架构师

ARCHITECT

特刊

深入浅出TensorFlow

郑泽宇著

SPECIAL ISSUE
Sep,2017

架构师特刊

Geekbang 极客邦科技 | InfoQ

卷首语

郑泽宇

从 2016 年以来，人工智能和深度学习这些概念已经为大部分人所熟知。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。如今，国务院更是发文明确表示人工智能将成为国家的战略核心，要大力投入人工智能的发展。在这样的背景下，如何上手人工智能、深度学习自然成为了很多人的希望了解的事情。

为了帮助大家能够在绕开复杂的数据公式的同时了解人工智能核心技术——深度学习，笔者于今年 2 月出版了图书《TensorFlow：实战 Google 深度学习框架》。该书深入浅出地介绍这些深度学习算法背后的理论并给出这些算法可以解决的具体问题。该书从实际问题出发，在实践中介绍深度学习的概念和 TensorFlow 的用法。

为了让广大读者更快的了解和掌握深度学习最基本和核心的概念，作者与 InfoQ 合作完成了 7 篇介绍 TensorFlow 的连载文章。这些文章覆盖了《TensorFlow：实战 Google 深度学习框架》书中最核心、最基础的内容，能够带领读者快速了解深度学习中比较基础的内容，让读者快速入门。

目录

深度学习及 TensorFlow 简介

2017 年 2 月 16 日，Google 正式对外发布 Google TensorFlow 1.0 版本，并保证本次的发布版本 API 接口完全满足生产环境稳定性要求。这是 TensorFlow 的一个重要里程碑，标志着它可以正式在生产环境放心使用。在国内，从 InfoQ 的判断来看，TensorFlow 仍处于创新传播曲线的创新者使用阶段，大部分人对于 TensorFlow 还缺乏了解，社区也缺少帮助落地和使用的中文资料。InfoQ 期望通过深入浅出 TensorFlow 系列文章能够推动 Tensorflow 在国内的发展。欢迎加入 QQ 群（群号：183248479）深入讨论和交流。

本文是整个系列的第一篇文章，将会介绍深度学习的发展历史以及深度学习目前成熟的应用，同时也会介绍目前主流的深度学习工具，以及 TensorFlow 相比于其他工具的优势。

从计算机发明之初，人们就希望它能够帮助甚至代替人类完成重复性劳作。利用巨大的存储空间和超高的运算速度，计算机已经可以非常轻易地完成一些对于人类非常困难，但对计算机相对简单的问题。比如统计一本书中不同单词出现的次数，存储一个图书馆中所有的藏书或是计算非常复杂的数学公式都可以轻松通过计算机解决。然而，一些人类通过直觉可以很快解决的问题，目前却很难通过计算机解决。人工智能领域需要解决的问题就是让计算机能像人类一样，甚至超越人类完成类似图像识别、语音识别等问题。

计算机要像人类一样完成更多智能的工作需要够掌握人类的经验。比如我们需要判断一封邮件是否为垃圾邮件，会综合考虑邮件发出的地址、邮件的标题、邮件的内容以及邮件收件人的长度，等等。这是我们受到无数垃圾邮件骚扰之后总结出来的经验。这个经验很难以固定的方式表达出来，而且不同人对垃圾邮件的判断也会不一样。如何让计算机可以跟人类一样从历史的经验中获取新的知识呢？这就是机器学习需要解决的问题。

什么是机器学习?

卡内基梅隆大学的 Tom Michael Mitchell 教授在 1997 年出版的书籍 Machine Learning 中有对机器学习进行非常专业的定义，这个定义在学术界内被多次引用。在这本书中对机器学习的定义为“如果一个程序可以在任务 T 上，随着经验 E 的增加，效果 P 也可以随之增加，则我们称这个程序可以从经验中学习”。让我们通过垃圾邮件分类的问题来解释机器学习的定义。在垃圾邮件分类问题中，“一个程序”指的是需要用到的机器学习算法，比如逻辑回归算法；“任务 T”是指区分垃圾邮件的任务；“经验 E”为已经区分过是否为垃圾邮件的历史邮件，在监督式机器学习问题中，这也被称之为训练数据，也就是说对于机器学习而言，所谓的经验其实就是数据；“效果 P”为机器学习算法在区分是否为垃圾邮件任务上的正确率。

在使用逻辑回归算法解决垃圾邮件分类问题时，我们会先从每一封邮件中抽取对分类结果可能有影响的因素，比如说上文提到的发邮件的地

址、邮件的标题及收件人的长度，等等。每一个因素被称之为一个特征（feature）。逻辑回归算法可以从训练数据中计算出每个特征和预测结果的相关度。比如在垃圾邮件分类问题中，我们可能会发现如果一个邮件的收件人越多，那么邮件为垃圾邮件的概率也就越高。在对一封未知的邮件做判断时，逻辑回归算法会根据从这封邮件中抽取得到的每一个特征以及这些特征和垃圾邮件的相关度来判断这封邮件是否为垃圾邮件。

在大部分情况下，在训练数据达到一定数量之前，越多的训练数据可以使逻辑回归算法对未知邮件做出的判断越精准。也就是说逻辑回归算法可以根据训练数据（经验 E）提高在垃圾邮件分类问题（任务 T）上的正确率（效果 P）。之所以说在大部分情况下，是因为逻辑回归算法的效果除了依赖于训练数据，也依赖于从数据中提取的特征。假设我们从邮件中抽取的特征只有邮件发送的时间，那么即使有再多的训练数据，逻辑回归算法也无法很好地利用。这是因为邮件发送的时间和邮件是否为垃圾邮件之间的关联不大，而逻辑回归算法无法从数据中习得更好的特征表达。这也是很多传统机器学习算法的一个共同的问题。

什么是深度学习?

对许多机器学习问题来说，特征提取不是一件简单的事情。在一些复杂问题上，要通过人工的方式设计有效的特征集合需要很多的时间和精力，有时甚至需要整个领域数十年的研究投入。例如，假设想从很多照片中识别汽车。现在已知的是汽车有轮子，所以希望在图片中抽取“图片中是否出现了轮子”这个特征。但实际上，要从图片的像素中描述一个轮子的模式是非常难的。虽然车轮的形状很简单，但在实际图片中，车轮上可能会有来自车身的阴影、金属车轴的反光，周围物品也可能会部分遮挡车轮。实际图片中各种不确定的因素让我们很难直接抽取这样的特征。

深度学习解决的核心问题之一就是自动地将简单的特征组合成更加复杂的特征，并使用这些组合特征解决问题。深度学习是机器学习的一个分支，它除了可以学习特征和任务之间的关联以外，还能自动从简单特征中

