了人类在这一数据集上的top-5错误率(5%)。在工业界，谷歌（Google）、脸书（Facebook）、百度、阿里巴巴等一系列国内外大公司纷纷对外公开宣布了人工智能将作为他们下一个战略重心。在人才方面，继深度学习界泰斗吴恩达（Andrew Ng）加入百度、Yann LeCun 加入脸书之后，各大 IT 公司开始哄抢学术界大牛。斯坦福大学教授、计算机视觉领域领军人物李飞飞（Feifei Li）于去年 11 月加入谷歌；卡内基梅隆大学教授、机器学习领域顶级人物 Alex Smola 于去年 6 月加入亚马逊（Amazon）。在工具方面，谷歌、脸书、百度、微软、亚马逊等公司相继开源了各自的深度学习框架。

深度学习可以帮助我们解决对于传统的机器视觉工具来说极具挑战性，甚至不可能的任务，因此，研究开发深度学习工具是非常符合客户和行业需求的。为了在企业开发机器学习套件AML 2.0中提供意见和参考，本文在针对谷歌发布的TensorFlow及微软发布的CNTK深度学习框架进行了研究，并在LabVIEW上编写了一套APIs调用TensorFlow和CNTK完成各项任务。

1.2 相关研究现状

1956年，几个计算机科学家相聚在达特茅斯会议（Dartmouth Conferences），提出了“人工智能”的概念。其中，机器学习是人工智能的一个分支，机器学习最基本的做法，是使用算法来解析数据、从中学习，然后对真实世界中的事件做出决策和预测。机器学习发展分为两个阶段，起源于上世纪20年代的浅层学习（Shallow Learning）和最近几年发展迅速的深度学习（Deep Learning）。

1943年，人工神经网络这一概念首次被提出，开启了人们对人工神经网络的研究 。1957年，感知器（Perceptron）人工神经网络模型被提出并实现，而Perceptron不仅是卷积网络，也是神经网络的始祖。

20世纪80年代, 日本科学家Kunihiko Fukushima提出Neocognitron（神经认知机）, Neocognitron是一种多层级的神经网络，具有一定程度的视觉认知的功能，并直接启发了后来的卷积神经网络。1997,CNN之父Yann LeCun提出LeNet-5，这是个多层级联的卷积结构，可对手写数字进行有效识别[1]。

此后，深度学习研究一直停滞不前，直到2006年，加拿大多伦多大学教授Geoffrey Hinton对深度学习的提出以及模型训练方法的改进打破了BP神经网络发展的瓶颈[2]，使得深度学习受到科研机构、工业界的高度关注。



图1 卷积神经网络发展图

从图1中可以看出，前三次关于卷积神经网络的技术突破，间隔时间非常长，需要十余年甚至更久才出现一次理论创新。而后于2012年，杰弗里·辛顿的学生Krizhevsky等人在图片分类比赛ImageNet中训练出了AlexNet[3]，以压倒性优势拿到了ILSVRC 2012的冠军。瞬间点燃了卷积神经网络研究的热潮。AlexNet成功应用了ReLU激活函数、Dropout、最大覆盖池化、LRN层、GPU加速等新技术，并启发了后续更多的技术创新。在AlexNet之后，卷积神经网络的发展可分为两类，一类是网络结构上的改进调整（图1中的左侧分支），另一类是网络深度的增加（图1中的右侧分支）。

2013年，颜水成教授发表Network in Network[4]，优化了卷积神经网络的结构，并推广了1\*1的卷积结构。2014年的Google Inception Net V1[5]，提出了Inception Module这个可以反复堆叠的高效的卷积网络结构，并获得了当年ILSVRC比赛的冠军。2015年初的Inception V2[6]提出了Batch Normalization，大大加速了训练过程，并提升了网络性能。2015年年末的Inception V3[7]则继续优化了网络结构，提出了Factorization in Small Convolutions的思想，分解大尺寸卷积为多个小卷积乃至一维卷积。

在另一条分支上，许多研究工作致力于加深网络层数，2014年，ILSVRC比赛的亚军VGGNet[8]全程使用3\*3的卷积，成功训练了深达19层的网络。2015年，微软的ResNet[9]成功训练了152层深的网络，一举拿下了当年ILSVRC比赛的冠军，top-5错误率降低至3.46%。其后又更新了ResNet V2[10]，增加了Batch Normalization，并去除了激活层而使用Identity Mapping或Preactivation，进一步提升了网络性能。此后，Inception ResNet V2融合了Inception Net优良的网络结构，和ResNet训练极深网络的残差学习模块，集两个方向之长，取得了更好的分类效果。

自AlexNet于2012年提出后，深度学习领域的研究发展极其迅速，基本上每年甚至每几个月都会出现新一代的技术。除了学术界，深度学习在工业界也持续升温， 2012年6月，谷歌首席架构师Jeff Dean和斯坦福大学教授AndrewNg主导著名的GoogleBrain项目，采用16万个CPU来构建一个深层神经网络，并将其应用于图像和语音的识别，最终大获成功。2012年11月，微软公司在一次活动上公开展示了基于深度学习的同声传译系统，效果惊艳，其中语音识别部分的关键技术也是用深度网络完成的。

2016年3月，谷歌开发的围棋深度学习系统 AlphaGo 以总比分 4：1 战胜了韩国棋手李世石，成为第一个在 19×19 棋盘上战胜人类围棋冠军的智能系统。AlphaGo 战胜李世石把深度学习的概念从学术界推向了大众，并点燃了大众对于人工智能的巨大热情。继 AlphaGo 之后，谷歌的 DeepMind 团队将深度学习的技术用在了智能数据中心上。通过增强学习（reinforcement learning），新的数据中心智能系统可以更好的配合机器内的风扇和数据中心的空调使得既可以保证所有机器的散热，又可以最大限度的降低能源的消耗。

同时，各种深度学习框架也层出不穷[15]，Theano 是深度学习领域最早的软件平 台，专注于底层基本运算。 以 Caffe、Torch、MXNet、CNTK 为主的深度学习功能性平台提供了完备的基本模块，支持快速神经网络模型的创建和训练， 以Keras为主的深度学习抽象化平台。其本身不具有底层运算协调能力，而是依托于TensorFlow或Theano进行底层运算 ，而目前最火的当属框架Tensorflow。由 Google 基于 DistBelief 进行研发 的第二代人工智能系统，该平台吸取了已有平台的长处，既能 让用户触碰底层数据，又具有现成的神经网络模块，可以使用户非常快速的实现建模 。

随着神经网络模型层数越来越深[15]、节点个数越来越多，需要训练的数据集越来越大，模型的复杂度也越来越高，因此在模型的实际训练中单CPU或单GPU的加速方案存在着严重的性能不足，一般需要十几天的时间才能使得模型的训练得到收敛，已远远不能满足训练大规模神经网络、开展更多实验的需求。故多 CPU 或多 GPU 的加速方案成为训练大规模神经网络 模型的首选。但是由于在图像识别或语言识别类应用中，深度神经网络模型的计算量十分巨大，且模型层与层之间存在的一定的数据相关性，因此如何划分任务量以及计算资源是设计 CPU 或 GPU 集群加速框架的一个重要问题。

尽管深度学习技术在图像处理、语音识别、自然语言处理等领域取得了突破性的进展 ，深度学习目前仍有大量工作需要研究[16]。模型方面是否有其他更为有效且有理论依据的深度模型学习算法，探索新的特征提取模型是值得深入研究的内容。此外有效的可并行训练算法也是值得研究的一个方向。当前基于最小批处理的随机梯度优化算法很难在多计算机中进行并行训练。通常办法是 利用图形处理单元加速学习过程，然而单个机器 GPU 对大规模数据识别或相似任务数据集并不适用。在深度学习应用拓 展方面，如何充分合理地利用深度学习在增强传统学习算法的性能仍是目前各领域的研究重点。

通过对比比较英文文献的ACM，arXiv和中文文献的知网上引用量排名靠前的论文，不难看出，我国国内的研究水平正处于飞速上升阶段，从一开始的综述，已经在向算法方向上努力。同国内论文的使用框架进行研究不同，在2005-2010年之间，外国文献的研究已经取得了较高水平，而从2012年开始，他们就的研究就开始着手提升性能、运算时间，对于框架，算法和硬件关联都有所涉猎，同时也有经过不同手段预处理数据的论文，也有对于神经网络训练训练结果有一定思考和改良的论文。

1.3 课题主要工作

1.4 论文组织结构

本文的组织结构如下:

第二章 相关知识概述

2.1 卷积神经网络(CNN)

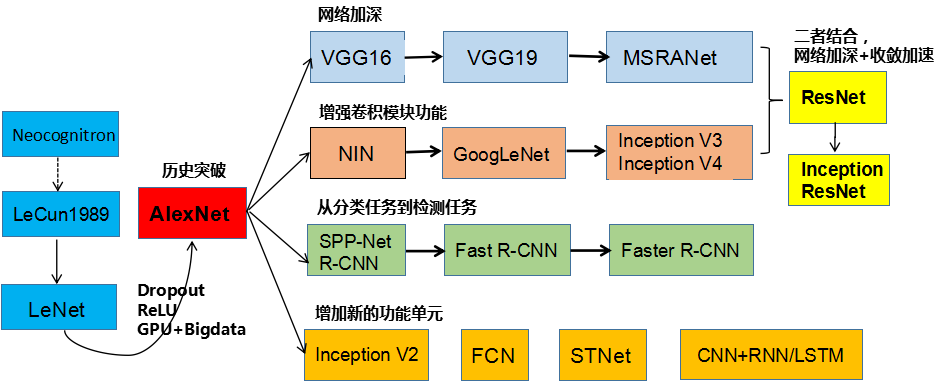


图1 卷积神经网络发展

卷积神经网络的发展起点是神经认知机模型，1989年诞生了经典的LeNet模型。LeNet模型虽小，但包含了卷积层、pooling层、全连接层这些现代CNN网络的基本组件。此后CNN的锋芒被SVM等手工设计的特征盖过。但随着ReLU和dropout的提出，以及GPU和大数据带来的历史机遇，CNN在2012年迎来了历史突破–AlexNet. AlexNet可以算是LeNet的一种更深更宽的版本，其获得了2012年ILSVRC比赛分类项目的冠军，top-5错误率16.4%，使用额外数据可达到15.3%。AlexNet包含了八个学习层——5个卷积层和3个全连接层。本文的网络架构采取了Alex Krizhevsky的AlexNet的简化版本，但并不与AlexNet完全相同。

**2.1.1 CNN结构**

我们首先LeNet5为例，介绍卷积神经网络中的一些组成

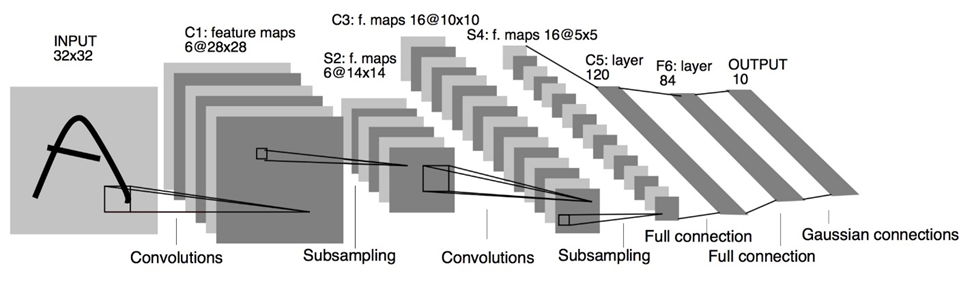
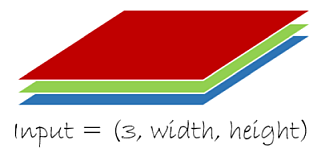


图2 LeNet5架构图

1. 输入层（input）

在处理图像的卷积神经网络中，输入数据通常会被规整成一个三维矩阵（分别是色彩通道数、图像宽度和高度），这会保存图像中像素之间的空间关系。



表现自然场景的图片经常会使用红绿蓝（RGB）三色通道，这种情况下输入数据大小就应该是（3,图像宽度,图像高度）或（图像宽度,图像高度，3）。

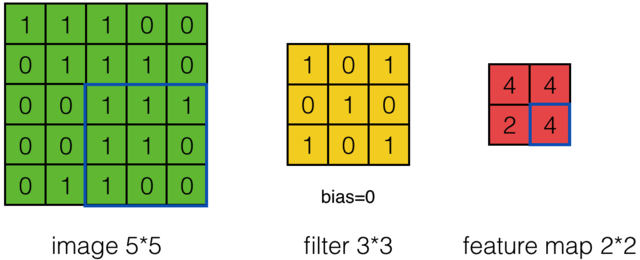
1. 卷积层（如图2 中的C1, C3, C5）

每个卷积层有多个特征图(feature map),每个特征图是通过一系列卷积过滤器(filter)提取输入的一种特征，然后每个特征图含有多个神经元。

* 滤波器

一个卷积层是一组滤波器，每个滤波器都由权重矩阵（W）和偏移量（b）定义。

滤波器扫描图片后对输入的值和权重进行点积运算，然后加上偏移量b值，就可以得到该像素点的输出值。每个滤波器都可以将图像生成为另一幅图像。比如两个滤波器就可以将生成两幅图像，这两幅图像可以看做是一张图像的不同的通道。



* 步幅参数和边界填充参数

通常来说滤波器会重复扫描，从左到右，从上到下。在大多数的自然场景图像处理中，每个卷积层有一个filter\_shape参数，制定滤波器的宽度和高度。然后还需要有个步幅参数（strides）来控制滤波器移动的幅度。最后还需要有一个布尔类型的参数pad，用于指示是否在输入图像的边界进行填充，以便于边界附近有一个完整的感受野。

以C1层进行说明：C1层是一个卷积层，有6个滤波器（提取6种局部特征），滤波器大小为5\*5，能够输出6个特征图Feature Map，大小为28\*28。C1有156个可训练参数（每个滤波器5\*5=25个unit参数和一个bias参数，一共6个滤波器，共(5\*5+1)6=156个参数），共156 (28\*28)=122,304个连接。）

1. 池化层（图2中的S2, S4）

池化层(Pooling层)也被称为下采样层，是为了降低网络训练参数及模型的过拟合程度。池化层节点上的计算要比普通前馈节点的计算简单得多，他没有权重，偏移量和激活函数，他使用一个简单的聚合函数（比如最大值函数、平均值函数）来求其输出值。

Max-Pooling: 选择Pooling窗口中的最大值作为采样值；

Mean-Pooling: 将Pooling窗口中的所有值相加取平均，以平均值作为采样值；

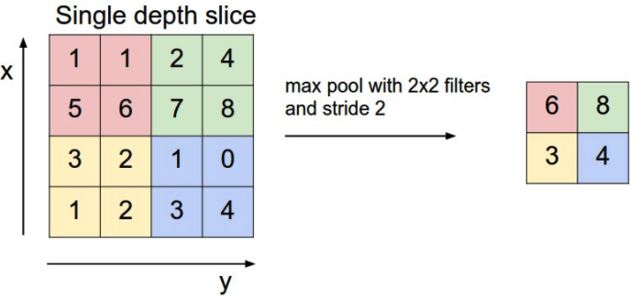


图3 一个2\*2的最大值池化示例

1. 全连接层（图2中的F6）

全连接指的是该层的每个神经元与上一层的所有神经元全部相连，全连接层一般用在CNN模型的最后几层，用于充当分类器的作用。如LeNet5中的C5和F6层就是全连接层。C5层的输入是16个5\*5的特征图(feature map),共16\*5\*5=400个输入神经元,C5的输出有120个神经元，所以C5层共有(400+1)\*120=48120个参数，其中120个是偏置参数。同理，F6层共有(120+1)\*84=10164个参数。

1. 输出层（Output）

输出层输出网络结果，每类一个单元。

**2.1.2 AlexNet**

AlexNet包含了6亿3000万个连接，6000万个参数和65万个神经元，拥有5个卷积层，其中3个卷积层后面连接了最大池化层，最后还有3个全连接层

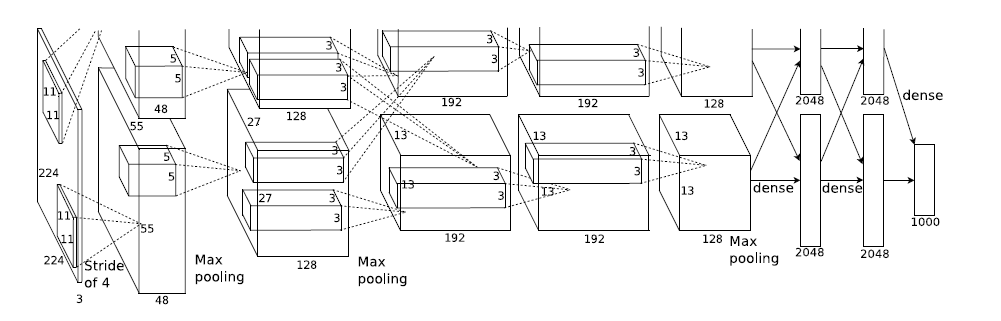


图4 AlexNet架构图

2012年，Hinton的学生Alex Krizhevsky提出了深度卷积神经网络模型AlexNet，它可以算是LeNet的一种更深更宽的版本。AlexNet可以说是神经网络在低谷期后的第一次发声，确立了深度学习（深度卷积网络）在计算机视觉的统治地位，同时也推动了深度学习在语音识别、自然语言处理、强化学习等领域的拓展。AlexNet中包含了几个比较新的技术点，也首次在CNN中成功应用了ReLU、Dropout和LRN等Trick。AlexNet将LeNet的思想发扬光大，把CNN的基本原理应用到了很深很宽的网络中。AlexNet主要使用到的新技术点如下。

1. 使用ReLU作为CNN的激活函数

激活函数常用于卷积层和全连接层之后，是用来引入非线性因素的，最开始使用的激活函数是sigmoid或tanh函数，例如上述的LeNet5中使用的就是sigmoid函数。但AlexNet成功使用ReLU作为CNN的激活函数，并验证其效果在较深的网络超过了Sigmoid，成功解决了Sigmoid在网络较深时的梯度弥散问题。现在已经取代sigmoid函数而广泛运用在各种CNN模型中。

1. Dropout

训练时使用Dropout随机忽略一部分神经元，以避免模型过拟合。所以每次输入一个样本，就相当于该神经网络就尝试了一个新的结构，但是所有这些结构之间共享权重。因为神经元不能依赖于其他特定神经元而存在，所以这种技术降低了神经元复杂的互适应关系。

1. 使用重叠的最大池化

此前CNN中普遍使用平均池化，AlexNet全部使用最大池化，避免平均池化的模糊化效果。并且AlexNet中提出让步长比池化核的尺寸小，这样池化层的输出之间会有重叠和覆盖，提升了特征的丰富性。

1. 局部响应归一化层（LRN）

本质上，这个层也是为了防止激活函数的饱和的。从功能上说，跟ReLU是重复的。本文中未使用LRN，而是在每层卷积层之后使用一个Batch Normalization（BN）层。

1. 数据增强

如果没有数据增强，仅靠原始的数据量，参数众多的CNN会陷入过拟合中，使用了数据增强后可以大大减轻过拟合，提升泛化能力。

AlexNet对数据做了以下处理：

随机crop。训练时候，对于256＊256的图片进行随机crop到224＊224，然后允许水平翻转，那么相当与将样本倍增到((256-224)^2)\*2=2048。

测试时候，对左上、右上、左下、右下、中间做了5次crop，然后翻转，共10个crop，之后对结果求平均。作者说，不做随机crop，大网络基本都过拟合(under substantial overfitting)。 对RGB空间做PCA，然后对主成分做一个(0, 0.1)的高斯扰动。结果让错误率又下降了1%。

本文中对图片数据进行了水平翻转，随机裁剪两种数据增强以及数据归一化操作。

2.2 微软认知工具集（Microsoft Cognitive Toolkit，CNTK）

CNTK 是一个统一的计算网络框架，它将深层神经网络描述为一系列通过有向图的计算步骤。在有向图中，每个节点代表一个输入值或一个网络参数，每个边表示在其中的一个矩阵运算。CNTK允许用户轻松实现和组合流行的模型类型，如前馈神经网络、卷积神经网络(CNNs)和递归神经网络(RNNs/LSTMs)。CNTK实现了支持跨多个GPU和服务器自动分化和并行化的随机梯度下降（SGD）学习。

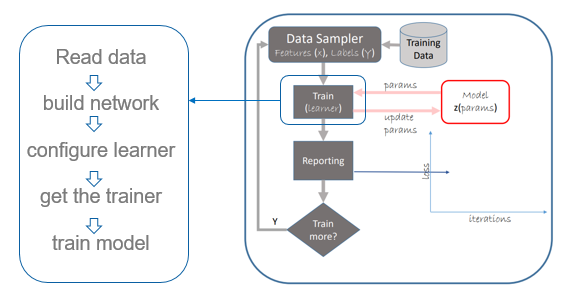


图5 CNTK训练流程图

~~CNTK的输入数据、输出数据和参数全部都用张量表示。每个张量有一个阶数，一个标量是一个0阶张量，一个向量是一个1阶张量，一个矩阵是一个2阶张量等等。我们经常用坐标轴来表示张量的不同维度。~~

~~每个CNTK张量有一些固定维度和一些动态的维度。张量在神经网络运行的整个生命周期在固定维度上的长度保持不变，动态维度在定义时与固定维度类似，但有一些不同：~~

~~其长度可能根据实力的不同而变化~~

~~在训练的取样确定之前，其长度通常是不确定的~~

~~他们可能是按顺序列好的~~

1. 读取数据

* 小数据集（在内存中）

当数据集很小并且可以在内存中加载时，用户可以使用NumPy或在数据中读入NumPy数组或SciPy稀疏(CSR)矩阵来生成自己的数据。

* 大数据（无法载入内存）

使用内置的MinibatchSource类

* + CNTK标准文本格式（CTF）: CTFDeserializer
  + 图片: ImageDeserializer.
  + 声音文件: HTKFeatureDeserializer, HTKMLFDeserializer
* 自定义数据

1. 用于图像分类的一些API

* 卷积层

def Convolution(filter\_shape, # e.g. (3,3)

num\_filters, # e.g. 64

activation, # relu or None...etc.

init, # Random initialization

pad, # True or False

strides) # strides e.g. (1,1)

* 池化层

# Max pooling

def MaxPooling(filter\_shape, # e.g. (3,3)

strides, # (2,2)

pad) # True or False

# Average pooling

def AveragePooling(filter\_shape, # e.g. (3,3)

strides, # (2,2)

pad) # True or False

* Dropout层

# Dropout

def Dropout(prob) # dropout rate e.g. 0.5

* 批量规范化（BN,Batch normalization）

# Batch normalization

def BatchNormalization(map\_rank) # For image map\_rank=1

1. 模型的保存和加载

保存：z.save(**'model\_path’**)

其中z表示最后一层神经网络的输出值，'model\_path‘为模型文件保存路径。

加载：z = C.load\_model(**'model\_path’**)

其中'model\_path‘为模型文件保存路径。

2.3 TensorFlow

TensorFlow是谷歌基于DistBelief进行研发的第二代人工智能学习系统， TensorFlow中的计算可以表示为一个计算图(computation graph)，又称有向图(directed graph)。在计算图中每一个运算操作(operation)将作为一个节点(node)，节点与节点之间的连接称为边(edge)。计算图的每一个节点可以有任意多个输入和输出，节点可以算是运算操作的实例化(instance).在计算图的edge中流动(flow)的数据称为张量(tensor)，故得名tensorflow.

在构造模型阶段，我们需要构建一个图(Graph)来描述我们的模型。所谓图，也可以理解为流程图，就是将数据的输入->中间处理->输出的过程表示出来，就像下面这样。此时是不会发生实际运算的。而在模型构建完毕以后，会进入训练步骤。此时才会有实际的数据输入，梯度计算等操作

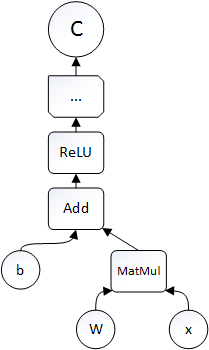


图6 计算图示例

1. 重要概念

图（Graph）：图描述了计算的过程，TensorFlow使用图来表示计算任务。

张量（Tensor）：TensorFlow使用tensor表示数据。每个Tensor是一个类型化的多维数组。

操作（op）：图中的节点被称为op（operation的缩写），一个op获得0个或多个Tensor，执行计算，产生0个或多个Tensor。

会话（Session）：图必须在称之为“会话”的上下文中执行。会话将图的op分发到诸如CPU或GPU之类的设备上执行。

变量（Variable）：运行过程中可以被改变，用于维护状态。

Tensorflow程序通常被组织成一个构建阶段和一个执行阶段。在构建阶段，op的执行步骤被描述成一个图。在执行阶段，使用会话执行图中的op。

构建图的第一步是创建源op（sources op）。源op不需要任何输入，例如常量（Constant）。源op的输出被传递给其他op做运算。

构造阶段完成后，才能在会话中启动图。启动图的第一步是创建一个Session对象。如果没有任何参数，会话构造器将启动默认图。

# 启动默认图.

sess = tf.Session()

# 调用 sess 的 'run()' 方法来执行op,

result = sess.run(product)

# 任务完成, 关闭会话.

sess.close()

Fetch: 为了取回操作的输出内容，在使用Session对象的run()方法执行图时，传入一些tensor，这些tensor会帮你取回结果。

Feed: feed使用一个tensor值临时替换一个操作的输出。可以把feed数据作为参数提供给run()方法。标记的方法是使用tf.placeholder()为这些操作创建占位符。

* ~~Tensor~~

~~Tensor的意思是张量，Tensor的生成方式有很多种，最简单的就如~~

~~a = tf.zeros(shape=[1,2])~~

~~值得注意的是因为在训练开始前，所有的数据都是抽象的概念，也就是说，此时a只是表示这应该是个1\*5的零矩阵，而没有实际赋值，也没有分配空间，只有在训练过程开始后，才能获得a的实际值~~

* ~~Variable~~

~~变量的意思。一般用来表示图中的各计算参数，包括矩阵，向量等。例如，我要表示上图中的模型，那表达式就是~~

~~y=Relu(Wx+b)~~

~~这里W和b是我要用来训练的参数，那么此时这两个值就可以用Variable来表示~~

~~W = tf.Variable(tf.zeros(shape=[1,2]))~~

~~与Tensor不同，Variable必须初始化以后才有具体的值。~~

* ~~Placeholder~~

~~又叫占位符，同样是一个抽象的概念。用于表示输入输出数据的格式。告诉系统：这里有一个值/向量/矩阵，现在我没法给你具体数值，不过我正式运行的时候会补上的！例如上式中的x和y。因为没有具体数值，所以只要指定尺寸即可~~

~~x = tf.placeholder(tf.float32,[1, 5],name='input')~~

~~y = tf.placeholder(tf.float32,[None, 5],name='input')~~

~~上面有两种形式，第一种x，表示输入是一个[1,5]的横向量。~~

~~而第二种形式，表示输入是一个[?,5]的矩阵。~~

* ~~Session~~

~~session，也就是会话。session是抽象模型的实现者。因为模型是抽象的，只有实现了模型以后，才能够得到具体的值。同样，具体的参数训练，预测，甚至变量的实际值查询，都要用到session。~~

1. 读取数据

* 预加载数据

在TensorFlow图中定义常量或变量来保存所有数据(仅适用于数据量比较小的情况)。

* 供给数据(Feeding)

TensorFlow的数据供给机制允许你在TensorFlow运算图中将数据注入到任一张量中。因此，python运算可以把数据直接设置到TensorFlow图中。

* 从文件读取数据

在TensorFlow图的起始， 让一个输入管线从文件中读取数据。

tensorflow可读取的文件类型包括CSV文件、二进制文件、TFRecords文件、图像文件等。 根据读取文件的不同格式，需要选择不同的文件阅读器

1. 模型保存及载入

TensorFlow有多种保存模型的方式，在此介绍以下三种

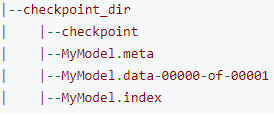
1. 使用tf.train.Saver()保存模型

tensorflow 提供了tf.train.Saver类来保存模型，但在tensorflow中，变量是存在于Session环境中，也就是说，只有在Session环境下才会存有变量值，因此，保存模型时需要传入session：

saver = tf.train.Saver()

saver.save(sess,"./checkpoint\_dir/MyModel")

得到checkpoint\_dir目录下保存的文件结构如下：



MyModel.meta文件保存的是图结构，meta文件是pb（protocol buffer）格式文件，包含变量、op、集合等。

ckpt文件是二进制文件，保存了所有的weights、biases、gradients等变量。

对应的加载模型方法：

with tf.Session() as sess:

#加载图结构

new\_saver = tf.train.import\_meta\_graph('./checkpoint\_dir/MyModel-1000.meta')

#加载参数

new\_saver.restore(sess, tf.train.latest\_checkpoint('./checkpoint\_dir'))

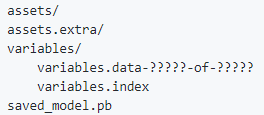
1. SavedModel

SavedModelBuilder提供保存MetaGraphDef结构的功能。MetaGraphDef是MetaGraph的proto buffer表达形式。MetaGraph是一个数据流图，以及相关的变量、资源和signatures。signature是一个graph的输入与输出的集合。每个加入到 SavedModel中的MetaGraphDef需要以用户指定的tag标注。tag提供了区分特定MetaGraphDef的方法。通常这些tag会标注MetaGraphDef的功能以及一些可选的硬件相关的信息。

保存：



得到的文件结构如下：



其中，saved\_model.pb保存图结构，variables目录保存变量等。

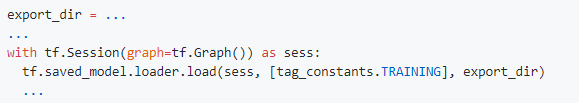
加载SaveModel：

调用python版本的SaveModel loader需要提供一下信息：

保存graph定义和变量的session。

用来标识MetaGraphDef的tag。

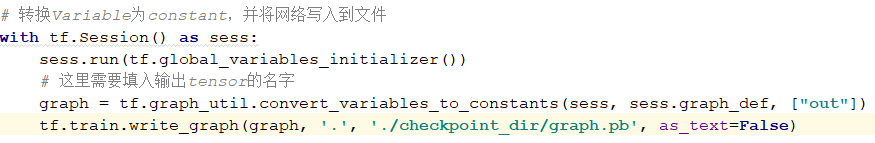
SavedModel对应的目录位置。



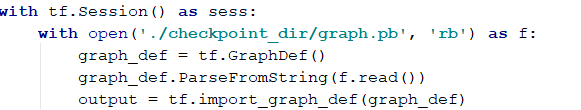
1. Frozen graph

将TensorFlow的模型导出为单个pb文件（同时包含模型架构定义与权重），但利用tf.train.write\_graph()默认情况下只导出了网络的定义（没有权重），因此我们需要先使用tf.graph\_util.convert\_variables\_to\_constants（）将变量转为常量，再保存。

保存：



加载：



本文中使用了saved model和Frozen graph这两种方法

2.4 数据集

**2.1.1 数据集格式**

训练卷积神经网络时，数据集至关重要。在本设计中，选取什么样的数据集，多大，是什么格式，如何选择一个通用的格式使得用户可以简单方便的使用我们的模型去训练他们自己的数据集等都是要考虑的问题。以深度学习视觉领域常用的开源数据集为例：

MNIST：MNIST是一个手写数字数据库，它有60000个训练样本集和10000个测试样本集，每个样本图像的宽高为28\*28。此数据集是以二进制存储的，不能直接以图像格式查看。当前主流深度学习框架几乎无一例外将MNIST数据集的处理作为介绍及入门第一教程，包括CNTK和TensorFlow。

CIFAR：CIFAR-10包含10个类别，50,000个训练图像，彩色图像大小：32x32，10,000个测试图像。有python，matlab和二进制三种格式。

Imagenet：Imagenet数据集有1400多万幅图片，涵盖2万多个类别；其中有超过百万的图片有明确的类别标注和图像中物体位置的标注。Imagenet是以图片形式保存的层级结构：目录->子目录->图片集。

以上三种数据集大小一次递增，格式各不相同，在CNTK和TensorFlow里读取的方式也各不相同。考虑到用户使用本系统的简便性以及数据集大小的不确定性，在训练模型时，本文采用类似Imagenet形式的方式为输入，即要求输入的数据集为以下格式：

<image root folder>

<sub-directory>

<class 1>

<image>

<class 2>

<image>

……

以CIFAR-10为例，首先将其转换成图7所示的层级结构的.png或.jpg格式图片

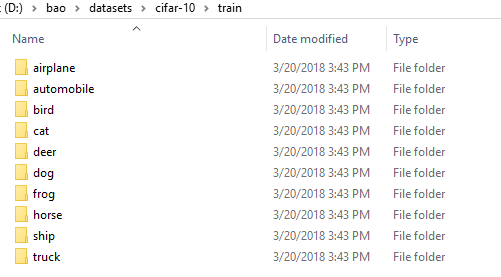


图7 cifar-10层级结构

每一个文件夹下存放同一类别的图片

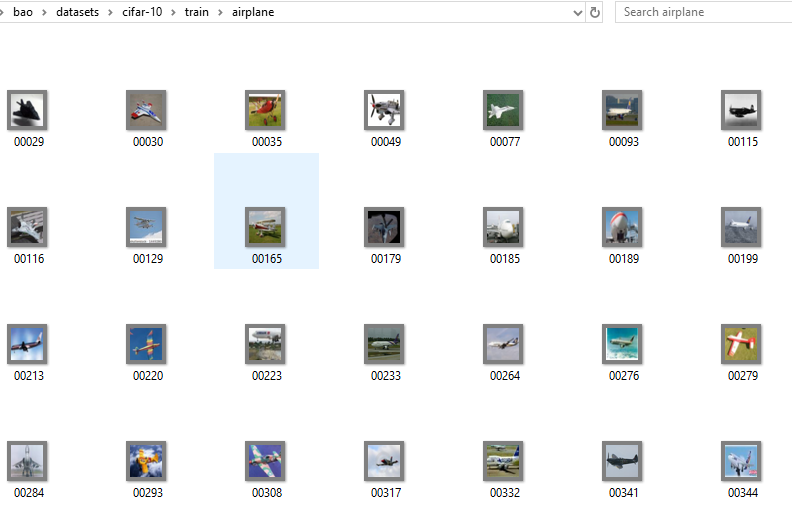


图8

1. 输入CNTK模型

以cifar-10为例，图片集存放在<folder>/cifar-10/train下，将此目录

1. 输入Tensorflow模型

**2.1.1 数据集种类**

本文用于在Python中训练模型并比较结果的数据集如下：

MNIST：将二进制格式转换成2.4.1中的图片格式，结果为大小是28\*28的灰度图，输入数据会被规整成一个三维矩阵（width，Height，1），输出类别为10。训练数据集大小为6000\*10.

Birds：包含六个不同类别鸟类的600张图像（每个100个样本）。图像是可变分辨率的彩色JPEG，输入数据会被规整成三维矩阵（width，Height，3）。6个类别分别是：Egret、Mandarin duck、Snowy owl、Puffin、Toucan、Wood duck。

Cats vs. Dogs（猫狗大战）：训练数据包含猫和狗各12500张图片

Cifar10：包含6万张32\*32的彩色图片，总共有十类，每类有6000张图片。有5万张训练图片和1万张测试图片。

**2.1.1 数据集处理**

1. 将图片转换成CNTK模型接受的格式

本文使用2.2.2中介绍的MinibatchSource类及它的ImageDeserializer方法读取数据。

工业级的训练数据通常都太大，一个取样包内存也装不下。为了应对这种情况，CNTK提供了MinibatchSource类，他提供如下功能：一个随机分块算法，只保存某个时间内存里面的数据；分布式读取，让每个工作的计算机读取不同的数据子集；一个图像和图像增强的转换器；多数据类型整合；异步加载数据以便在数据读取或者准备时运算设备不会等着。

ImageDeserializer从文件中读取图片及其对应标签以以下的格式：

<full path to image> <tab> <numerical label (0-based class id)>

或sequenceId <tab> path <tab> label

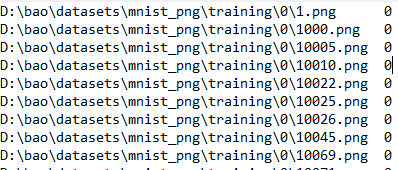
即文件格式应如2.4.3.1所示：

图2 .4.3.1

为了获得这种格式的文本文件，我们对输入的数据集路径做如图2.4.3.2所示处理：

create\_class\_mapping\_from\_folder函数查看根目录下的所有文件夹，并将其名字转换成标签，然后返回一个可以被create\_map\_file\_from\_folder函数用的数组。create\_map\_file\_from\_folder函数浏览整个文件夹，将其路径和标签以索引的方式写入根目录中的map.txt文件。

ImageDeserializer函数接受map.txt文件，从图片的路径中加载进内存，经过随机裁剪，缩放到模型需要的大小，调换色彩通道（RGB→BGR），然后转换成依据图片高度，宽度和色彩通道的连续数组，也就是一个width×height×channel的扁平状态的数组



图2.4.3.2 CNTK数据集处理流程

1. 将图片转换成TensorFlow模型接受的格式

本文采取2.3.1中介绍从文件中加载数据的方法，通过tensorflow特有的tfrecords进行存储和读取数据。

* TFRecords文件

TFRecords 是 Tensorflow 默认的数据格式，TFRecords数据文件是一种将图像数据和标签统一存储的二进制文件，能更好的利用内存，在tensorflow中快速的复制，移动，读取，存储等。

TFRecords文件包含了tf.train.Example 协议缓冲区(protocol buffer，协议缓冲区包含了特征 Features)。将协议缓冲区序列化为一个字符串， 并且通过tf.python\_io.TFRecordWriter class写入到TFRecords文件。

* tfrecords文件的制作

create\_class\_mapping\_from\_folder函数查看根目录下的所有文件夹，并将其名字转换成标签，然后返回一个可以被get\_annotations函数用的数组。get\_annotations浏览整个文件夹，功能与create\_map\_file\_from\_folder函数相似，只不过create\_map\_file\_from\_folder返回一个map.txt文件路径，而get\_annotations返回将图片路径和标签拼合在一起的列表。

convert\_to\_tfrecord函数获取数据， 将数据填入到tf.train.Example协议内存块(protocol buffer)，将协议内存块序列化为一个字符串.