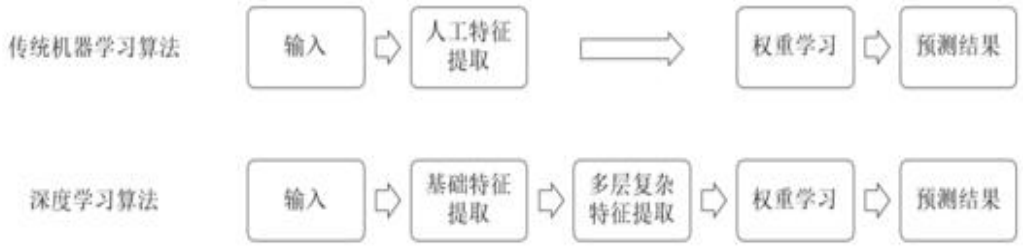


图 1 传统机器学习和深度学习流程对比

提取更加复杂的特征。图 1 中展示了深度学习和传统机器学习在流程上的差异。如图 1 所示，深度学习算法可以从数据中学习更加复杂的特征表达，使得最后一步权重学习变得更加简单且有效。在图 2 中，展示了通过深度学习解决图像分类问题的具体样例。深度学习可以一层一层地将简单特征逐步转化成更加复杂的特征，从而使得不同类别的图像更加可分。比如图 2 中展示了深度学习算法可以从图像的像素特征中逐渐组合出线条、边、角、简单形状、复杂形状等更加有效的复杂特征。

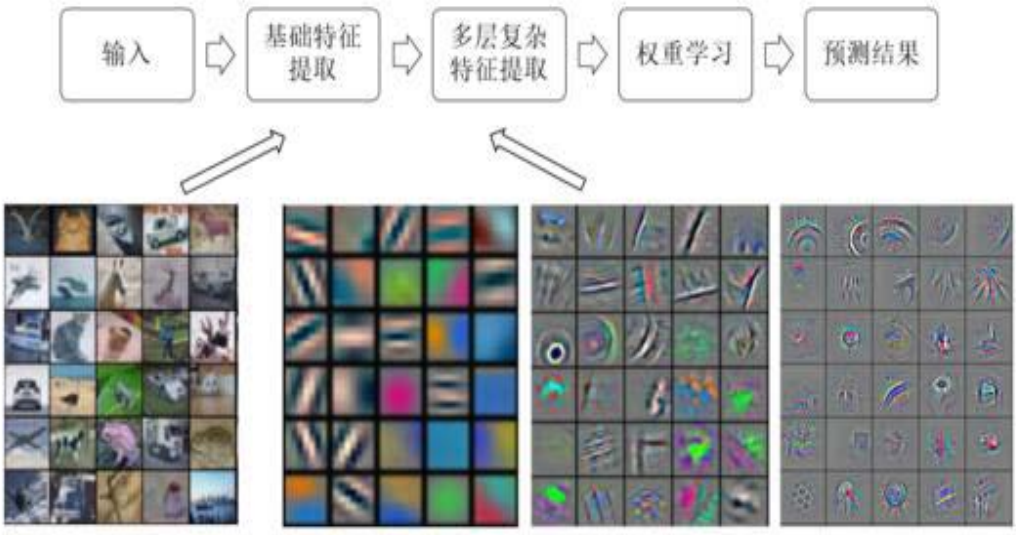


图 2 深度学习在图像分类问题上的算法流程样例

人工智能、机器学习和深度学习的关系

总的来说，人工智能、机器学习和深度学习是非常相关的几个领域。图 3 总结了它们之间的关系。人工智能是一类非常广泛的问题，机器学习

是解决这类问题的一个重要手段，深度学习则是机器学习的一个分支。在很多人工智能问题上，深度学习的方法突破了传统机器学习方法的瓶颈，推动了人工智能领域的发展。

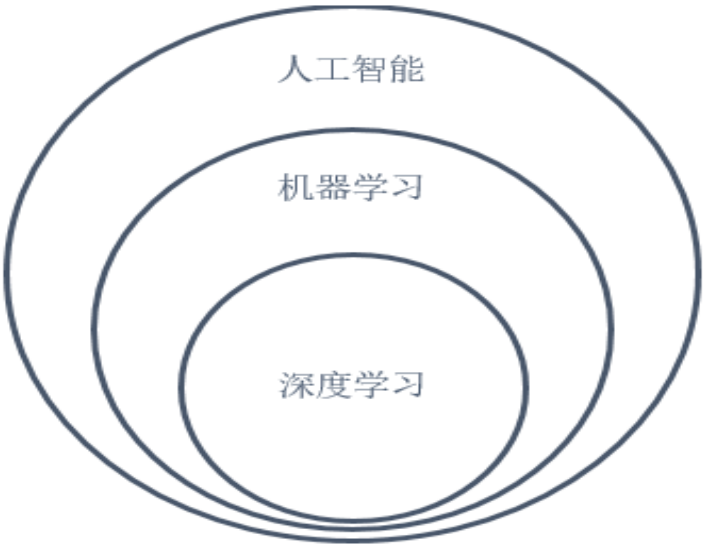


图 3 人工智能、机器学习以及深度学习之间的关系图

图 4 展示了“deep learning”（深度学习）这个词在最近十年谷歌搜索的热度趋势。从图中可以看出，从 2012 年之后，深度学习的热度呈指数上升，到 2016 年时，深度学习已经成为了谷歌上最热门的搜索词。深度学习这个词并不是最近才创造出来的，它基本就是深层神经网络的代名词。受到人类大脑结构的启发，神经网络的计算模型于 1943 年首次提出。之后感知机的发明使得神经网络成为真正可以从数据中“学习”的模型。但由于感知机的网络结构过于简单，导致无法解决线性不可分问题。再加上神经网络所需要的计算量太大，当时的计算机无法满足计算需求，使得神经网络的研究进入了第一个寒冬。

到 20 世纪 80 年代，深层神经网络和反向传播算法的提出很好地解决了这些问题，让神经网络进入第二个快速发展期。不过，在这一时期中，以支持向量机为主的传统机器学习算法也在飞速发展。在 90 年代中期，在很多机器学习任务上，传统机器学习算法超越了神经网络的精确度，使得神经网络领域再次进入寒冬。直到 2012 年前后，随着云计算和海量数

据的普及，神经网络以“深度学习”的名字再次进入大家的视野。2012 年，深度学习算法 AlexNet 赢得图像分类比赛 ILSVRC（ImageNet Large Scale Visual Recognition Challenge）冠军，深度学习从此开始受到学术界广泛的关注。