通过tf.python\_io.TFRecordWriter写入到TFRecords文件。

* tfrecords文件的解析和读取

read\_and\_decode函数用tf.train.string\_input\_producer生成一个解析队列，返回serialized\_example对象，之后调用tf.parse\_single\_example操作将Example协议缓冲区(protocol buffer)解析为张量

tf.train.shuffle\_batch函数通过随机抽取产生batches

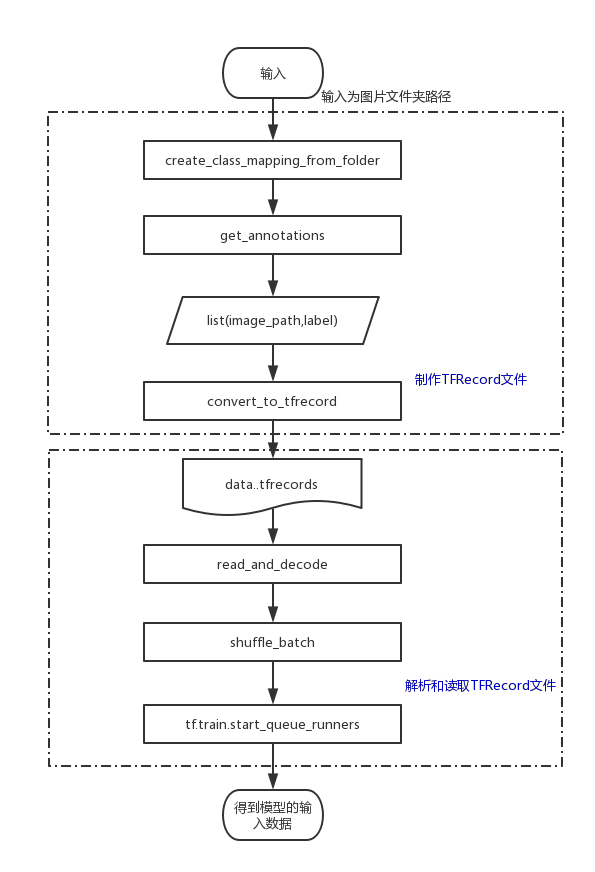
tf.train.start\_queue\_runners创建了线程来进行数据读取

图2.4.3.3

2.5 LabVIEW

LabVIEW 是一种图形化编程环境，通过它能够快速方便的创建具有专业用户界面的应用程序。数以百 万计的工程师和科学家通过 LabVIEW 的直观图标和连线，开发复杂的测量、测试及控制系统应用程 序。此外， LabVIEW 平台可在不同的终端和 OS 之间切换。事实上， LabVIEW 提供了与上千种硬件 设备间的无缝集成及上百种用于高级分析和数据可视化的内置库，用户可使用内置的库创建自定义需 求的虚拟仪器。

LabVIEW 程序称为虚拟仪器 (VI)，它的外观和操作类似于真实的物理仪器 （例如，示波器和万用 表）。 VI 具有前面板和程序框图。用户界面在 LabVIEW 中被称为前面板。程序框图是指在用户界面 后台的编程。前面板创建完毕后，便可使用图形化的函数添加源代码来控制前面板上的对象。程序框 图上的代码为图形化代码，也称为 G 代码或程序框图代码。

与文本编程语言 （C++ 和 Visual Basic）相比， LabVIEW 使用图标代替文本行创建应用程序。在文本 编程中，由指令决定程序的执行次序。 LabVIEW 使用图形化数据流编程。在图形化数据流编程中，由 数据流经程序框图上节点的顺序决定程序的执行次序。图形化编程和数据流执行是 LabVIEW 与其他 通用编程语言的两大主要区别。

2.6 本章小结

第三章 系统设计

本章主要介绍了本课题系统的具体技术实现方法。具体内容包括系统架构概述、相关技术知识介绍以及具体的功能实现方法。

3.1 系统概述

针对谷歌发布的TensorFlow及微软发布的CNTK深度学习框架进行研究，学习使用卷积神经网络进行图像分类。最终在LabVIEW上编写了一套APIs以调用TensorFlow和CNTK完成各项功能， 使LabVIEW用户可以直接在LabVIEW上完成图像分类的训练和预测等任务。

主要内容如下：

1. 基于Python toolkit 及Python node in LabVIEW 2018 方式设计一套LabVIEW APIs，调用 CNTK python编写的以Python函数为单位的接口（一个API调用一个python函数实现一项功能），分别实现图片集分类的训练，预测以及模型文件的保存及读取。
2. 基于Python node in LabVIEW 2018 方式设计一套LabVIEW APIs，调用 TensorFlow python编写的以Python函数为单位的接口（一个API调用一个python函数实现一项功能），分别实现图片集分类的训练，预测以及模型文件的保存及读取。
3. 基于call library node 方式编写一组LabVIEW APIs，调用 TensorFlow C编写的动态链接库，读取（加载，运行和关闭模型文件）TensorFlow模型文件，针对给定的图片进行预测类别。

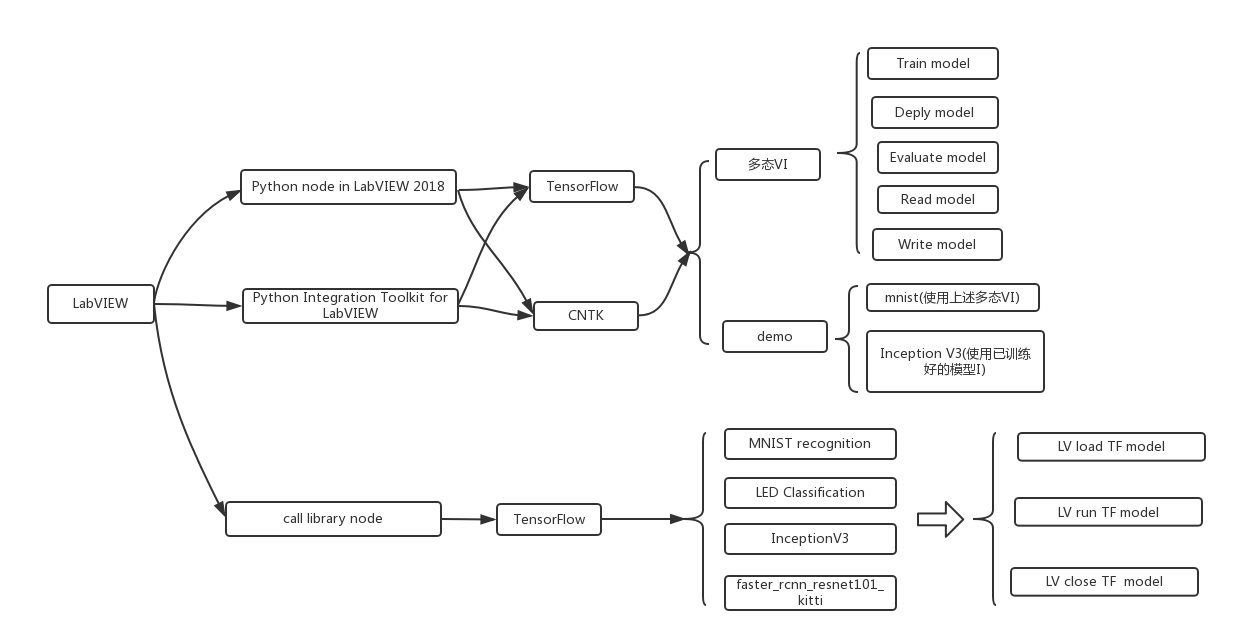


图3.1.1

3.2 Python node in LabVIEW 2018

在LabVIEW中使用Python节点直接调用Python代码。

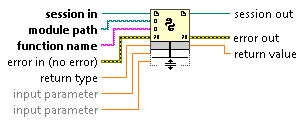
Python node包含三个LabVIEW Vis，分别为Open Python Session，Python Node ，Close Python Session。

Open Python Session：使用特定版本的Python打开Python会话。输入Python版本（2.7或者3.6）输出一个python会话。

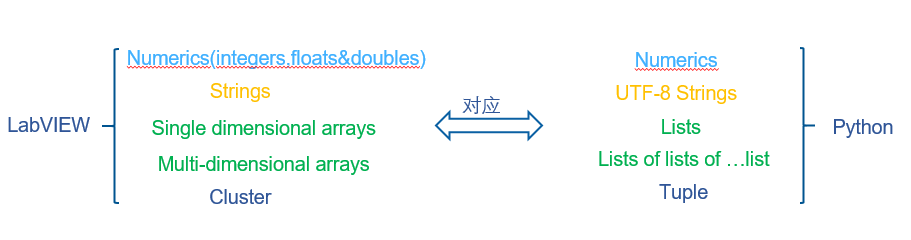


Python Node ：直接调用Python函数。

Python Node是可扩展的，它显示了输入和输出的数据类型。通过配置Python Node来指定Python会话、模块路径和函数名。Python Node不支持实时或FPGA目标。



Python Node支持以下数据类型：数值、数组（包括多维数组）、字符串、簇。其中，与Python对应关系为



Close Python Session：关闭一个Python会话



以Deploy Model为例，如图3.2.1 其实现的原理为：

a). 通过Open Python Session Python找到python dll，记录进程的ID（Session ID），之后每一步都需要进程的ID一步一步的传下去。

例如，如果用户正在运行64位的LabVIEW，并请求python 3.5版本，那么我们只说LoadLibrary('Python35.dll')，Windows通常在标准的搜索目录中搜索以找到64位的Python35。如果找到了dll并加载它。

b).在 Python Node中指定要调用的python文件(CNN\_CNTK.py)和函数名(deploy)，将输入参数(location of model,image)传入python文件中。并提前指定Python函数返回的数据类型。

c).调用函数后，将函数的返回值传回LabVIEW

d).最后结束进程，内存资源释放，重新给系统调配。

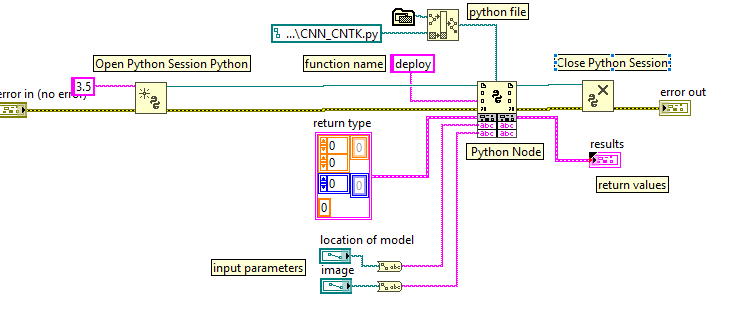
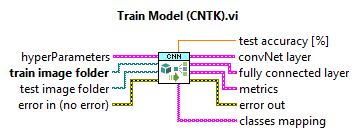


图3.2.1

* + 1. **train model**



train model的流程图如图3.2.2所示

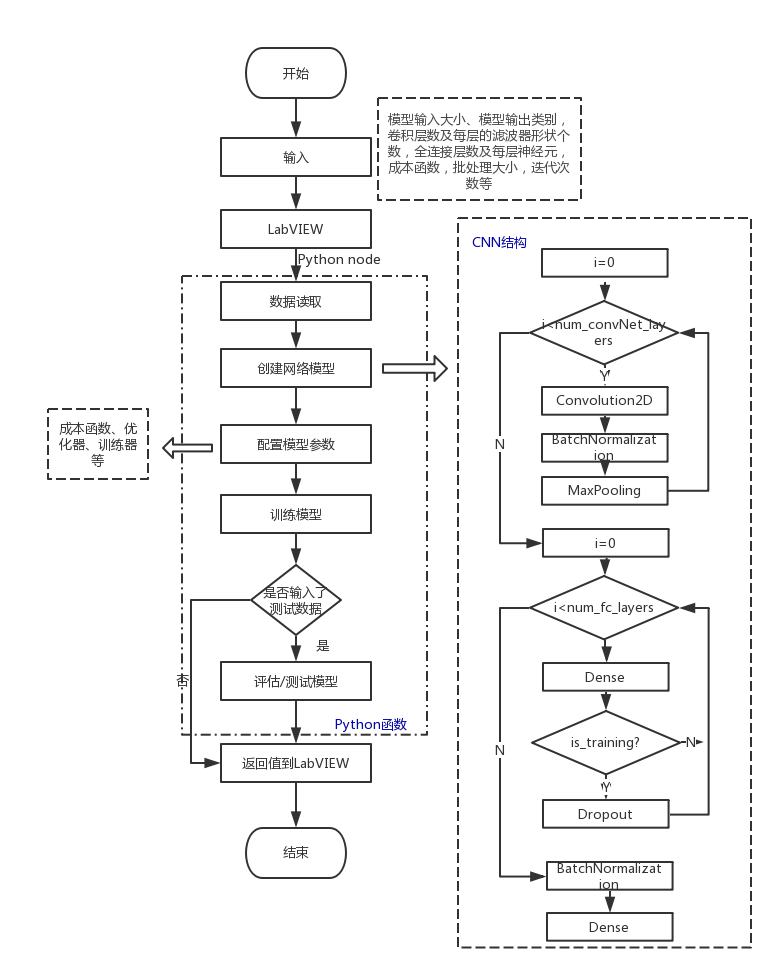
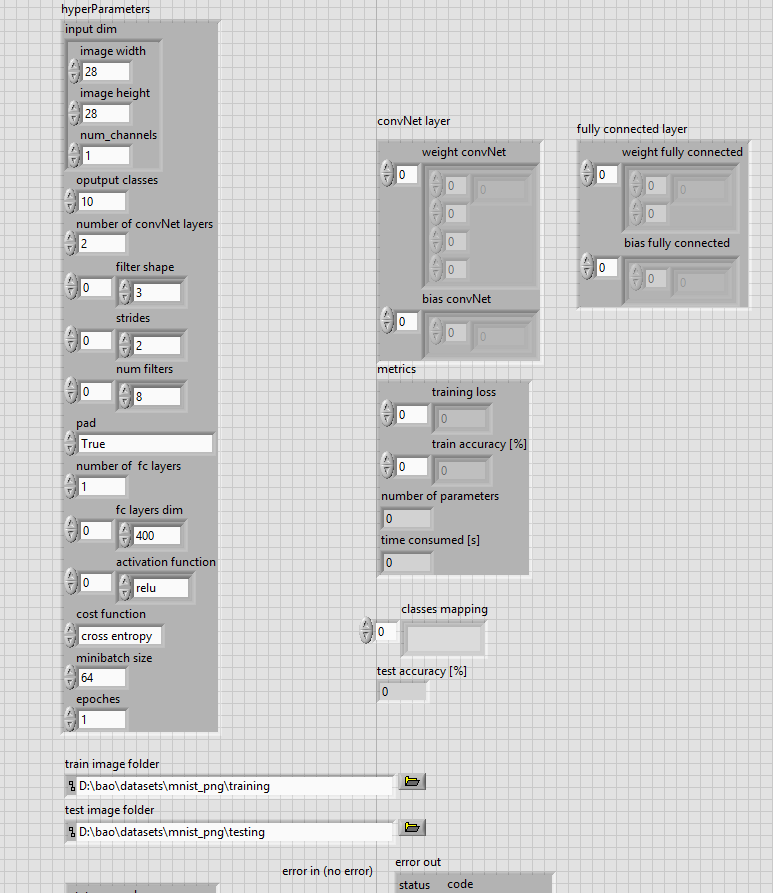


图3.2.2

LabVIEW：用户在LabVIEW前面板中（图3.2.3左部分）输入训练卷积神经网络所需要的参数，如训练数据集路径（train image folder）、测试数据集路径（test image folder，可选）、模型输入大小（input dim）、模型输出类别（oputput classes），卷积层数（number of convNet layers）及每层的滤波器形状个数（filter shape，strides，num filters，pad），全连接层数（number of fc layers）及每层神经元（fc layers dim），激活函数（activation function）、成本函数（cost function），批处理大小（minibatch size），迭代次数（epoches）等，

接着，Python Node调用convnet\_train函数，将用户的输入传入Python函数。

convnet\_train函数包含数据读取、创建网络模型、配置模型参数、评估/测试模型几部分。



3.23

* 数据读取

数据读取如2.4.3及2.4.4中所描述

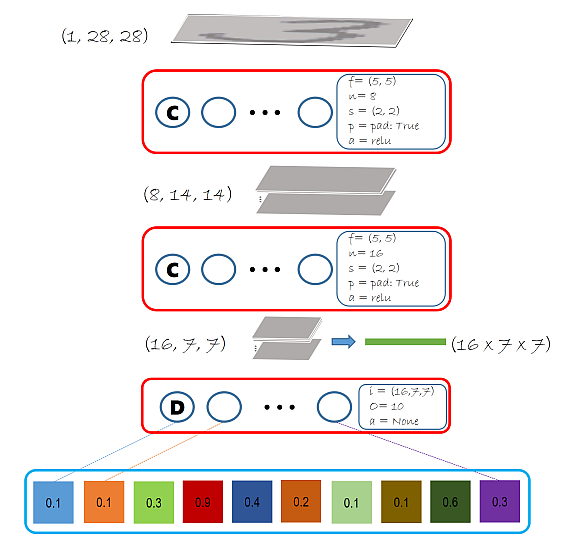
* 卷积神经网络模型创建

如图3.2.2中右侧CNN结构示意图所示，卷积神经网络由m个卷积层，m个批量规范化（BN,Batch normalization）层，m个池化层和n个全连接层，n个舍弃层（Dropout）以及一个输出层组成，其中每层的参数值（滤波器，激活函数等）由输入指定。

1. CNTK

在CNTK中若要使用创建的模型，首先要定义两个变量image\_input和label\_input。image\_input是输入的图像，大小为（image\_channel ，image\_width,image\_height），第二个人是与之对应的lable值，采用one-hot编码，表明了图像对应的数字。input\_map在训练时读取数据使用。

image\_input = C.input\_variable(input\_dim\_model, np.float32,name=**'images'**)  
label\_input = C.input\_variable(num\_classes, name=**'labels'**)  
input\_map = {  
 image\_input: minibatch\_source.streams.features,  
 label\_input: minibatch\_source.streams.labels  
}



以MNIST数据集为例，其输入的图像大小为（1，28，28），第一层。。。

~~了解参数：~~

~~我们的模型有两个卷积层，每个都有权重和偏移量。这样加起来就是4个参数张量，然后再加上最后一个全连接层的权重和偏移量张量，合起来就是6个参数张量。~~

~~然后我们再来细数一下参数：~~

~~\* 第一个卷积层。有8个滤波器，每个的大小是（1×5×5），加起来，权重矩阵就有200个值，8个偏移量。~~

~~\* 第二个人卷积层。有16个滤波器，每个的大小是（8×5×5），其中8代表了第二层的输入数据的通道个数，也是第一层的输出数据的通道个数。这些数据加起，权重矩阵有3200个值，16个偏移量~~

~~\* 最后的全连接层。有16×7×7个输入只，10个输出值，因此有16×7×7×10个权重值和10个偏移量。~~

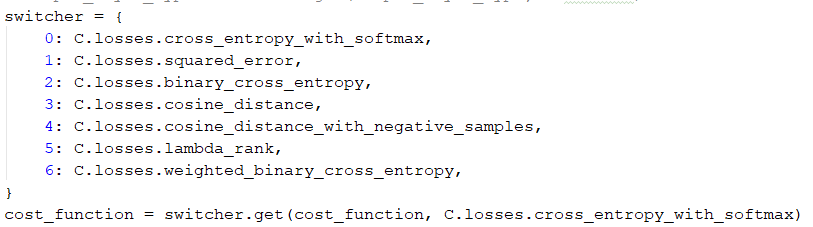
~~这些全部加起来就是11274个参数。~~

* 训练模型参数

我们想方设法使网络的输出值和标签之际的差距减小：

ce = cost\_function(z, label\_input)

其中cost\_function可以为



为了评估我们的分类，我们比较了网络的输出值和对应的标签：  
pe = C.metrics.classification\_error(z, label\_input)

我们使用随机梯度下降算法作为哦我们的优化器，通常我们会随机生成模型参数，随机梯度下降算法会计算成本函数以及运算结果和标签之间大差值，使用梯度下降的方法来给模型生成一个新的参数，进入下一个迭代。

learner = C.learners.momentum\_sgd(z.parameters, lr\_schedule, mm\_schedule)

选择一小段数据集，使用他们的平均成本函数值和插值来更新模型参数，这个数据集叫取样集（minibatch），使用取样集我们可以简化大的训练数据集，我们一遍又一遍的使用不同的取样集来更新模型参数，以此来减少成本函数值，当误差率不再明显变化或者说我们设置好的训练轮数到了，这个模型就训练好了。

* 评估/测试模型

若LabVIEW中指定了测试数据，则使用测试数据来评估一下我们的训练结果，这个工作用trainer.test\_minibatch完成。

最后将convnet\_train函数的到的结果返回到LabVIEW，其中包括卷积层的权重矩阵和偏置值，全连接层的权重矩阵和偏置值，每个训练迭代的训练损失和正确率，训练参数总数，训练时间，类别映射，测试准确率（weight\_convLayer, weight\_fcLayer, bias\_convLayer, bias\_fcLayer, training\_loss, eval\_accuracy,

float(num\_parameter), train\_time, list(class\_mapping),test\_acc）。

其中权重矩阵和偏置值需要转换成Python Node能接受的数据类型。以weight\_convLayer为例，weight\_convLayer是一个五维数组，每一维里又是大小不一的四位数组，要将其转换成lists(tuple(lists(lists(lists(lists(float))))))数据类型，对应的LabVIEW数据类型为array(cluster(array(array(array(array(float)))))) 数据类型。

1. TensorFlow

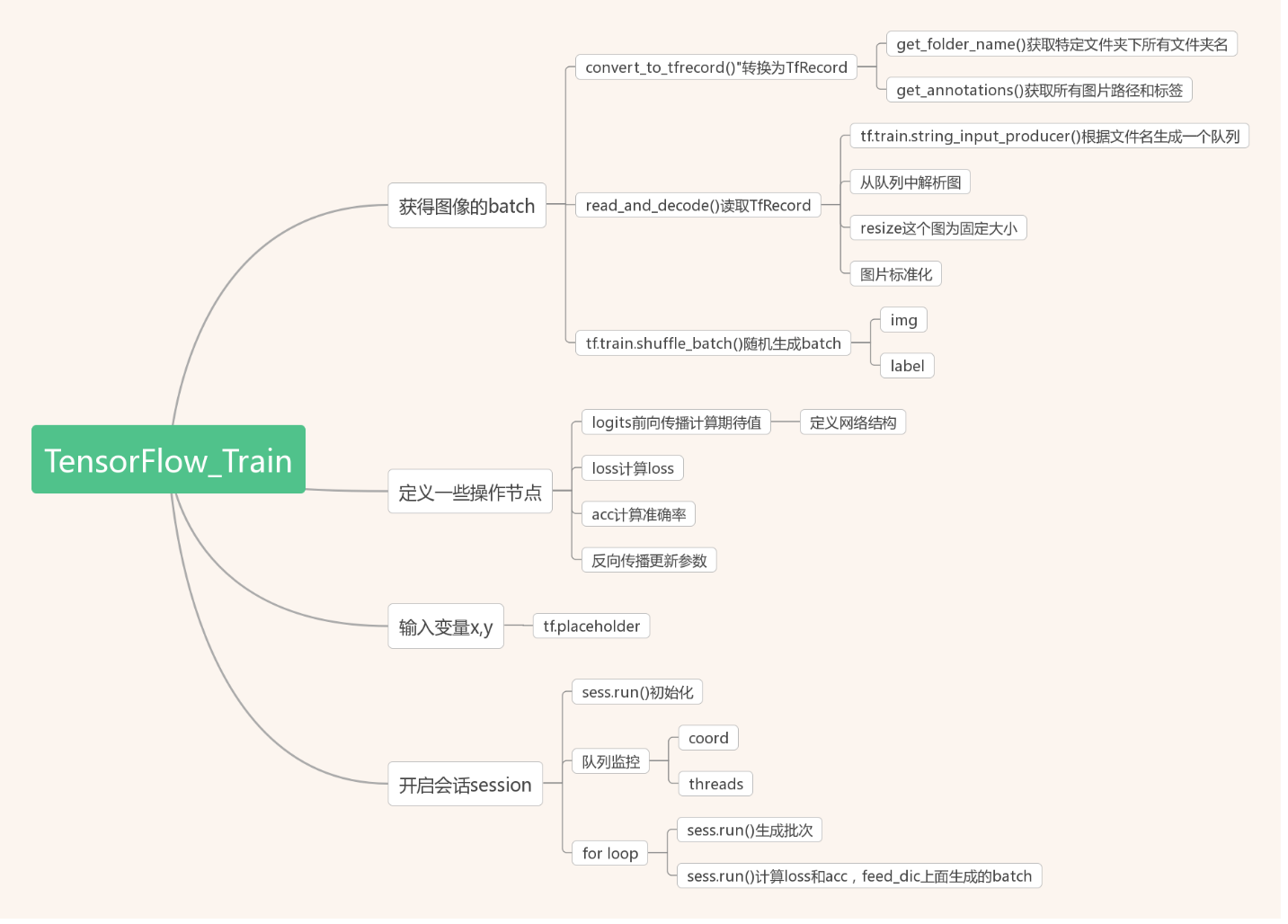


图3.2.4

TensorFlow的训练过程如图3.2.4的思维导图所示，在读取TFrecords文件中保存的img和label之后，使用tf.train.shuffle\_batch随机从训练数据中随机抽出batch\_size个数据样本。

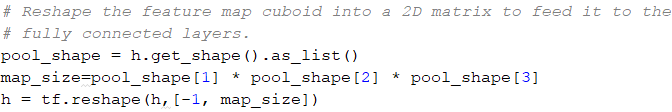
在TensorFlow中若要使用创建的模型，也首先要定义两个变量x和y\_。x是输入的图像，大小为（None,image\_width,image\_height ，image\_channel ），第二个人是与之对应的lable值，采用one-hot编码，表明了图像对应的数字。

x = tf.placeholder(tf.float32, shape=[**None**, image\_width, image\_height, num\_channels], name=**'x'**)  
y\_ = tf.placeholder(tf.int32, shape=[**None**,num\_classes], name=**'y\_'**)

placeholder()是用来来传递一个tensor到session.run()中

**with** tf.Session() **as** sess:

feed={x: image\_batch\_1, y\_: label\_batch\_1}  
\_, err, ac= sess.run([optimizer, loss, acc], feed\_dict=feed)

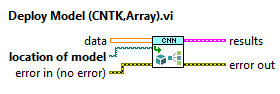
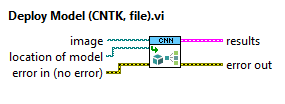
TensorFlow的网络结构中与CNTK不同的一点在：TensorFlow中卷积层后加全连接层时要手动将卷积层的输出reshape为全连接层可接受的输入形状

这也导致在输入参数相同的情况下，

Tensorflow与CNTK训练的权重矩阵和偏置项的大小会所差别，这个后文会举例加以说明。

TensorFlow中训练模型参数，评估/测试模型的思想与CNTK相同，只是实现的API不同，在此不再赘述。

* + 1. **deploy model**



LabVIEW：输入为图片及模型文件路径，返回为每张图片的类别概率分布以及预测的最有可能的类别。其中输入图片可以为单张图片路径或者多张图片文件夹或者四位数组数据。

Python：加载模型文件，获取网络模型的输入大小（image\_channel, image\_width, image\_height）,判断输入的待预测图片，若是图片路径（此时为单张图片），则打开此图片，调整大小到模型要求的大小，即（image\_channel, image\_width, image\_height），得到像素点，添加进待预测列表；若为文件夹路径，则有多张图片，分别打开图片，调整大小到模型要求的大小，得到像素点，添加进待预测列表；若已经为像素数据，则直接添加进列表。得到待预测列表后，使用模型进行预测，得到每个样本对应分类的概率分布及最大可能类别，返回到LabVIEW。

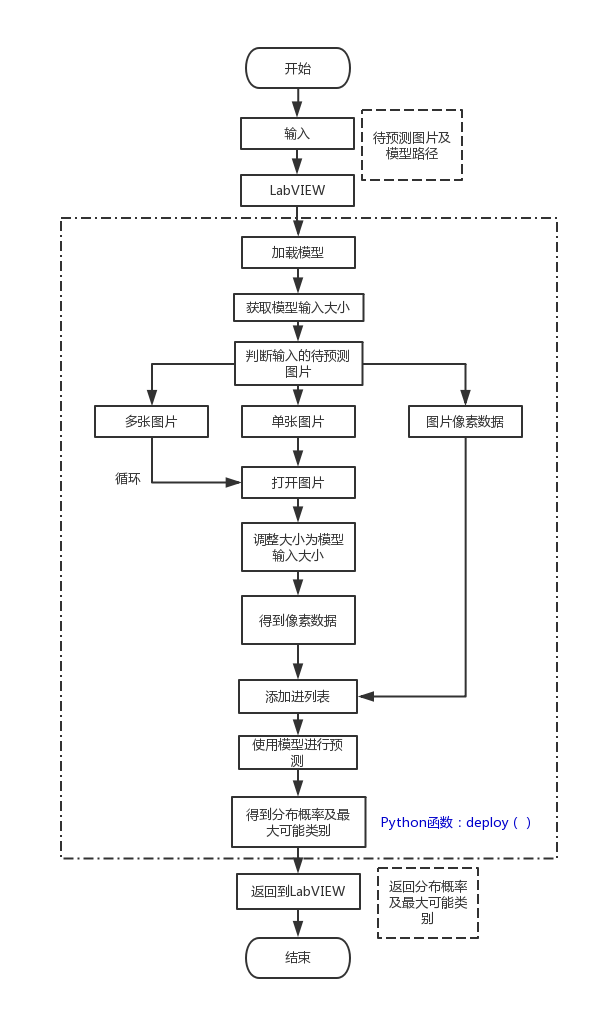
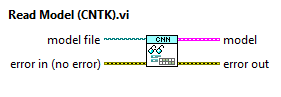


图3.2.2.1

**3.2.3 read model**



一个训练好的卷积神经网络模型包括模型输入大小（input dim）、模型输出类别（oputput classes），卷积层数（number of convNet layers）及每层的滤波器形状个数（filter shape，strides，num filters，pad），全连接层数（number of fc layers）及每层神经元（fc layers dim），激活函数（activation function），卷积层权重矩阵（weight\_convLayer）和偏置值（bias\_convLayer），全连接层权重矩阵（weight\_fcLayer）和偏置值（bias\_fcLayer）。Read model模块的目的就是从一个已训练好的模型文件中读出以上数据。

1. CNTK

在CNTK中直接能读取的只有权重矩阵和偏置值，

parameters = loaded\_model.parameters

for parameter in parameters:

print(parameter.name, parameter.shape, "\n", parameter.value)

其中 parameters是网络结构里所有层的权重矩阵和偏置值，parameter.name为‘W‘(权重名)或者’b’（偏置名），parameter.shape为权重和偏置的维度，parameter.value为权重和偏置的值。

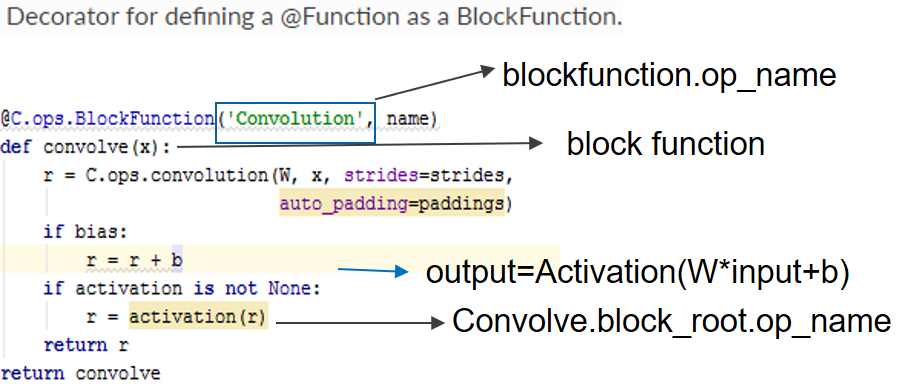
CNTK函数结构：

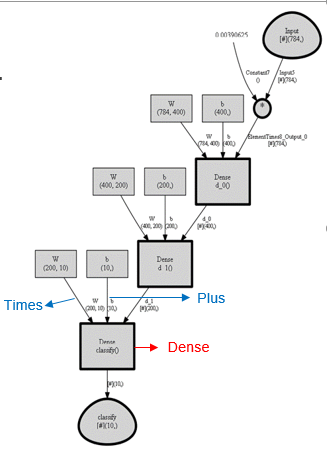
CNTK具有原始函数（primitive functions）、块函数（block functions）和复合函数（composite functio）。

原始函数是图中的单个操作（比如Times, Plus等操作.)。

复合函数是一个完整的子图，它具有一个作为root\_function的原始操作，然后扩展到包含所有它的依赖项。（宏观上是整个计算图）

块函数是一个封装机制，一般来说一个网络层是一个块函数（比如Convolution, Dense等）。





节点（Node）: 泛型类，可以是原始函数，也可以是块函数。

函数属性：

arguments：不属于类型参数或常量的函数的所有输入变量的列表。

attributes：函数的属性列表。

block\_root：返回块函数下的函数图的根。如果不是块函数，则抛出异常。

Inputs：函数的输入变量的列表。这里的“Inputs”表示输入函数的所有变量，包括这个函数的子元素的任何参数/常量变量。

name：函数的名称

op\_name：此函数执行的操作的名称

output：单个输出变量如果函数只有一个输出变量，否则引发异常

outputs：该函数包含的所有输出变量的列表

parameters：该函数的所有参数变量的列表

root\_function：该函数的函数图的根的原始函数

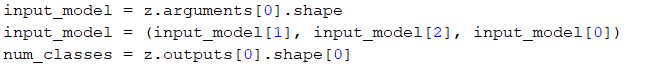
uid：该函数的内部生成的惟一名称。

**depth\_first\_search**(root, visitor, depth**=**0)：

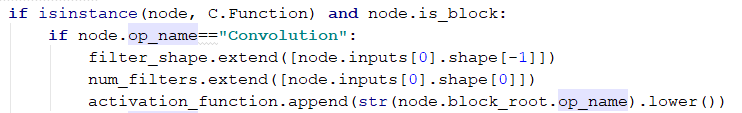
一个模型是一个计算图，depth\_first\_search函数从“根”（最后一个输出）开始检索，遍历图中的节点，在每个节点上使用“visitor”功能来检查它是否应该被返回。

visitor (Python function or lambda): 以节点为参数的函数， 如果该节点需要被返回则visitor 函数返回``True``。

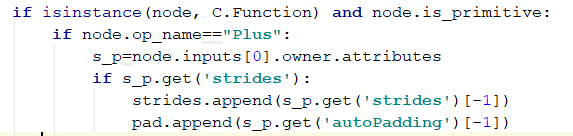
* 获取模型的输入输出



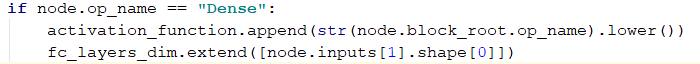
* 获取卷积层的滤波器大小（filter\_shape）及个数（num\_filters）以及卷积层的激活函数（activation\_function）



* 获取卷积层的步幅参数（strides）和边界填充参数（pad）



* 获取全连接层的神经元个数（fc\_layers\_dim）和全连接层的激活函数（activation\_function）



* 获取卷积层和全连接层的权重矩阵和偏置值并将其转换成可传入Python Node的数据格式



1. TensorFlow

GraphDef

TensorFlow计算的基础是图对象。它包含一个节点网络，每个节点表示一个操作，并将其作为输入和输出连接在一起。在创建了一个图形对象之后，可以通过调用as\_graph\_def()来保存它，它返回一个GraphDef对象。

GraphDef类是一个对象由ProtoBuf库创建，protobuf工具解析这个文本文件，并生成用于加载、存储和操作图形的代码。如果一个表示模型的TensorFlow文件是单独的，它很可能包含一个由probuf代码保存的这些GraphDef对象的序列化版本。

Nodes

一旦将文件加载到graph\_def变量中，就可以访问它内部的数据了。最重要的部分是节点成员中存储的节点列表。

nodes=graph.node

其中，graph是从模型文件中创建的图

每个node是一个NodeDef对象, 在[tensorflow/core/framework/node\_def.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/node_def.proto)中定义。这些是TensorFlow图形的基本构建块，每一个都定义了一个操作以及它的输入连接。