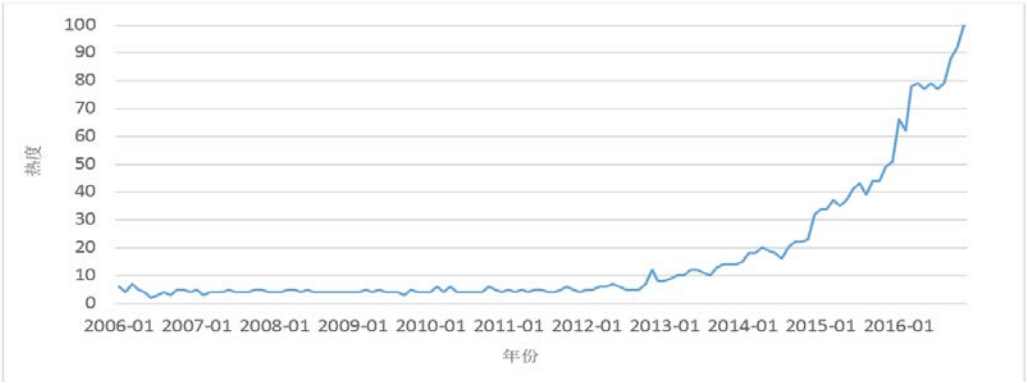


图 4 Deep Learning 最近十年在谷歌搜索的热度趋势

深度学习的应用

图 5 展示了历年 ILSVRC 比赛的情况，从图中可以看到，在深度学习被使用之前，传统计算机视觉的方法在 ImageNet 数据集上最低的 Top5 错误率为 26%。从 2010 年到 2011 年，基于传统机器学习的算法并没有带来正确率的大幅提升。在 2012 年时，Geoffrey Everest Hinton 教授的研究小组利用深度学习技术将 ImageNet 图像分类的错误率大幅下降到了 16%。而且，从 2012 年到 2015 年间，通过对深度学习算法的不断研究，

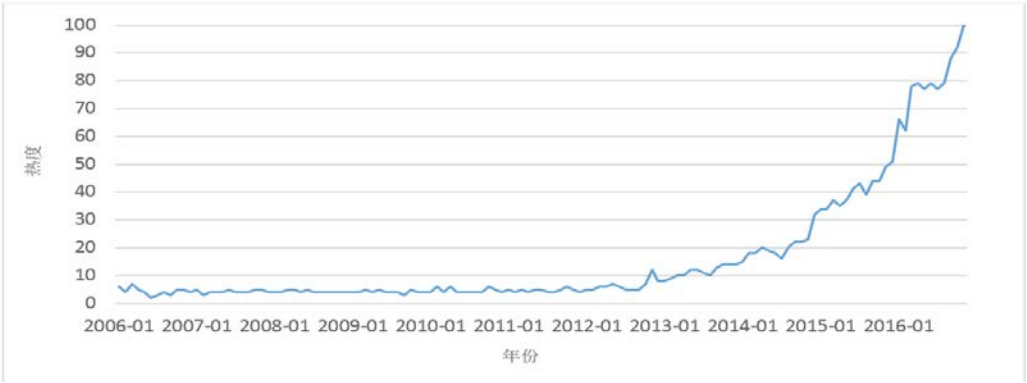


图 5 历年 ILSVRC 图像分类比赛最佳算法的错误率

ImageNet 图像分类的错误率以每年 4% 的速度递减。这说明深度学习完全打破了传统机器学习算法在图像分类上的瓶颈，让图像分类问题得到了更好的解决。如图 5 所示，到 2015 年时，深度学习算法的错误率为 4%，已经成功超越了人工标注的错误率（5%），实现了计算机视觉研究领域的一个突破。

在技术革新的同时，工业界也将图像分类、物体识别应用于各种产品中了。在谷歌，图像分类、物体识别技术已经被广泛应用于谷歌无人驾驶车、YouTube、谷歌地图、谷歌图像搜索等产品中。谷歌通过图像处理技术可以归纳出图片中的主要内容并实现以图搜图的功能。这些技术在国内的百度、阿里、腾讯等科技公司也已经得到了广泛的应用。

在物体识别问题中，人脸识别是一类应用非常广泛的技术。它既可以应用于娱乐行业，也可以应用于安防、风控行业。在娱乐行业中，基于人脸识别的相机自动对焦、自动美颜基本已经成为每一款自拍软件的必备功能。在安防、风控领域，人脸识别应用更是大大提高了工作效率并节省了人力成本。比如在互联网金融行业，为了控制贷款风险，在用户注册或者贷款发放时需要验证本人信息。个人信息验证中一个很重要的步骤是验证用户提供的证件和用户是同一个人。通过人脸识别技术，这个过程可以被更加高效地实现。

深度学习在语音识别领域取得的成绩也是突破性的。2009 年深度学习的概念被引入语音识别领域，并对该领域产生了巨大的影响。在短短几年时间内，深度学习的方法在 TIMIT 数据集上将基于传统的混合高斯模型（gaussian mixture model，GMM）的错误率从 21.7% 降低到了使用深度学习模型的 17.9%。如此大的提高幅度很快引起了学术界和工业界的广泛关注。从 2010 年到 2014 年间，在语音识别领域的两大学术会议 IEEE-ICASSP 和 Interspeech 上，深度学习的文章呈现出逐年递增的趋势。在工业界，包括谷歌、苹果、微软、IBM、百度等在内的国内外大型 IT 公司提供的语音相关产品，比如谷歌的 Google Now，苹果的 Siri、微软的 Xbox 和 Skype 等，都是基于深度学习算法。

深度学习在自然语言处理领域的应用也同样广泛。在过去的几年中，深度学习已经在语言模型（language modeling）、机器翻译、词性标注（part-of-speech tagging）、实体识别（named entity recognition，NER）、情感分析（sentiment analysis）、广告推荐以及搜索排序等问题上取得了突出成就。在机器翻译问题上，根据谷歌的实验结果，在主要的语言对上，使用深度学习可以将机器翻译算法的质量提高 55% 到 85%。表 1 对比了不同算法翻译同一句话的结果。从表中可以直观地看到深度学习算法带来翻译质量的提高。在 2016 年 9 月，谷歌正式上线了基于深度学习的中译英软件。现在在谷歌翻译产品中，所有从中文到英文的翻译请求都是由基于深度学习的翻译算法完成的。

表 1 不同翻译算法的翻译效果对比表

中文原句	李克强此行将启动中加总理年度对话机制，与加拿大总理杜鲁多举行两国总理首次年度对话。
基于传统机器学习算法的翻译结果	Li Keqiang premier added this line to start the annual dialogue mechanism with the Canadian Prime Minister Trudeau two prime ministers held its first annual session.
基于深度学习算法的翻译结果	Li Keqiang will start the annual dialogue mechanism with Prime Minister Trudeau of Canada and hold the first annual dialogue between the two premiers.
人工翻译结果	Li Keqiang will initiate the annual dialogue mechanism between premiers of China and Canada during this visit, and hold the first annual dialogue with Premier Trudeau of Canada.

要将深度学习更快且更便捷地应用于新的问题中，选择一款深度学习工具是必不可少的步骤。

TensorFlow 是谷歌于 2015 年 11 月 9 日正式开源的计算框架。TensorFlow 计算框架可以很好地支持深度学习的各种算法，但它的应用也不限于深度学习。

TensorFlow 是由 Jeff Dean 领头的谷歌大脑团队基于谷歌内部第一代深度学习系统 DistBelief 改进而来的通用计算框架。DistBelief 是谷歌 2011 年开发的内部深度学习工具，这个工具在谷歌内部已经获得了巨大的成功。

基于 DistBelief 的 ImageNet 图像分类系统 Inception 模型赢得了 ImageNet2014 年的比赛（ILSVRC）。通过 DistBelief，谷歌在海量的非标注 YouTube 视屏中习得了“猫”的概念，并在谷歌图片中开创了图片搜索的功能。使用 DistBelief 训练的语音识别模型成功将语音识别的错误率降低了 25%。在一次 BBC 采访中，当时的谷歌首席执行官 Eric Schmidt 表示这个提高比率相当于之前十年的总和。

虽然 DistBelief 已经被谷歌内部很多产品所使用，但是 DistBelief 过于依赖谷歌内部的系统架构，很难对外开源。为了将这样一个在谷歌内部已经获得了巨大成功的系统开源，谷歌大脑团队对 DistBelief 进行了改进，并于 2015 年 11 月正式公布了基于 Apache 2.0 开源协议的计算框架 TensorFlow。相比 DistBelief，TensorFlow 的计算模型更加通用、计算速度更快、支持的计算平台更多、支持的深度学习算法更广而且系统的稳定性也更高。关于 TensorFlow 平台本身的技术细节可以参考谷歌的论文 TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems。

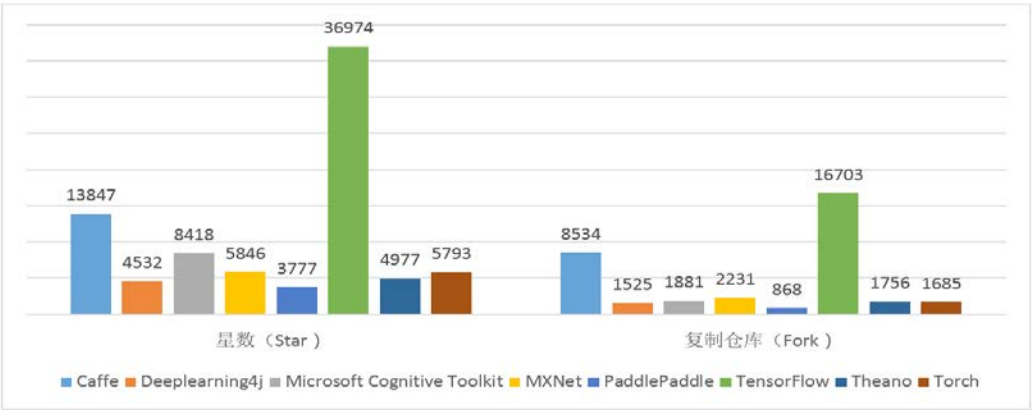
如今在谷歌内部，TensorFlow 已经得到了广泛的应用。在 2015 年 10 月 26 日，谷歌正式宣布通过 TensorFlow 实现的排序系统 RankBrain 上线。相比一些传统的排序算法，使用 RankBrain 的排序结果更能满足用户需求。在 2015 年彭博（Bloomberg）的报道中，谷歌透露了在谷歌上千种排序算法中，RankBrain 是第三重要的排序算法。基于 TensorFlow 的系统 RankBrain 能在谷歌的核心网页搜索业务中占据如此重要的地位，可见 TensorFlow 在谷歌内部的重要性。包括网页搜索在内，TensorFlow 已经被成功应用到了谷歌的各款产品之中。如今，在谷歌的语音搜索、广告、电商、图片、街景图、翻译、YouTube 等众多产品之中都可以看到基于 TensorFlow 的系统。在经过半年的尝试和思考之后，谷歌的 DeepMind 团队也正式宣布其之后所有的研究都将使用 TensorFlow 作为实现深度学习算法的工具。

除了在谷歌内部大规模使用之外，TensorFlow 也受到了工业界和学术

界的广泛关注。在 Google I/O 2016 的大会上，Jeff Dean 提到已经有 1500 多个 GitHub 的代码库中提到了 TensorFlow，而只有 5 个是谷歌官方提供的。如今，包括优步（Uber）、Snapchat、Twitter、京东、小米等国内外科技公司也纷纷加入了使用 TensorFlow 的行列。正如谷歌在 TensorFlow 开源原因中所提到的一样，TensorFlow 正在建立一个标准，使得学术界可以更方便地交流学术研究成果，工业界可以更快地将机器学习应用于生产之中。

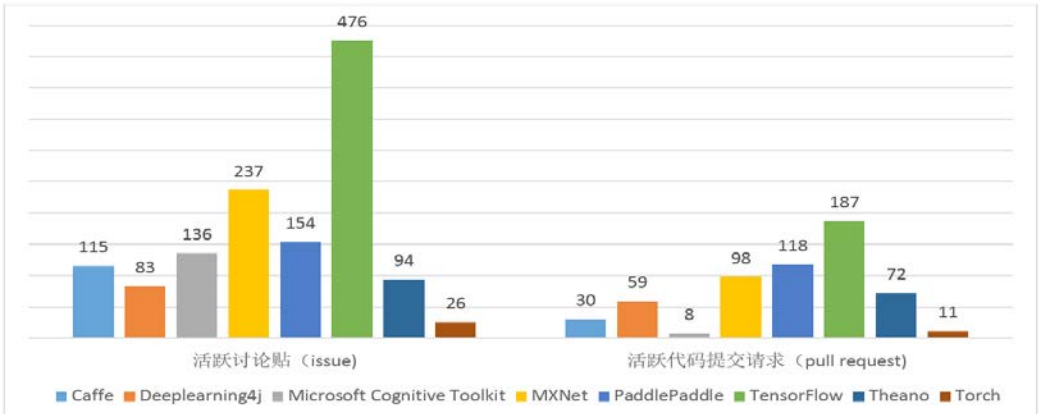
其他已开源的深度学习工具

除了 TensorFlow，目前还有一些主流的深度学习开源工具。笔者认为，不同的深度学习工具都在发展之中，比较当前的性能、功能固然是选择工具的一种方法，但更加重要的是比较不同工具的发展趋势。深度学习本身就是一个处于蓬勃发展阶段的领域，所以对深度学习工具的选择，笔者认为应该更加看重工具在开源社区的活跃程度。只有社区活跃度更高的工具，才有可能跟上深度学习本身的发展速度，从而在未来不会面临被淘汰的风险。