NodeDef的有以下属性：

name：每个node都有一个唯一的标识符，该标识符不会被图中的任何其他节点使用。如果在构建图形时没有指定一个name，那么TensorFlow会自己选择一个反映操作名称的函数，例如“MatMul”，并将其与一个单调递增的数字(如“5”)连接起来。在定义节点之间的连接时，以及在运行时为整个图设置输入和输出时使用该名称。

op：这定义了运行的操作，例如"Add"、“MatMul”或“Conv2D”。当一个图运行时，这个op名称将在注册表中查找它的实现。

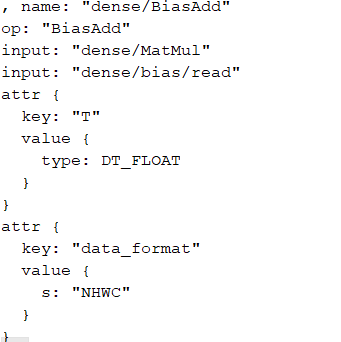
input：字符串列表，每个字符串都是另一个节点的名称，有的会后跟冒号和输出端口号。例如，有两个输入的节点可能有一个列表，比如["some\_node\_name"， "another\_node\_name"]，它相当于["some\_node\_name:0"， "another\_node\_name:0"]，并将节点的第一个输入定义为“some\_node\_name”节点的第一个输出，第二个输入来自“another\_node\_name”的第一个输出

attr：保存节点所有属性的键/值。attr是节点的永久属性，在运行时不会改变的东西，比如卷积的过滤器的大小，或者常数项的值。属性值有很多不同类型,比如字符串,整数,张量数组值等。

每个属性都有一个惟一的名称字符串，在定义操作时将列出预期的属性。如果一个属性没有出现在一个节点中，但是它在操作定义中有一个默认值，那么在创建图形时使用默认值。

通过调用node.name、node.op 等等来访问所有这些成员。在Python中。存储在GraphDef中的节点列表是模型体系结构的完整定义。

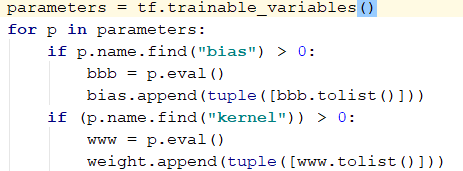
我们取出由frozengraph.pb模型文件创建的图中的一个node来看，如下图



name是这个操作的标识符，为"dense/BiasAdd"，可得这个操作为全连接层中加上偏置项，它由两个输入，一个是该全连接层上一个op也就是点积运算的输出，另一个是偏置项值的读取。

权重矩阵和偏置项：

在train model中权重矩阵和偏置项可以通过tf.trainable\_variables()中来读取，

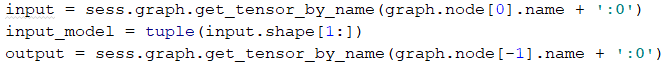


但在read model中，从frozengraph.pb模型文件中读取的图，权重矩阵和偏置项是作为常量存在的，它们保存在Const 操作中。在Python中，通过调用诸如some\_node\_def.attr['value'].tensor，我们可以从一个表示Const op的节点中得到一个TensorProto对象。

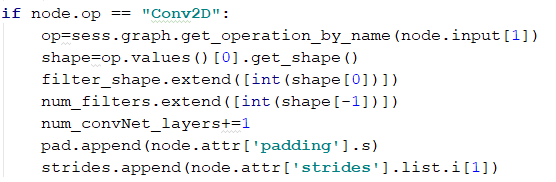
这将返回一个表示权重数据的对象。数据本身将被存储在一个带有后缀\_val的列表中，该列表由对象的类型表示，例如32位浮点数据类型的float\_val。

当在不同的框架之间转换时，卷积权重值的排序通常不同。在TensorFlow中，对Conv2D操作的过滤器权重存储在第二个输入中，其大小为[filter\_height, filter\_width, input\_depth, output\_depth]中

* 获取模型的输入输出



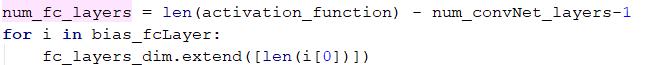
* 获取卷积层的滤波器大小（filter\_shape）及个数（num\_filters）以及步幅参数（strides）和边界填充参数（pad）



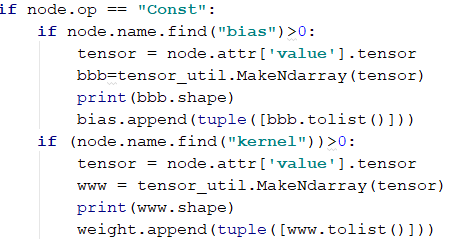
* 获取激活函数（activation\_function）



* 获取全连接层的神经元个数（fc\_layers\_dim）

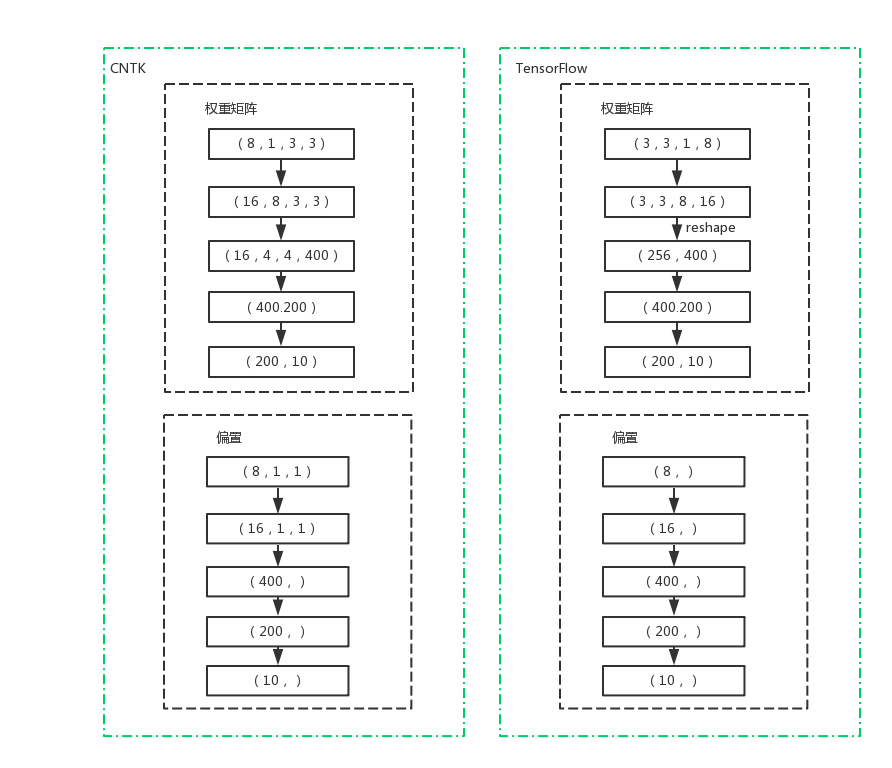


* 获取卷积层和全连接层的权重矩阵和偏置值并将其转换成可传入Python Node的数据格式

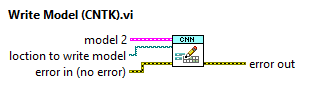


CNTK与TensorFlow权重矩阵和偏置值的不同：

使模型输入大小（input dim）=（28，28，1）、模型输出类别（oputput classes）=10，卷积层数（number of convNet layers）=2及每层的滤波器形状个数（filter shape=[(3,3）,(3,3)]，strides=[(2,2）,(1,1)]，num filters=[8,16]，pad=True），全连接层数（number of fc layers）=2及每层神经元（fc layers dim）=[400,200]

则

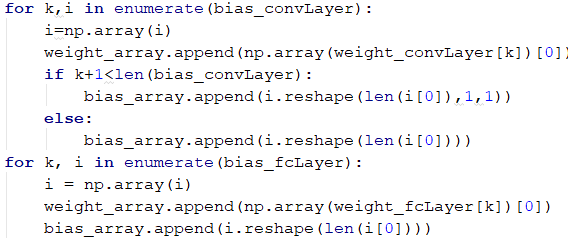
**3.2.4 Write Model**



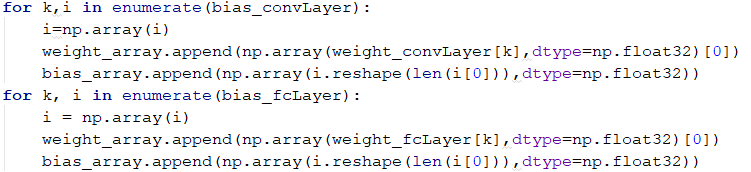
Write model模块的目的就是根据Read Model可读出来的数据不需要训练生成一个可用于deploy的模型。因此传入的参数需要包括模型输入大小（input dim）、模型输出类别（oputput classes），卷积层数（number of convNet layers）及每层的滤波器形状个数（filter shape，strides，num filters，pad），全连接层数（number of fc layers）及每层神经元（fc layers dim），激活函数（activation function），卷积层权重矩阵（weight\_convLayer）和偏置值（bias\_convLayer），全连接层权重矩阵（weight\_fcLayer）和偏置值（bias\_fcLayer）。

在传入初始值进网络结构中前，要将其转换成CNTK和TensorFlow可接受的数据形式（比如array）和大小

CNTK：



TensorFlow：



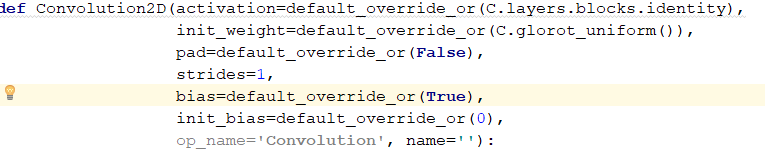
从这里又可看出CNTK与TensorFlow权重矩阵和偏置项的形状不同

1. CNTK

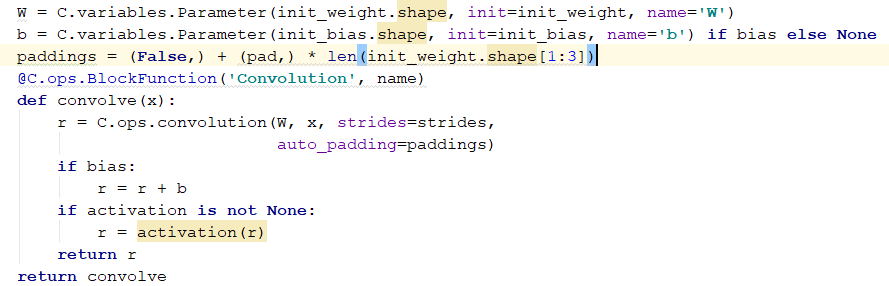
由于CNTK本身的bug，原本的网络层API(CNTK.layers.Convolution2D)接受数组类型的初始化权重矩阵和偏置值会产生错误，因此，我们需要使用CNTK的底层API (C.variables.Parameter)来构建可初始化权重矩阵和偏置值的卷积层和全连接层。

MyFunction.Convolution2D():

函数原型：

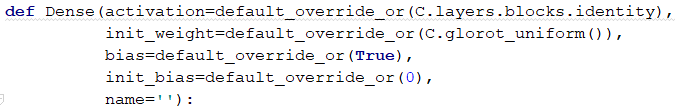


具体实现：

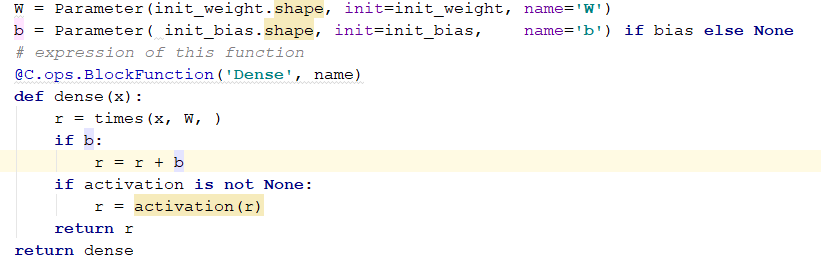


MyFunction.Dense():

函数原型:



具体实现：

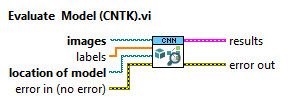


BN层和池化层可直接使用CNTK的网络层API而不用我们重新实现。

1. TensorFlow

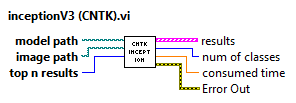
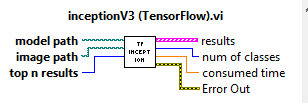
使用TensorFlow网络层的APIs直接构造网络结构即可。

**3.2.5 Evaluate model**



Evaluate model并没有调用python，而是在Deploy model的基础上传入labels与deploy model返回的预测值对比得到准确率。

**3.2.6 Inception V3 demo**

InceptionV3.vi根据已经训练好的Inception V3模型文件InceptionV3\_ImageNet\_CNTK.model和classify\_image\_graph\_def.pb预测任意一张图片的top-n类别



3.3 Python Integration Toolkit for LabVIEW

Python Integration Toolkit 是Python和LabVIEW之间的桥梁，共有11个子VI，其中本设计只用到了其中的5个子VI，分别是New Session，

* New Session：启动新的Python会话（session）。

为磁盘上的.py文件提供路径。这个模块中声明的变量和函数将从LabVIEW中获得。这个VI返回一个会话标识符（session ID）。将标识符传递给其他工具包中的VI比如Pack、Call Value或者 Get Value。完成后，用该标识符调用Close会话，以关闭会话并关闭Python。



* Close Session：结束会话并关闭Python。

和LabVIEW中的其他“清理”一样，这个VI将错误传递到Error out。一个无效的或已经关闭的会话ID将被完全忽略。



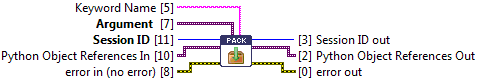
* Call: 调用Python函数、方法或可调用对象。

使用Pack收集参数以发送到Python函数，并将它们提供给终端中的Python对象引用(Python Object References)。函数返回的任何值都将在Python对象引用终端上提供。使用Unpack将这些数据转换为LabVIEW值。



* Pack:准备在Python函数调用中使用的对象。

在使用Call调用Python函数之前，对该函数的任何参数都必须“packed”，或者以Python能够理解的形式收集。

调用Pack.vi一次传入一个参数，将Python对象引用（**Python Object References）**终端连接到一起，并将其连接到Call上相应的终端。

* Unpack ：检索Python函数调用的返回值，并转换为LabVIEW对象。

在用Call调用了一个函数之后，检索(或“unpack”)任何由该调用返回的Python对象。将Python对象从Call发送到Unpack.vi。如果多个对象从调用中返回，则连接多个Unpack实例。

这是一个“多态”VI;它有一个下拉菜单，允许选择通过值终端返回的数据类型。选择与所返回的Python对象类型相对应的类型(string、int等)。

基本类型包括整数（integers）、浮点数（floats）、字符串（strings）和一维数组（1D arrays）。对于其他类型(例如2D数组)，或者不能提前知道类型的情况，选择“Variant”，并使用LabVIEW的*Variant to Data*函数。



此处以deploy model为例，Python Integration Toolkit for LabVIEW实现的原理大概如下，

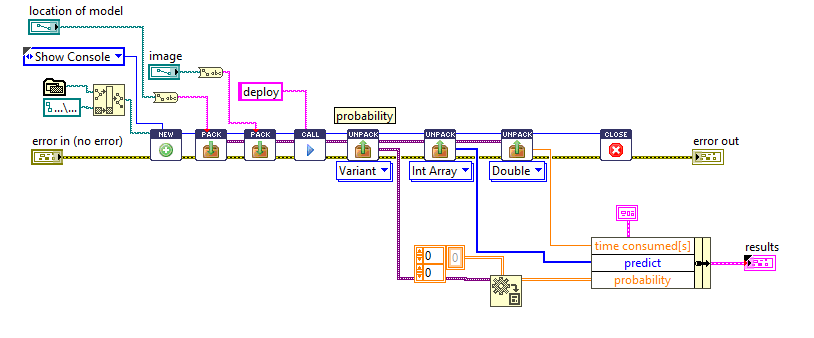
a) 通过python.exe调用文件，并记录进程的ID（Session ID），之后每一步都需要进程的ID一步一步的传下去。

b) 将需要传入的参数打包，捆绑在一起，成为一个独立的对象。

c) 通过调用函数的方法，将捆绑的参数传入python文件中，python文件只有提供调用的函数。

d) 调用函数后，将函数的返回值传回LabVIEW，如果有多个返回值，返回的就和传入参数的时候类似，是捆绑在一起的返回值，需要一个一个地进行解析。

e) 最后结束进程，内存资源释放，重新给系统调配。



**3.3.1 Train model**

流程与3.2.1相同，只是从LabVIEW传入参数和返回值到LabVIEW中不同。

1），在传入数据时，要将LabVIEW中所有的cluster转换成Array，因为通过Python Integration Toolkit不能处理cluster数据类型。

2）将string类型的数据传入python时要在Python里将bytes类型转换成str类型

3）返回权重矩阵和偏置值时，在python中将其reshape为一维数组，同时将原始形状的二维数组一起传回LabVIEW，LabVIEW中在通过传回来的原始形状的二维数组将权重矩阵和偏置值reshape。如图3.3.1.1

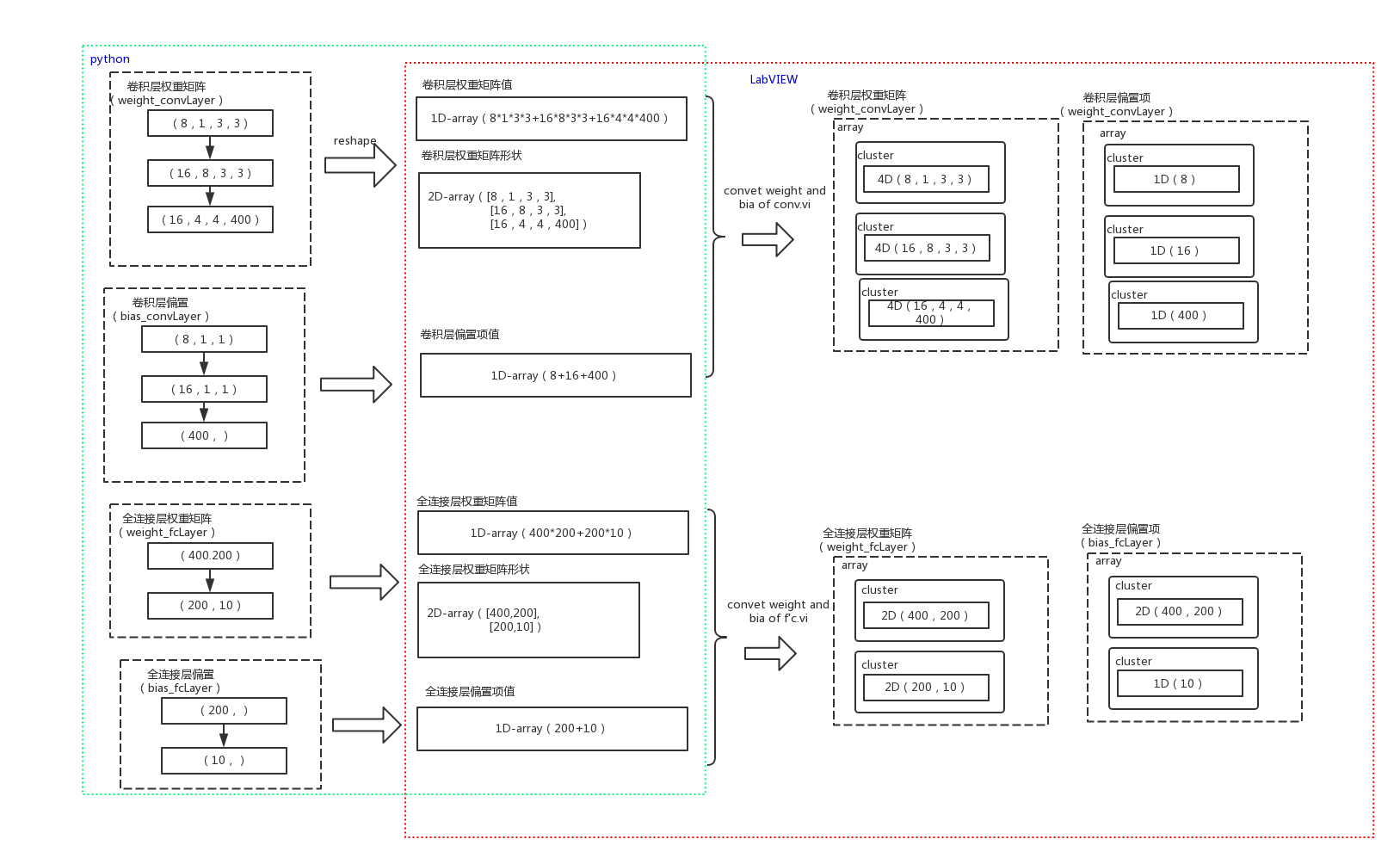


图3.3.1.1

在训练模型时，使模型输入大小（input dim）=（28，28，1）、模型输出类别（oputput classes）=10，卷积层数（number of convNet layers）=2及每层的滤波器形状个数（filter shape=[(3,3）,  
(3,3)]，strides=[(2,2）,(1,1)]，num filters=[8,16]，pad=True），全连接层数（number of fc layers）=2及每层神经元（fc layers dim）=[400,200]，得到的权重矩阵和偏置项的大小如图3.3.1.1

左侧所示，

* 首先，在Python中，以卷积层权重矩阵为例，取出权重矩阵，为三个四维数组，其大小分别为（8，1，3，3）、（16，8，3，3）、（16，4，4，400），将其reshape成一个一维数组，并将原始大小保存在二维数组中2D-array（[8，1，3，3], [16，8，3，3], [16，4，4，400]）。
* 通过python toolkit传回LabVIEW

在LabVIEW中，“Unpack”返回值，如图3.3.1.2，其中通过原始数据以及原始形状恢复如图图3.3.1.3 和图3.3.1.4.由于LabVIEW要求多维数组中同一维的大小必须相同，因此在每个四维数组外加一层cluster再组成权重矩阵。

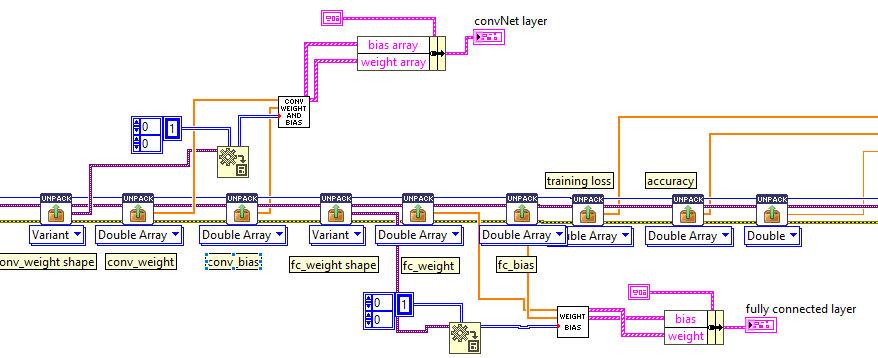


图3.3.1.2

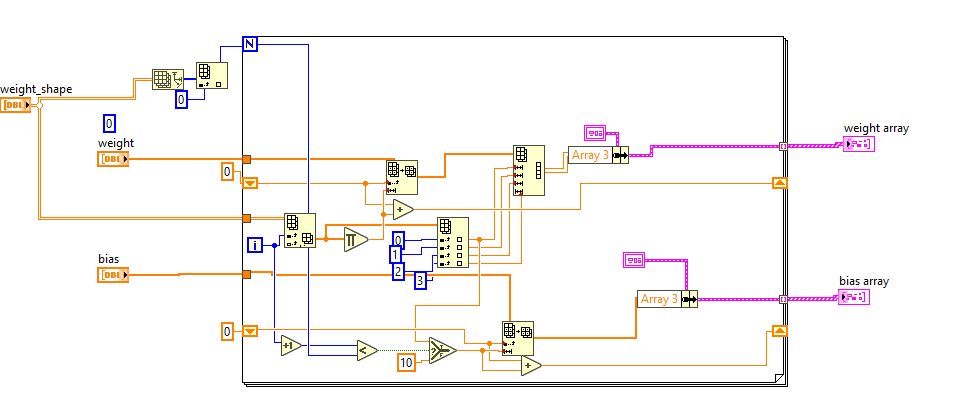


图3.3.1.3

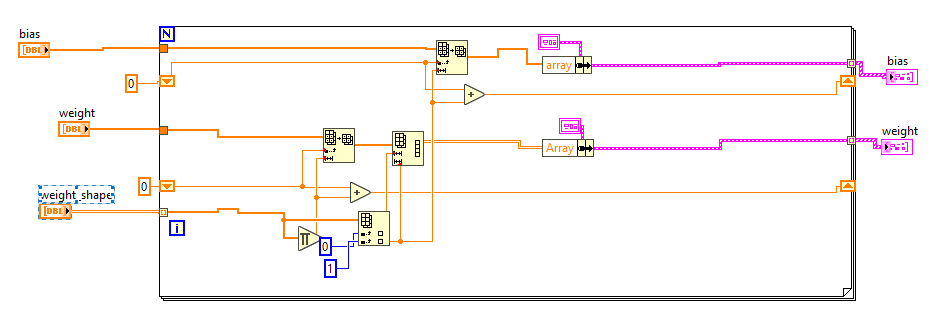


图3.3.1.4

**3.3.2 Deploy model**

与Python Node基本无区别

**3.3.2 read model**

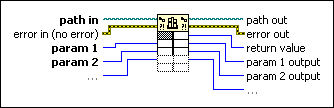
与3.3.1中差别相同

**3.3.2 Write model**

与3.3.1中对权重矩阵和偏置项处理相反，在LaVIEW中将权重矩阵和偏置项reshape为一维数组传入Python，在Python中恢复原始形状。

3.4 Call Library Function Node

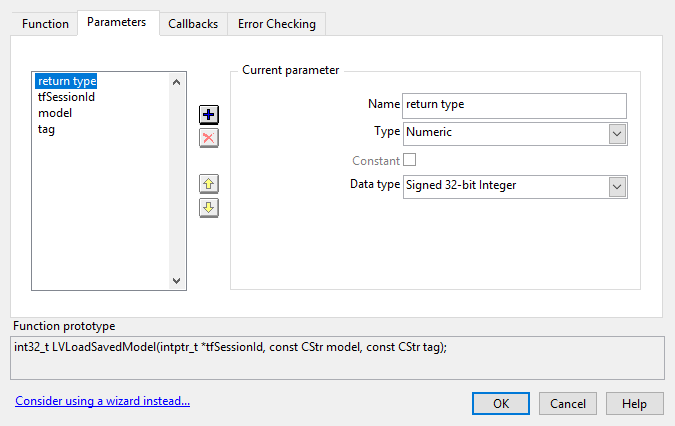
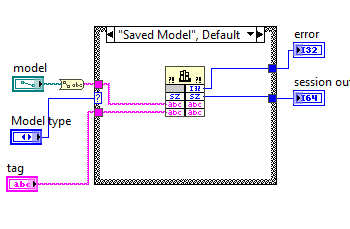
Call Library Function Node可以调用一个DLL或者共享库函数。



该函数传递给C的参数类型可以为Numeric 、Array 、String 、Waveform 、Digital Waveform 、Digital Data 、ActiveX 、Adapt to Type 、Instance Data Pointer。接受C的返回类型可以为Void, Numeric, or String 中的一种。

以LV Load TF Model 为例，在Call Library Function Node的Configure中配置相关参数，首先在

Function中指明共享库（.dll或者.so）的路径和要调用的函数名称，其次在Parameters中配置要输入的参数以及返回值，由图中可以看出，LV Load TF Model接受三个输入参数，分别为tfSessionId，model和tag，返回值的类型是numeric。



与Python相比，TensowFlow C API提供了另一种接口。它虽然不支持所有的TensorFlow特性，但是在C中运行预定义的图形已经足够。

此部分设计使用了TensorFlow C编译出我们需要的共享库，再利用该共享库读取（加载，运行和关闭模型文件）训练好的的TensorFlow模型文件并再LabVIEW里预测类别。

这里列出了我们在TensorFlow C AP上做过的事情：

在Win-64 和 Linux X64上构建一个共享库(ni\_tensorflow)来发布必要的C API,

为了运行一个预定义的图结构,主要有3个步骤:

TF\_LoadSessionFromSavedModel/TF\_LoadSessionFromFrozenGraph 创建一个会话（session）

TF\_SessionRun 运行该会话

TF\_CloseSession to 终止该会话

示例代码被创建为调用ni\_tensorflow共享库来加载和运行预定义的图形，它可以是frozen graph或SavedModel。。在LabVIEW 64位中打开LV项目，我们可以看到不同模型的3个最高级别:LED分类、MNIST和InceptionV3。这些VIs都调用lvtfloader.dll。lvtfloader.dll是一个包装器，可以在TensorFlow数据类型和LV数据类型之间进行转换。按照示例代码，您还可以使用您自己的程序加载和运行经过训练的模型。

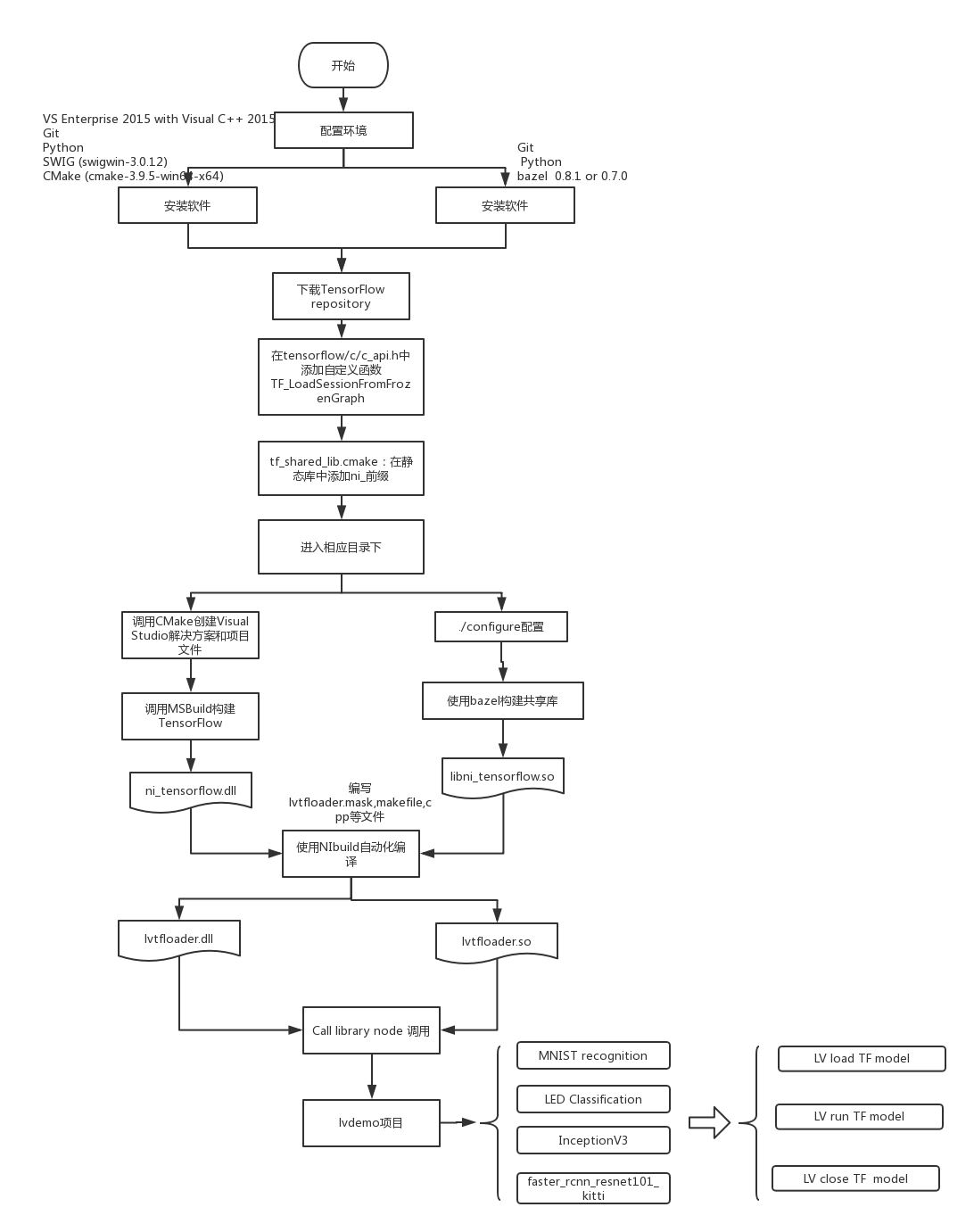


图3.4.1

此部分设计流程如图3.4.1，为了能够运行VI，我们首先需要编译出它所需要的共享库（Windows下的lvtfloader.dll和Linux下的liblvtfloader.so），而该共享库又依赖于tensorflow c\_api.h编译的tensorflow.dll。

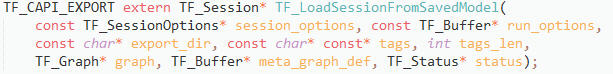
**3.4.1使用Cmake和bazel编译ni\_tensorflow.dll和libni\_tnsorflow.so**

* 安装软件

Windows下安装软件VS Enterprise 2015 with Visual C++ 2015、Git、Python 、SWIG (swigwin-3.0.12)、CMake (cmake-3.9.5-win64-x64)，Linux下安装软件Git、 Python、bazel 0.8.1 or 0.7.0

* 通过git下载tensorflow库
* 添加自定义函数TF\_LoadSessionFromFrozenGraph

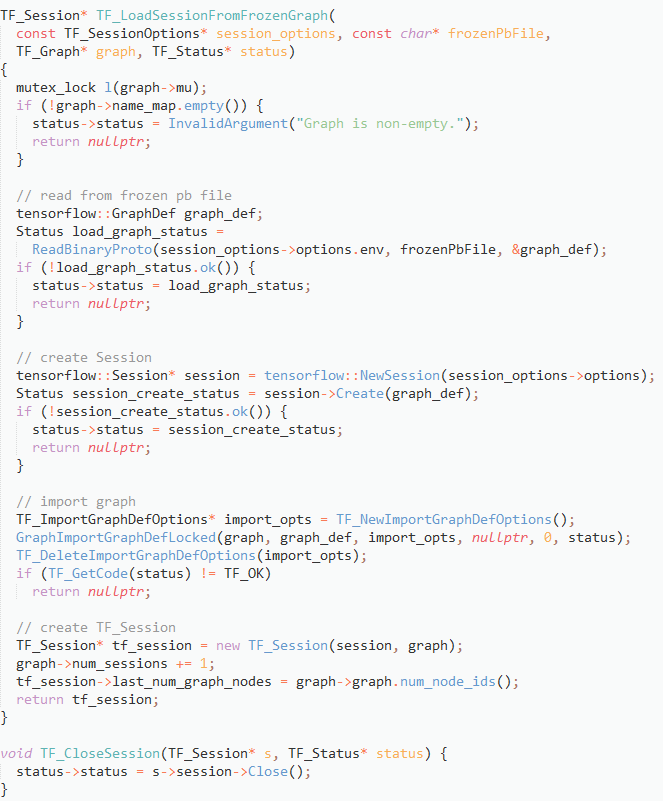
由于本设计支持tensorflow的两种模型文件savedModel和FrozenGraph（2.3节中介绍的b和c），但由于tensorflow的c\_api.h中值支持savedModel，因此我们仿照savedModel的写法添加函数TF\_LoadSessionFromFrozenGraph以支持frozenGraph（即单个pb文件）模型文件的加载及预测。





TF\_LoadSessionFromFrozenGrap函数使用session\_options从单个冻结的（frozen）pb文件中创建一个新的会话（TF\_Session）

具体实现如下：



* 在静态库名称前添加前缀ni\_

修改tf\_shared\_lib.cmake文件使得生成的静态库拥有ni\_前缀（比如，原本生成的静态库名称为tensorflow.dll,这步后生成的静态库名称为ni\_tensorflow.dll）

* 编译生成静态库

Windows下使用Cmake编译生成ni\_tensorflow.dll，Linux下使用Bazel编译生成libni\_tensorflow.so。

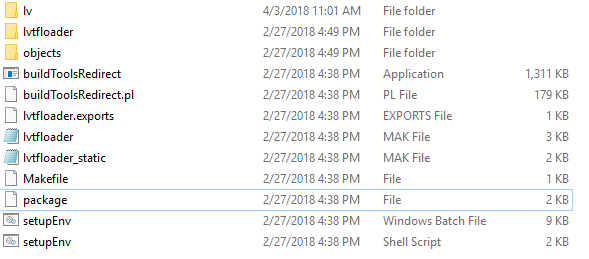
**3.4.2**使用NIBuild编译lvtfloader.dll / lvtfloader.so

此部分目的在于使用NIbuild根据ni\_tensorflow.dll / libni\_tensorflow.so得到lvtfloader.dll / lvtfloader.so（这两个共享库的作用主要是在TensorFlow数据类型和LabVIEW数据类型之间进行转换）。

NIBuild(即Build Services)为构建组件（components）提供了一种易于使用和平台独立的环境，并且在GNU之上建立了一个简单的界面，使组件的构建自动化。

NIBuild是一个易于设置和使用、执行常见的命名约定和版本控制方案的跨平台(Linux、Windows、Mac OS X、32和64位版本)编译工具。

NIBuild中所需要的文件如下图，



* 在Windows 64位和Linux 64位机器上配置环境
* 编写包文件（Package File）

包文件是一种可读的文本文件，具有两种类型的块:组件（component）和依赖项（dependency**）**.