（a）不同深度学习工具社区流行度指标比较

图 6 对比了不同深度学习工具在 GitHub 上活跃程度的一些指标。图 6（a）中比较了不同工具在 GitHub 上受关注的程度。从图中可以看出，无论是在获得的星数（star）还是在仓库被复制的次数上，TensorFlow 都



（b）同深度学习工具社区参与度指标比较

图 6 不同深度学习工具在 GitHub 上活跃程度对比图，图中数据获取的时间为 2016 年 11 月 17 日

要远远超过其他的深度学习工具。如果说图 6(a) 只能代表不同深度学习工具在社区受关注程度，那么图 6(b) 对比了不同深度学习工具社区参与度。图 6(b) 中展示了不同深度学习工具在 GitHub 上最近一个月的活跃讨论贴和代码提交请求数量。活跃讨论帖越多，可以说明真正使用这个工具的人也就越多；提交代码请求数量越多，可以说明参与到开发这个工具的人也就越多。从图 6（b）中可以看出，无论从哪个指标，TensorFlow 都要远远超过其他深度学习工具。大量的活跃开发者再加上谷歌的全力支持，笔者相信 TensorFlow 在未来将有更大的潜力。

TensorFlow 解决 MNIST 问题入门

2017 年 2 月 16 日，Google 正式对外发布 Google TensorFlow 1.0 版本，并保证本次的发布版本 API 接口完全满足生产环境稳定性要求。这是 TensorFlow 的一个重要里程碑，标志着它可以正式在生产环境放心使用。在国内，从 InfoQ 的判断来看，TensorFlow 仍处于创新传播曲线的创新者使用阶段，大部分人对于 TensorFlow 还缺乏了解，社区也缺少帮助落地和使用的中文资料。InfoQ 期望通过深入浅出 TensorFlow 系列文章能够推动 Tensorflow 在国内的发展。

本文是整个系列的第二篇文章，将会简单介绍 TensorFlow 安装方法、TensorFlow 基本概念、神经网络基本模型，并在 MNIST 数据集上使用 TensorFlow 实现一个简单的神经网络。

TensorFlow 安装

Docker 是新一代的虚拟化技术，它可以将 TensorFlow 以及 TensorFlow 的所有依赖关系统一封装到 Docker 镜像当中，从而大大简化了安装过程。Docker 是可移植性最强的一种安装方式，它支持大部分的操作系统（比如 Windows，Linux 和 Mac OS）。对于 TensorFlow 发布的每一个版本，谷歌都提供了官方镜像。在官方镜像的基础上，才云科技提供的镜像进一步整合了其他机器学习工具包以及 TensorFlow 可视化工具 TensorBoard，使用起来可以更加方便。目前才云科技提供的镜像有：

```
cargo.caicloud.io/tensorflow/tensorflow:0.12.0 cargo.caicloud.io/tensorflow/
tensorflow:0.12.0-gpu cargo.caicloud.io/tensorflow/tensorflow:0.12.1 cargo.
caicloud.io/tensorflow/tensorflow:0.12.1-gpu cargo.caicloud.io/tensorflow/
tensorflow:1.0.0
cargo.caicloud.io/tensorflow/tensorflow:1.0.0-gpu
```

当 Docker 安装完成之后（Docker 安装可以参考 https://docs.docker.com/engine/installation/），可以通过以下命令来启动一个 TensorFlow 容器。在第一次运行的时候，Docker 会自动下载镜像。

```
$ docker run -p 8888:8888 -p 6006:6006 \
    cargo.caicloud.io/tensorflow/tensorflow:1.0.0
```

在这个命令中，-p 8888:8888 将容器内运行的 Jupyter 服务映射到本地机器，这样在浏览器中打开 localhost:8888 就能看到 Jupyter 界面。在此镜像中运行的 Jupyter 是一个网页版的代码编辑器，它支持创建、上传、修改和运行 Python 程序。

-p 6006:6006 将容器内运行的 TensorFlow 可视化工具 TensorBoard 映射到本地机器，通过在浏览器中打开 localhost:6006 就可以将 TensorFlow 在训练时的状态、图片数据以及神经网络结构等信息全部展示出来。此镜像会将所有输出到 /log 目录底下的日志全部可视化。

-it 将提供一个 Ubuntu 14.04 的 bash 环境，在此环境中已经将 TensorFlow 和一些常用的机器学习相关的工具包（比如 Scikit）安装完毕。

注意这里无论本地机器操作系统是什么，这个 bash 环境都是基于 Ubuntu 14.04 的。这是由编译 Docker 镜像的方式决定的，和本地的操作系统没有关系。

虽然有支持 GPU 的 Docker 镜像，但是要运行这些镜像需要安装最新的 NVidia 驱动以及 nvidia-docker。在安装完成 nvidia-docker 之后，可以通过以下的命令运行支持 GPU 的 TensorFlow 镜像。在镜像启动之后可以通过和上面类似的方式使用 TensorFlow。

```
$ nvidia-docker run -it -p 8888:8888 -p 6006:6006 \
cargo.caicloud.io/tensorflow/tensorflow:1.0.0-gpu
```

除了 Docker 安装，在本地使用最方便的 TensorFlow 安装方式是 pip。通过以下命令可以在 Linux 环境下使用 pip 安装 TensorFlow 1.0.0。

```
$ sudo apt-get install python-pip python-dev  # 安装pip和Python 2.7
$ sudo pip install tensorflow                 # 安装只支持CPU的TensorFlow
$ sudo pip install tensorflow-gpu             # 安装支持GPU的TensorFlow
```

目前只有在安装了 CUDA toolkit 8.0 和 CuDNN v5.1 的 64 位 Ubuntu 下可以通过 pip 安装支持 GPU 的 TensorFlow，对于其他系统或者其他 CUDA/CuDNN 版本的用户则需要从源码进行安装来支持 GPU 使用。从源码安装 TensorFlow 可以参考 https://www.tensorflow.org/install/。

TensorFlow 样例

TensorFlow 对 Python 语言的支持是最全面的，所以本文中将使用 Python 来编写 TensorFlow 程序。下面的程序给出一个简单的 TensorFlow 样例程序来实现两个向量求和。

```
import tensorflow as tf
a = tf.constant([1.0, 2.0], name="a")
b = tf.constant([2.0, 3.0], name="b")
result = a + b
print result        # 输出“Tensor("add:0", shape=(2,), dtype=float32) ”
sess = tf.Session()
print sess.run(result)    # 输出“[ 3.  5.]”
sess.close()
```