组件部分指明构建给定二进制文件所需的依赖项。依赖项可以位于这个包文件中的其他组件，也可以是在一个依赖块中列出的组件。在组件部分中还有其他信息，比如人类可读的名称、类型、版本等等，包中的第一个组件称为“组件”，其他组件称为“子组件”。

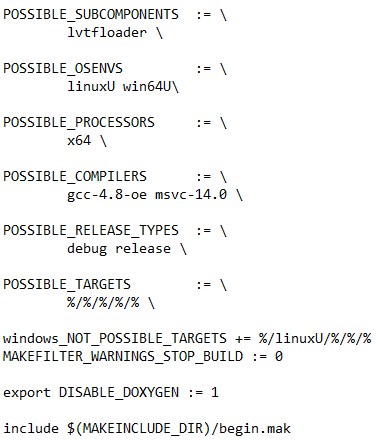
一个依赖项列出了一个Perforce位置，如果它不是包的一部分，则可以找到给定组件的包文件。当构建过程构建一个组件时，它会检查组件的依赖项列表，并将适当的头和库路径放入编译器和链接器步骤中。



本设计的包文件中组件为lvtfloader，依赖性为nibuild、tensorflow、ni\_tensorflow、lvexportsv。

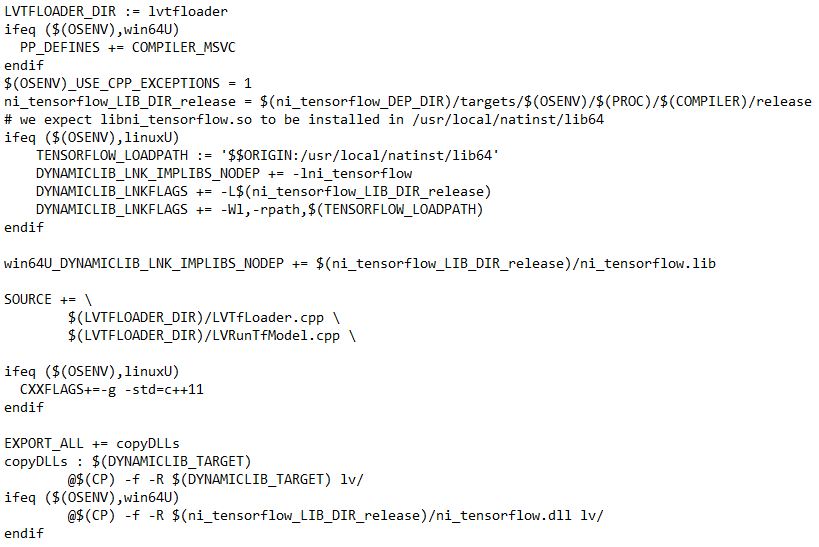
* 编写顶层的Makefile文件

顶层的Makefile文件定义了该组件应该构建的环境、处理器、编译器和构建类型



子组件.mak文件

如前所述，每个组件(和子组件)都必须有相应的.mak文件。mak文件包括指向源文件的路径，其中包括设置组件的各种属性的其他标志。为了生成我们需要的共享库文件，我们编写了lvtfloader.mak文件，lvtfloader.mak指明了源文件的路径，依赖的共享库ni\_tensorflow.dll路径已经生成的共享库路径。



由代码中可看出源文件为LVTfLoader.cpp和LVRunTfModel.cpp

* 自动化编译

打开命令窗口（CMD）进入相应目录运行

setupEnv.bat （windows下）或. ./setupEnv.sh （Linux下）：将构建服务工具添加到shell的路径中、

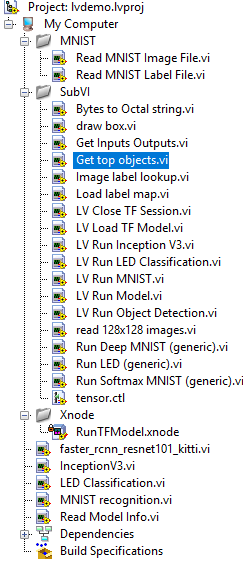
make sync：同步包文件中定义的所有依赖项

make

打开命令

完成第2部分内容后，我们得到了LabVIEW所需的共享库lvtfloader.dll / lvtfloader.so。

**3.4.3** LabVIEW项目

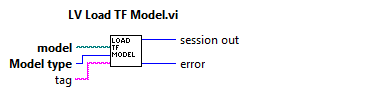


Lvdemo.lvproj项目主要包含四个针对不同模型的高层VIs：MNIST recognition、LED Classification、InceptionV3和faster\_rcnn\_resnet101\_kitti

1. MNIST recognition

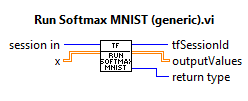
MNIST recognition用来加载MNIST训练好的模型并预测手写的0-9的数字。包含四个子VI：LV Load TF Model.vi、Run Softmax MNIST 、Run Deep MNIST 、LV Close TF Session。

* LV Load TF Model.vi



LV Load TF Model 接受三个参数：待打开的模型文件，模型文件类型（Saved Model或Frozen Graph）以及tag（serve或train），返回一个session引用。如果模型文件类型为Saved Model则该VI调用lvtfloader.dll中的LVLoadSavedModel函数，如果如果模型文件类型为Frozen Graph则该VI调用lvtfloader.dll中的LVLoadFrozenGraph函数。

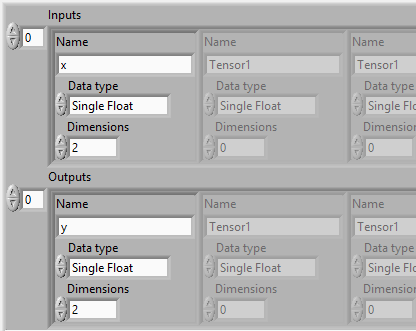
* Run Softmax MNIST



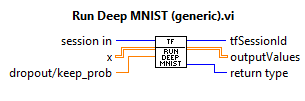
Run Softmax MNIST接受两个参数：session（一般来说是LV Load TF Model返回的session）和待预测的手写数字数据（一个二维数组），返回session和预测结果。

Run Softmax MNIST中调用了RunTFModel.xnode，RunTFModel.xnode是用来配置输入/输出来运行模型。因为对于tensorflow来说，要使用一个模型文件进行预测，需要输入模型文件的输入/输出。

对于 Softmax MNIST model所使用的模型文件，它要求的输入为一个TF\_Tensor。（TF\_Tensor是一个单数据类型的多维数组。）该TF\_Tensor的名称为“x“的二维数组，要求的输出是名称为“y“的二维数组。则Softmax MNIST model对应的Xnode的配置如下：

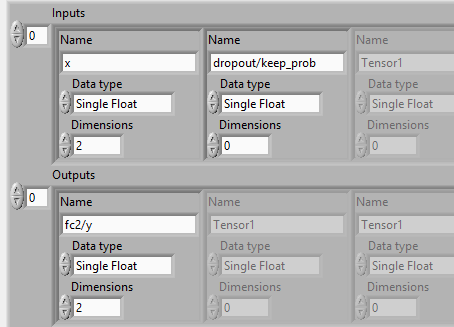


* Run Deep MNIST



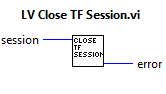
Run Deep MNIST与Run Softmax MNIST的功能类似，它接受三个参数：session（一般来说是LV Load TF Model返回的session）、待预测的手写数字数据（一个二维数组）以及dropout的概率，返回session和预测结果。

Run Deep MNIST中调用了RunTFModel.xnode，其对应的Xnode的配置如下：



因为Deep MNIST model所使用的模型文件由两个输入和一个输出。

* LV Close TF Session

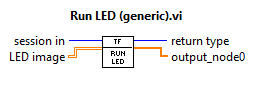


LV Close TF Model 接受一个session，返回一个错误值。若错误值为0，则说明程序正常运行，否则说明程序产生了错误。该VI调用lvtfloader.dll中的LVCloseTfSession函数。

2）LED Classification

LED Classification用来加载一个frozen Graph模型文件并预测一个LED灯是否是好的。它输入多个LED灯管的图片返回它的预测结果。包含三个子VI：LV Load TF Model.vi、LV Run LED Classification、LV Close TF Session。其中Load和Close模型文件与1）中相同

* Run LED

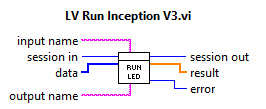
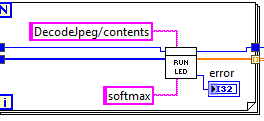


该Vi输入session和LED灯管的图片数据，输出预测结果。

1. Inception V3

Inception V3使用谷歌公司训练好的Inception V3（frozen Graph）模型文件classify\_image\_graph\_def.pb预测任意输入的一张图片的类别。它输入一个jpg图片路径返回它预测的前五个结果。包含三个子VI：LV Load TF Model.vi、LV Run Inception V3、LV Close TF Session。其中Load和Close模型文件与1）中相同

* LV Run Inception V3

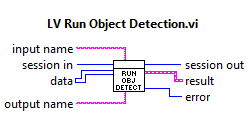
 

LV Run Inception V3输入session、模型文件输入/输出的名称以及待预测图片数据，返回预测结果。classify\_image\_graph\_def.pb模型文件输入张量名称为DecodeJpeg/contents，输出张量名称为softmax。该VI调用lvtfloader.dll中的LVRunInceptionV3函数。

1. faster\_rcnn\_resnet101\_kitti

faster\_rcnn\_resnet101\_kitti使用训练好的模型文件frozen\_inference\_graph.pb从图片中检测到人和汽车的物体。它输入一个jpg图片路径返回它预测的前五个结果。包含三个子VI：LV Load TF Model.vi、LV Run Object Detection、LV Close TF Session。其中Load和Close模型文件与1）中相同

* LV Run Object Detection



LV Run Object Detection输入session、模型文件输入/输出的名称以及待预测图片数据，返回预测结果。frozen\_inference\_graph.pb模型文件输入张量名称为image\_tensor，输出张量名称为空。该VI调用lvtfloader.dll中LVRunObjejctDetection函数。

3.5 本章小结

本章通过对系统框架和所用的相关技术知识的介绍，展示了系统的组成、架构和基本的功能。又通过对各大功能和其下的小模块的详细介绍，展示了系统每个功能和模块的运行原理、设计情况和实现方法。

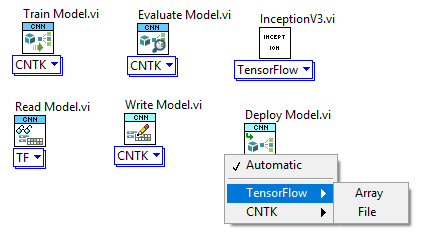
第四章 系统展示与结果对比

本章的内容包括系统最终结果的展示、实验结果的分析。

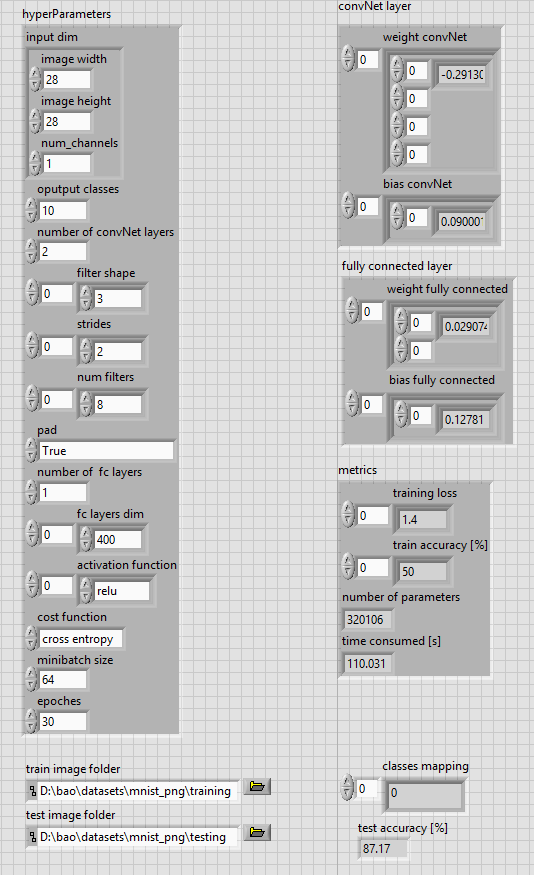
4.1 LabVIEW MNIST demo

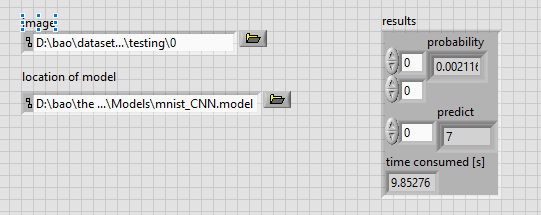
由调用python代码的VI组成的多态VI如图4.1.1所示。Train Model包含了Train Model(CNTK).vi和Train Model(TensorFLow).vi，Deploy Model包含了Train Model(CNTK,Array).vi、Train Model(CNTK,File).vi、Train Model(TensorFLow,Array).vi，

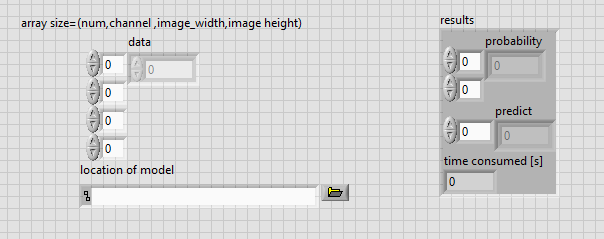
和Train Model(TensorFLow, File).vi。在以后的使用中直接使用多态VI/

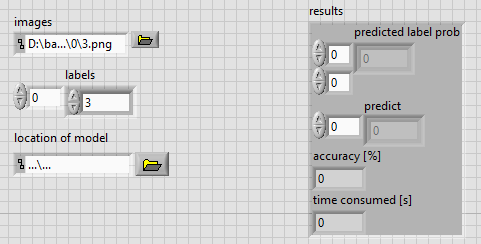


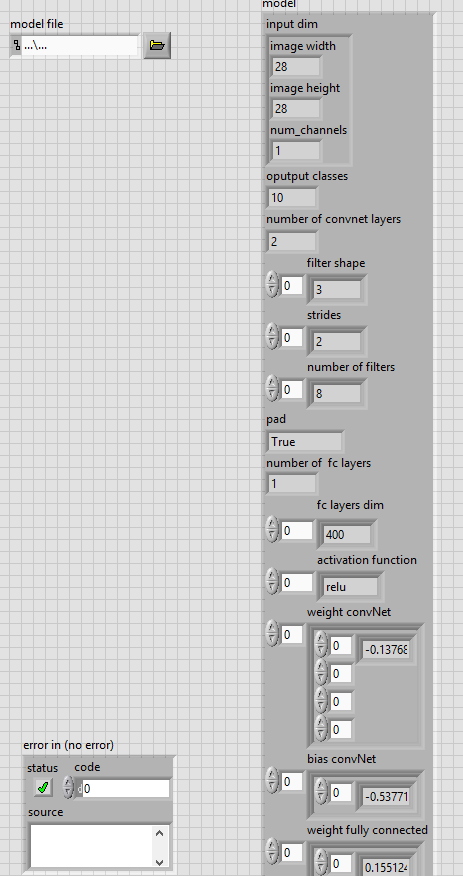
Train Model：

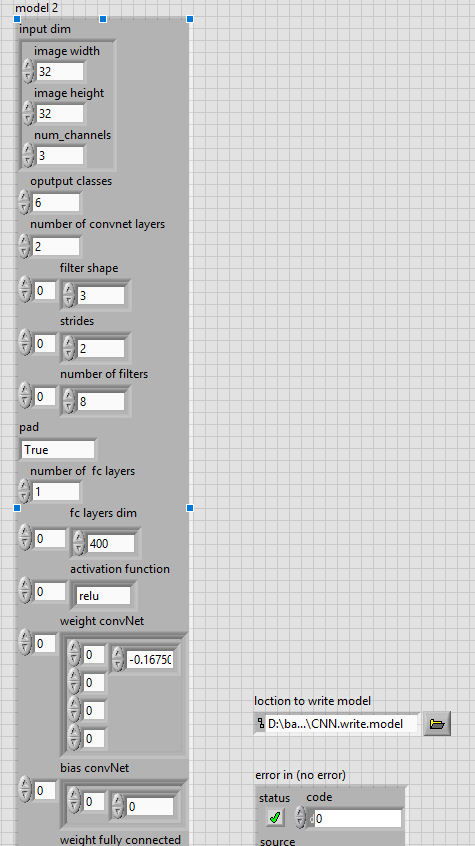


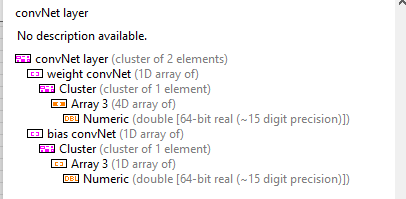














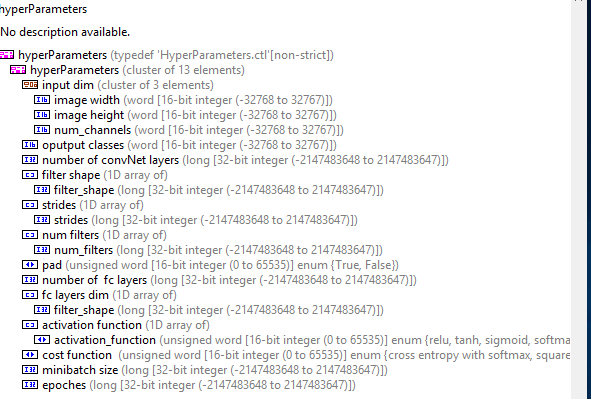
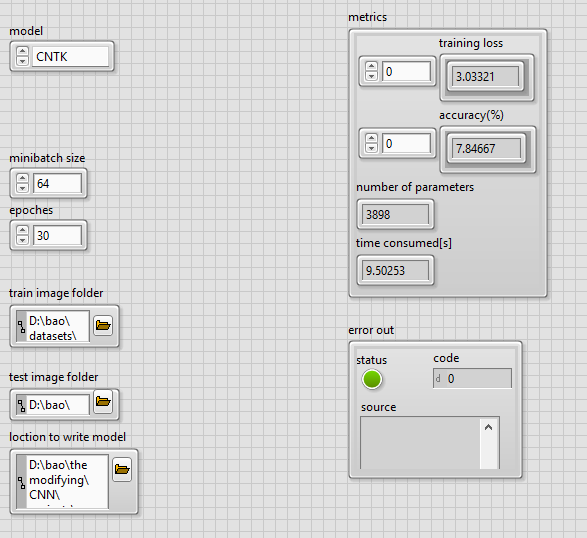


图4.1.1

我们为Train Model，Deploy Model，Read Model和Write Model设计了两个应用的VI：Model Evaluate and Hand Written.vi和Model Read Train And Write.vi，以MNIST数据为输入。

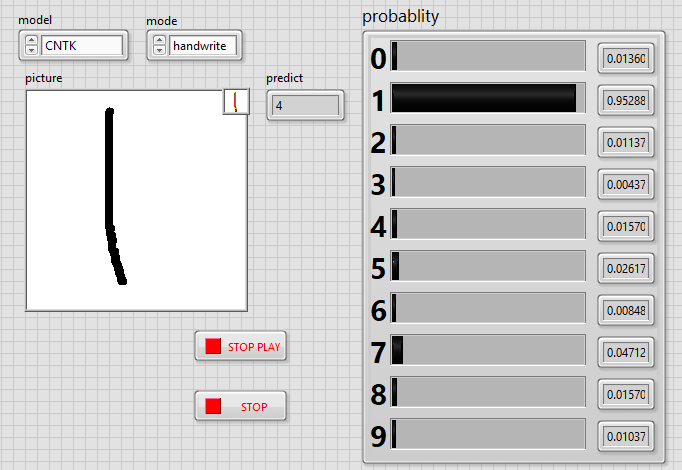
**4.1.1** Model Read Train And Write.vi

该demo先用Read Model从一个模型文件中读出模型输入大小（input dim）、模型输出类别（oputput classes），卷积层数（number of convNet layers）及每层的滤波器形状个数（filter shape，strides，num filters，pad），全连接层数（number of fc layers）及每层神经元（fc layers dim），激活函数（activation function），再将这些值传入Train Model，并为Train Model设置训练数据集，批处理大小和epoches，训练结束后，根据Train Model返回的权重矩阵和偏置项的值写入Write Model。用户可以选择是使用CNTK还是TensorFlow。