TensorFlow 基本概念

TensorFlow 的名字中已经说明了它最重要的两个概念——Tensor 和 Flow。Tensor 就是张量。在 TensorFlow 中，所有的数据都通过张量的形式来表示。从功能的角度上看，张量可以被简单理解为多维数组。但张量在 TensorFlow 中的实现并不是直接采用数组的形式，它只是对 TensorFlow 中运算结果的引用。在张量中并没有真正保存数字，它保存的是如何得到这些数字的计算过程。在上面给出的测试样例程序中，第一个 print 输出的只是一个引用而不是计算结果。

一个张量中主要保存了三个属性：名字（name）、维度（shape）和类型（type）。张量的第一个属性名字不仅是一个张量的唯一标识符，它同样也给出了这个张量是如何计算出来的。张量的命名是通过“node:src_output”的形式来给出。其中 node 为计算节点的名称，src_output 表示当前张量来自节点的第几个输出。

比如张量“add:0”就说明了 result 这个张量是计算节点“add”输出的第一个结果（编号从 0 开始）。张量的第二个属性是张量的维度（shape）。这个属性描述了一个张量的维度信息。比如“shape=(2,) ”说明了张量 result 是一个一维数组，这个数组的长度为 2。张量的第三个属性是类型（type），每一个张量会有一个唯一的类型。TensorFlow 会对参与运算的所有张量进行类型的检查，当发现类型不匹配时会报错。

如果说 TensorFlow 的第一个词 Tensor 表明了它的数据结构，那么 Flow 则体现了它的计算模型。Flow 翻译成中文就是“流”，它直观地表达了张量之间通过计算相互转化的过程。

TensorFlow 是一个通过计算图的形式来表述计算的编程系统。TensorFlow 中的每一个计算都是计算图上的一个节点，而节点之间的边描述了计算之间的依赖关系。图 1 展示了通过 TensorBoard 画出来的测试样例的计算图。

图 1 中的每一个节点都是一个运算，而每一条边代表了计算之间的依

赖关系。如果一个运算的输入依赖于另一个运算的输出，那么这两个运算有依赖关系。在图 1 中，a 和 b 这两个常量不依赖任何其他计算。而 add 计算则依赖读取两个常量的取值。于是在图 1 中可以看到有一条从 a 到 add 的边和一条从 b 到 add 的边。在图 1 中，没有任何计算依赖 add 的结果，于是代表加法的节点 add 没有任何指向其他节点的边。所有 TensorFlow 的程序都可以通过类似图 1 所示的计算图的形式来表示，这就是 TensorFlow 的基本计算模型。

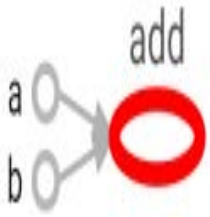


图1 通过TensorBoard可视化测试样例的计算图

TensorFlow 计算图定义完成后，我们需要通过会话（Session）来执行定义好的运算。会话拥有并管理 TensorFlow 程序运行时的所有资源。当所有计算完成之后需要关闭会话来帮助系统回收资源，否则就可能出现资源泄漏的问题。TensorFlow 可以通过 Python 的上下文管理器来使用会话。以下代码展示了如何使用这种模式。

```
# 创建一个会话，并通过Python中的上下文管理器来管理这个会话。
with tf.Session() as sess
# 使用这创建好的会话来计算关心的结果。
sess.run(...)
# 不需要再调用“Session.close()”函数来关闭会话，
# 当上下文退出时会话关闭和资源释放也自动完成了。
```

通过 Python 上下文管理器的机制，只要将所有的计算放在“with”的内部就可以。当上下文管理器退出时候会自动释放所有资源。这样既解决了因为异常退出时资源释放的问题，同时也解决了忘记调用 Session.close 函数而产生的资源泄。

TensorFlow 实现前向传播

为了介绍神经网络的前向传播算法，需要先了解神经元的结构。神经元是构成一个神经网络的最小单元，图 2 显示了一个神经元的结构。

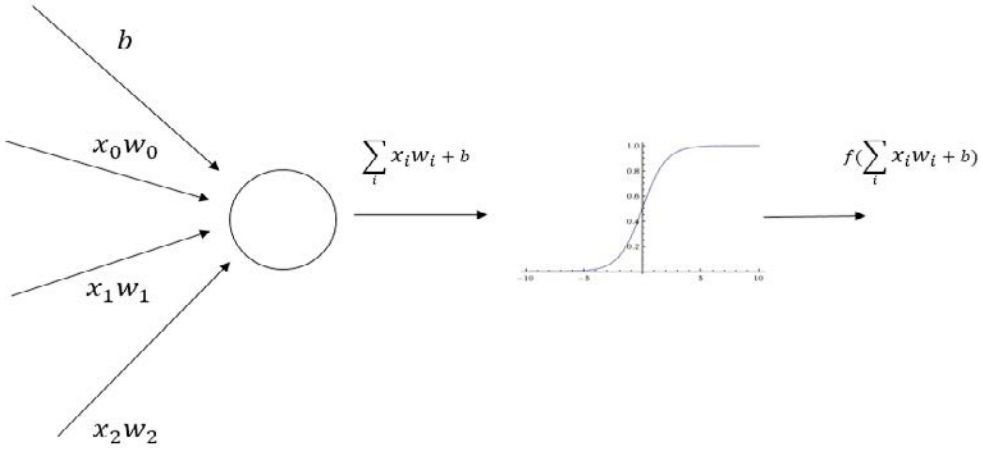


图 2 神经元结构示意图

从图 2 可以看出，一个神经元有多个输入和一个输出。每个神经元的输入既可以是其他神经元的输出，也可以是整个神经网络的输入。所谓神经网络的结构就是指的不同神经元之间的连接结构。如图 2 所示，神经元结构的输出是所有输入的加权和加上偏置项再经过一个激活函数。图 3 给出了一个简单的三层全连接神经网络。之所以称之为全连接神经网络是因为相邻两层之间任意两个节点之间都有连接。这也是为了将这样的网络结构和后面文章中将要介绍的卷积层、LSTM 结构区分。图 3 中除了输入层之外的所有节点都代表了一个神经元的结构。本小节将通过这个样例来解

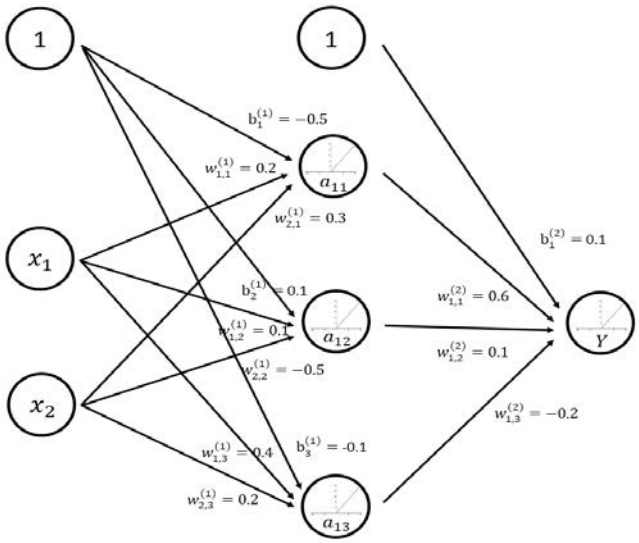


图 3 三层全连接神经网络结构图

释前向传播的整个过程。

计算神经网络的前向传播结果需要三部分信息。第一个部分是神经网络的输入，这个输入就是从实体中提取的特征向量。第二个部分为神经网络的连接结构。神经网络是由神经元构成的，神经网络的结构给出不同神经元之间输入输出的连接关系。神经网络中的神经元也可以称之为节点。在图 3 中，a11 节点有两个输入，他们分别是 x1 和 x2 的输出。而 a11 的输出则是节点 Y 的输入。最后一个部分是每个神经元中的参数。图 3 用 w 来表示神经元中的权重，b 表示偏置项。W 的上标表明了神经网络的层数，比如 W(1) 表示第一层节点的参数，而 W(2) 表示第二层节点的参数。W 的下标表明了连接节点编号，比如 W1,2(1) 表示连接 x1 和 a12 节点的边上的权重。给定神经网络的输入、神经网络的结构以及边上权重，就可以通过前向传播算法来计算出神经网络的输出。下面公式给出了在 ReLU 激活函数下图 3 神经网络前向传播的过程。

```
a11=f(W1,1(1)x1+W2,1(1)x2+b1(1))=f(0.7×0.2+0.9×0.3+(-0.5))=f(-0.09)=0
a12=f(W1,2(1)x1+W2,2(1)x2+b2(1))=f(0.7×0.1+0.9×(-0.5)+0.1)=f(-0.28)=0
a13=f(W1,3(1)x1+W2,3(1)x2+b3(1))=f(0.7×0.4+0.9×0.2+(-0.1))=f(0.36)=0.36
Y=f(W1,1(2)a11+W1,2(2)a12+W1,3(2)a13+b1(2))=f(0.054+0.028+(-
0.072)+0.1)=f(0.11)=0.11
```

在 TensorFlow 中可以通过矩阵乘法的方法实现神经网络的前向传播过程。

```
a = tf.nn.relu(tf.matmul(x, w1)+b1)
y = tf.nn.relu(tf.matmul(a, w2)+b2)
```

在上面的代码中并没有定义 w1、w2、b1、b2，TensorFlow 可以通过变量（tf.Variable）来保存和更新神经网络中的参数。比如通过下面语句可以定义 w1：

```
weights = tf.Variable(tf.random_normal([2, 3], stddev=2))
```

这段代码调用了 TensorFlow 变量的声明函数 tf.Variable。在变量声明函数中给出了初始化这个变量的方法。TensorFlow 中变量的初始值可以设置成随机数、常数或者是通过其他变量的初始值计算得到。在上面的样例

中，tf.random_normal([2, 3], stddev=2) 会产生一个 2×3 的矩阵，矩阵中的元素是均值为 0，标准差为 2 的随机数。tf.random_normal 函数可以通过参数 mean 来指定平均值，在没有指定时默认为 0。通过满足正太分布的随机数来初始化神经网络中的参数是一个非常常用的方法。下面的样例介绍了如何通过变量实现神经网络的参数并实现前向传播的过程。

```
import tensorflow as tf
# 声明变量。
w1 = tf.Variable(tf.random_normal([2, 3], stddev=1, seed=1))
b1 = tf.Variable(tf.constant(0.0, shape=[3]))
w2 = tf.Variable(tf.random_normal([3, 1], stddev=1, seed=1))
b2 = tf.Variable(tf.constant(0.0, shape=[1]))
# 暂时将输入的特征向量定义为一个常量。注意这里x是一个1*2的矩阵。
x = tf.constant([[0.7, 0.9]])
# 实现神经网络的前向传播过程，并计算神经网络的输出。
a = tf.nn.relu(tf.matmul(x, w1)+b1)
y = tf.nn.relu(tf.matmul(a, w2)+b2)

sess = tf.Session()
# 运行变量初始化过程。
init_op = tf.global_variables_initializer()
sess.run(init_op)
# 输出[[3.95757794]]
print(sess.run(y))
sess.close()
```

TensorFlow 实现反向传播

在前向传播的样例程序中，所有变量的取值都是随机的。在使用神经网络解决实际的分类或者回归问题时需要更好地设置参数取值。使用监督学习的方式设置神经网络参数需要有一个标注好的训练数据集。以判断零件是否合格为例，这个标注好的训练数据集就是收集的一批合格零件和一批不合格零件。监督学习最重要的思想就是，在已知答案的标注数据集上，

模型给出的预测结果要尽量接近真实的答案。通过调整神经网络中的参数对训练数据进行拟合，可以使得模型对未知的样本提供预测的能力。

在神经网络优化算法中，最常用的方法是反向传播算法（backpropagation）。图 4 展示了使用反向传播算法训练神经网络的流程图。本文将不过多讲解反向传播的数学公式，而是重点介绍如何通过 TensorFlow 实现反向传播的过程。

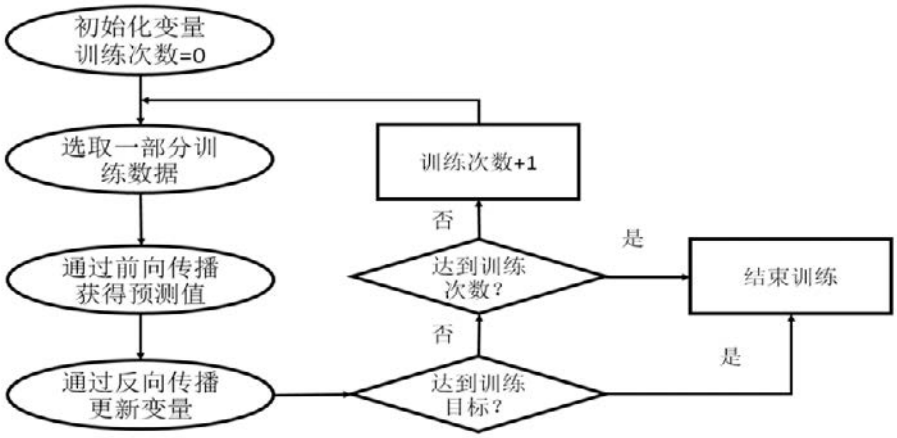


图 4 使用反向传播优化神经网络的流程图

从图 4 中可以看出，通过反向传播算法优化神经网络是一个迭代的过程。在每次迭代的开始，首先需要选取一小部分训练数据，这一小部分数据叫做一个 batch。然后，这个 batch 的样例会通过前向传播算法得到神经网络模型的预测结果。因为训练数据都是有正确答案标注的，所以可以计算出当前神经网络模型的预测答案与正确答案之间的差距。最后，基于这预测值和真实值之间的差距，反向传播算法会相应更新神经网络参数的取值，使得在这个 batch 上神经网络模型的预测结果和真实答案更加接近。通过 TensorFlow 实现反向传播算法的第一步是使用 TensorFlow 表达一个 batch 的数据。在上面的样例中使用了常量来表达过一个样例：

```
x = tf.constant([[0.7, 0.9]])
```