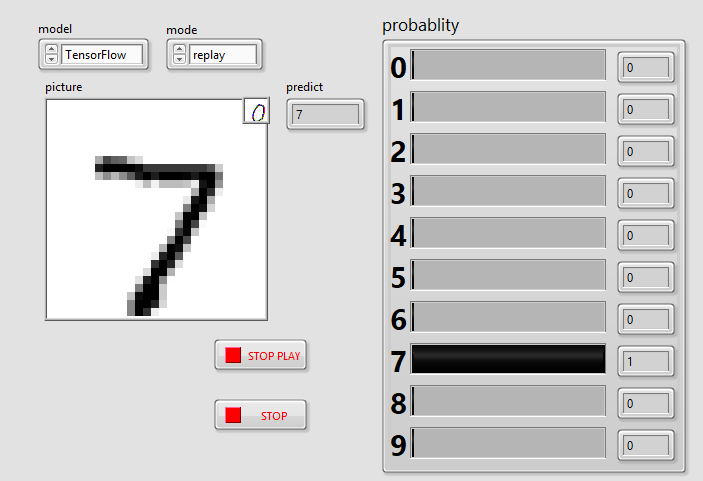


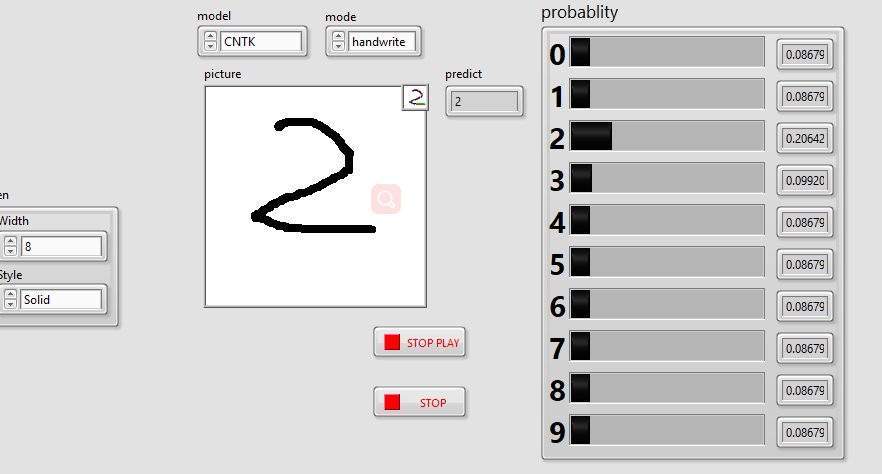
**4.1.2** Model Evaluate and Hand Written.vi

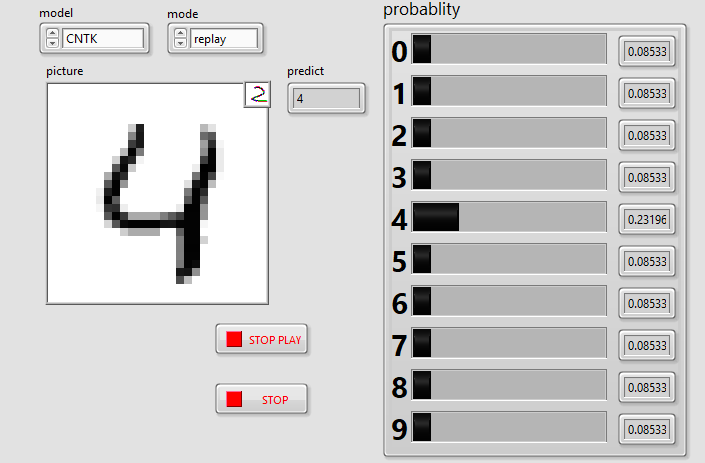
该demo使用Deploy Model对手写数字进行实时预测以及对输入的数据进行轮播展示，





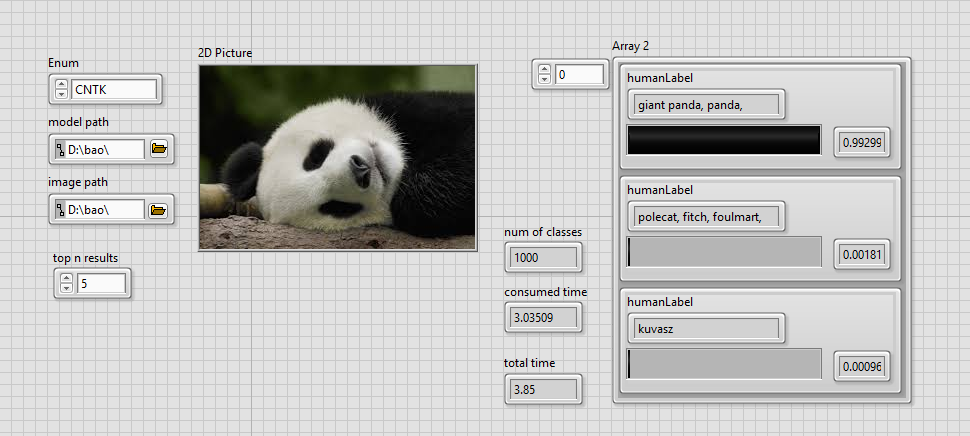






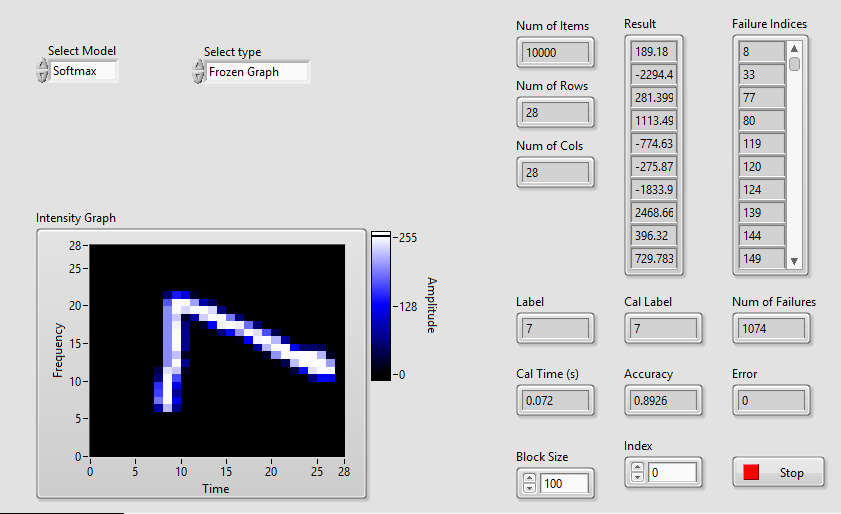
4.2 Inception V3 demo

该demo使用InceptionV3.vi对输入的图片进行预测

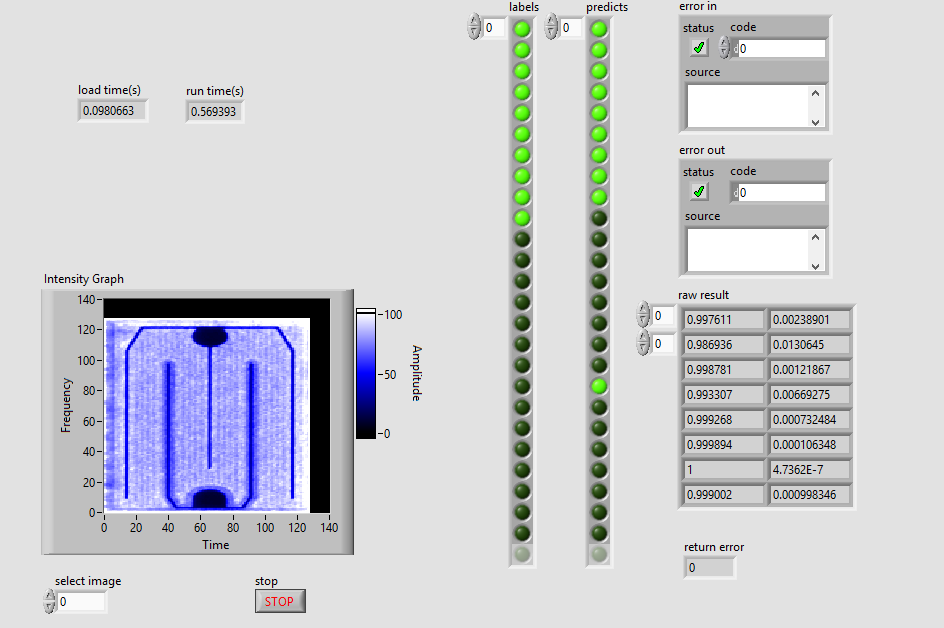


4.3 **lvdemo.lvproj demo**

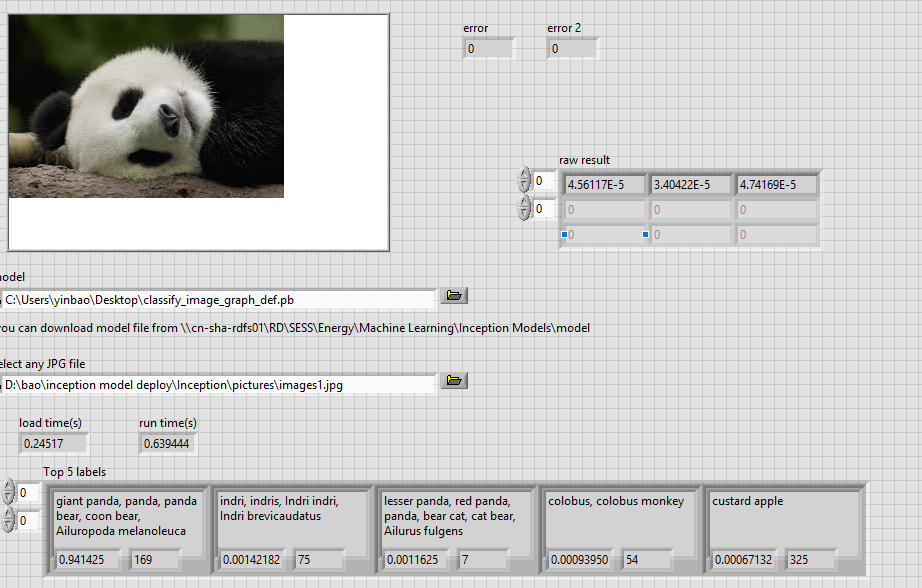
MNIST recognition、



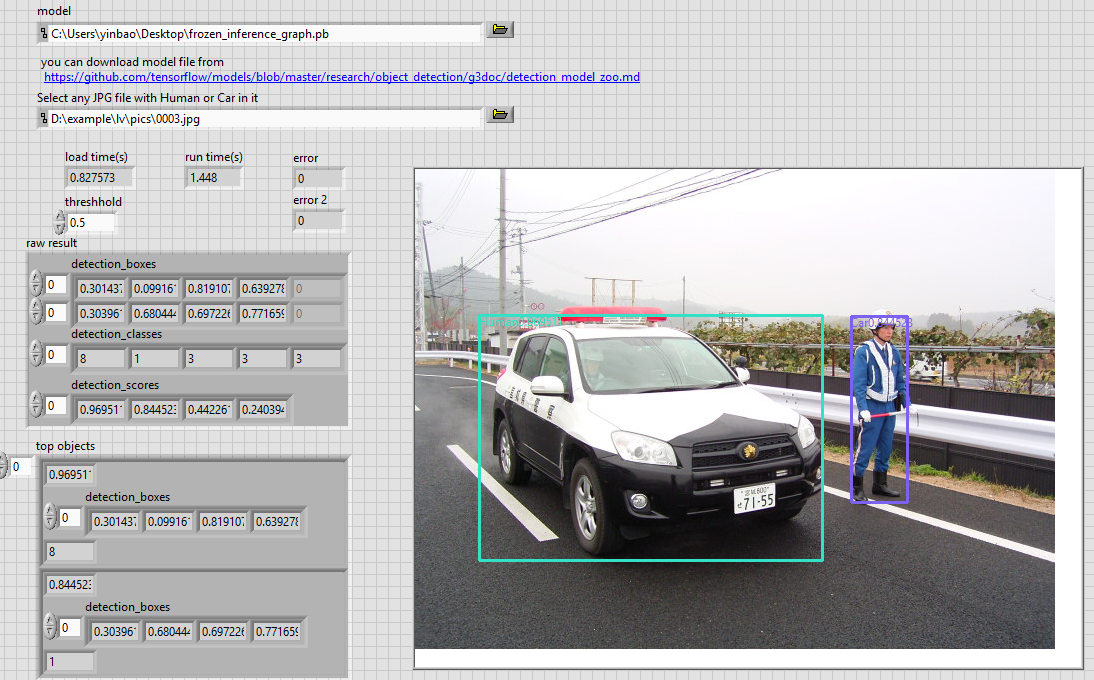
LED Classification



InceptionV3



faster\_rcnn\_resnet101\_kitti



4.4 消耗时间对比

|  |  |  |
| --- | --- | --- |
|  | **Python node** | **Python Integration Toolkit** |
| Train (CNTK) | 202.932 | 108.942 |
| Train (Tensorflow) | 153.367 | 54.583 |
| Deploy (CNTK ,Array) | 0.723 | 1.878 |
| Deploy (Tensorflow,Array) | 1.827 | 1.321 |
| Deploy (CNTK ,File) | 1.294 |  |
| Deploy (Tensorflow,File) | 2.997 |  |
| Read (CNTK) | 1.252 | 1.811 |
| Read (Tensorflow) | 3.053 | 3.523 |
| write (CNTK) | 1.214 | 1.349 |
| Write (Tensorflow) | 1.063 | 1.409 |

|  |  |  |
| --- | --- | --- |
|  | **Python node** | **Call Library Node** |
| Inception V3 (CNTK) | 2.025 | \ |
| Inception V3 (Tensorflow) | 1.913 | 0.655 |

* + **MNIST数据集：Filter shape:[(3,3),(3,3)]**
  + **Strides:[(2,2),(1,1)]**
  + **Number of filter:(8,16)**
  + **Weight dims : [8x1x3x3,**
  + **16x8x3x3,**
  + **16x7x7x10,]**
  + **Activation function : [relu, relu]**
  + **Cost function : cross entropy with softmax**
  + **Minibatch size=64**
  + **Iteration=18750**
  + **Totally 1898 parameters**

~~4.5 准确度~~

|  |  |  |
| --- | --- | --- |
|  | **CNTK** | **Tensorflow** |
| Test-Accuracy | 97.05% | 98.47% |
| Train-Time-cost | 358.6s | 397.995s |

**Trained with 60000 different test images**

**Tested with 10000 different test images**

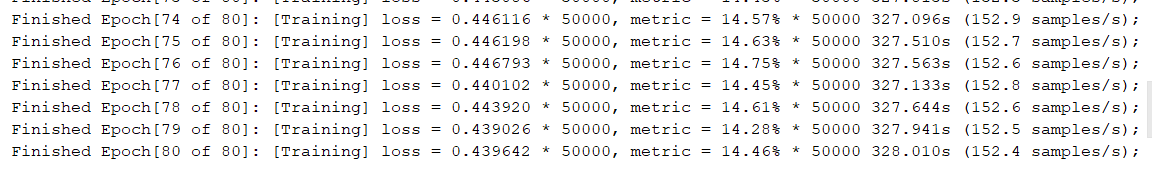
**Test Machine info :**

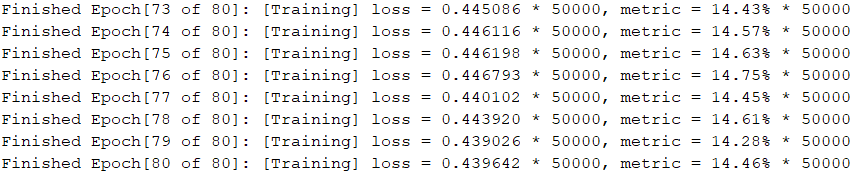
**OS : Windows 10 Enterprise**

**CPU : Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz**

**RAM : 8.00 GB**

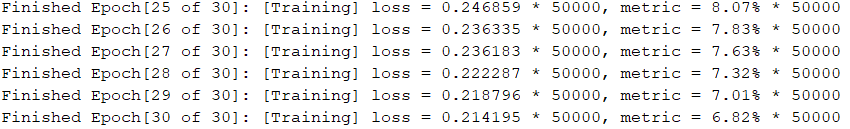
**ConvNet\_CIFAR10\_DataAug.py**







**ConvNet\_CIFAR10：**



**MNIST：**

4.6 本章小结

本章介绍了实验的最终结果，并且对于系统的实际效果进行了展示，同时对实验结果进行了一些简单的分析，验证了课题的意义及方案的可行性。

第五章 总结与展望

5.1 总结

5.2 展望

参考文献

[1] LeCun, Yann, et al. "Backpropagation applied to handwritten zip code recognition." Neural computation 1.4 (1989): 541-551

[2] Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets[J]. Neural computation, 2006, 18(7): 1527-1554.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

[4] Lin M, Chen Q, Yan S. Network in network[J]. arXiv preprint arXiv:1312.4400, 2013.

[5] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.

[6] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167 ,2015.

[7] Christian Szegedy, Vincent Vanhoucke, et al. Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567 ,2015.

[8]Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.

[10] He, Kaiming, et al. "Identity mappings in deep residual networks." European Conference on Computer Vision. Springer International Publishing, 2016.

[11] M. Abadi, A. Agarwal et al., aTensorflow: Large-scale machine learning on heterogeneous distributed systems,” arXiv preprint arXiv:1603.04467, Mar.2016.

[12] Amit Agarwal, Eldar Akchurin.An Introduction to Computational Networks and the Computational Network Toolkit. 2016

[13] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In ECCV 2014

[14] 神经网络七十年： 回顾与展望[J]，计算机学报，2016，（8），1697-1716

[15] 张军阳,王慧丽,郭阳,扈啸.深度学习相关研究综述[J].计算机应用研究,2018,(07):1-12.

[16] 孙志军，薛 磊，许阳明，王正，深度学习研究综述[J]，计算机应用研究，2012，（08），2806-2810

[17] 邹蕾，张先锋，人工智能及其发展应用[J]，理论研究，2012，（02），11-13

[18] 余凯，贾磊，陈雨强，徐伟，深度学习的昨天、今天和明天[J]，计算机研究与发展，2013，（9），1799-1804

致 谢