但如果每轮迭代中选取的数据都要通过常量来表示，那么 TensorFlow 的计算图将会太大。因为每生成一个常量，TensorFlow 都会在计算图中

增加一个节点。一般来说，一个神经网络的训练过程会需要经过几百万轮甚至几亿轮的迭代，这样计算图就会非常大，而且利用率很低。为了避免这个问题，TensorFlow 提供了 placeholder 机制用于提供输入数据。placeholder 相当于定义了一个位置，这个位置中的数据在程序运行时再指定。这样在程序中就不需要生成大量常量来提供输入数据，而只需要将数据通过 placeholder 传入 TensorFlow 计算图。在 placeholder 定义时，这个位置上的数据类型是需要指定的。和其他张量一样，placeholder 的类型也是不可以改变的。placeholder 中数据的维度信息可以根据提供的数据推导得出，所以不一定要给出。下面给出了通过 placeholder 实现前向传播算法的代码。

```
x = tf.placeholder(tf.float32, shape=(1, 2), name="input")
# 其他部分定义和上面的样例一样。
print(sess.run(y, feed_dict={x: [[0.7,0.9]]}))
```

在调用 sess.run 时，我们需要使用 feed_dict 来设定 x 的取值。在得到一个 batch 的前向传播结果之后，需要定义一个损失函数来刻画当前的预测值和真实答案之间的差距。然后通过反向传播算法来调整神经网络参数的取值使得差距可以被缩小。损失函数将在后面的文章中更加详细地介绍。以下代码定义了一个简单的损失函数，并通过 TensorFlow 定义了反向传播的算法。

```
# 定义损失函数来刻画预测值与真实值得差距。
cross_entropy = -tf.reduce_mean(
    y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))
# 定义学习率。
learning_rate = 0.001
# 定义反向传播算法来优化神经网络中的参数。
train_step =
    tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

在上面的代码中，cross_entropy 定义了真实值和预测值之间的交叉熵（cross entropy），这是分类问题中一个常用的损失函数。第二行 train_step 定义了反向传播的优化方法。目前 TensorFlow 支持 10 种不同的优化

器，读者可以根据具体的应用选择不同的优化算法。比较常用的优化方法有三种：tf.train.GradientDescentOptimizer、class tf.train.AdamOptimizer 和 tf.train.MomentumOptimizer。

TensorFlow 解决 MNIST 问题

MNIST 是一个非常有名的手写体数字识别数据集，在很多资料中，这个数据集都会被用作深度学习的入门样例。MNIST 数据集是 NIST 数据集的一个子集，它包含了 60000 张图片作为训练数据，10000 张图片作为测试数据。在 MNIST 数据集中的每一张图片都代表了 0-9 中的一个数字。图片的大小都为 28×28，且数字都会出现在图片的正中间。图 5 展示了一张数字图片及和它对应的像素矩阵：

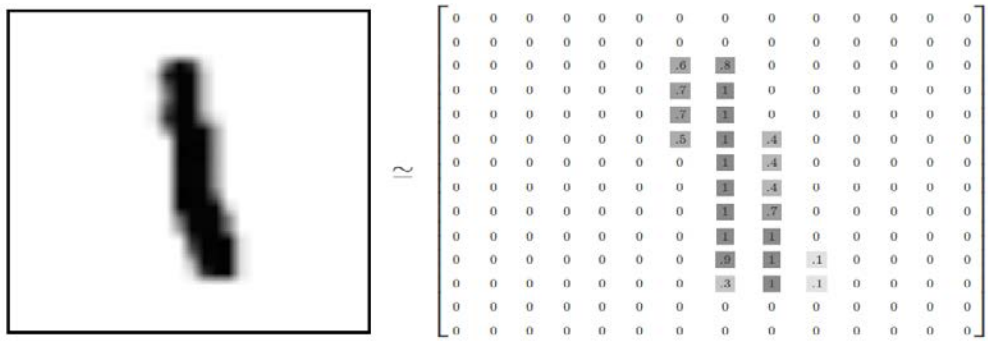


图 5 MNIST 数字图片及其像素矩阵

在图 5 的左侧显示了一张数字 1 的图片，而右侧显示了这个图片所对应的像素矩阵。MNIST 数据集中图片的像素矩阵大小为 28×28，但为了更清楚的展示，图 5 右侧显示的为 14×14 的矩阵。在 Yann LeCun 教授的网站中（http://yann.lecun.com/exdb/mnist）对 MNIST 数据集做出了详细的介绍。TensorFlow 对 MNIST 数据集做了更高层的封装，使得使用起来更加方便。下面给出了样例 TensorFlow 代码来解决 MNIST 数字手写体分类问题。

```
import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data
```

```
# MNIST数据集相关的常数。
INPUT_NODE = 784      # 输入层的节点数。对于MNIST数据集，这个就等于图片的像素。
OUTPUT_NODE = 10      # 输出层的节点数。这个等于类别的数目。因为在MNIST数据集中
                      # 需要区分的是0~9这10个数字，所以这里输出层的节点数为10。

# 配置神经网络的参数。
LAYER1_NODE = 500    # 隐藏层节点数。这里使用只有一个隐藏层的网络结构作为样例。
                     # 这个隐藏层有500个节点。
BATCH_SIZE = 100     # 一个训练batch中的训练数据个数。数字越小时，训练过程越接近
                     # 随机梯度下降；数字越大时，训练越接近梯度下降。
LEARNING_RATE = 0.01           # 学习率。
TRAINING_STEPS = 10000         # 训练轮数。

# 训练模型的过程。
def train(mnist):
    x = tf.placeholder(tf.float32, [None, INPUT_NODE], name='x-input')
    y_ = tf.placeholder(tf.float32, [None, OUTPUT_NODE], name='y-input')

# 定义神经网络参数。
weights1 = tf.Variable(
    tf.truncated_normal([INPUT_NODE, LAYER1_NODE], stddev=0.1))
bias1 = tf.Variable(tf.constant(0.0, shape=[LAYER1_NODE]))
weights2 = tf.Variable(
    tf.truncated_normal([LAYER1_NODE, OUTPUT_NODE], stddev=0.1))
bias2 = tf.Variable(tf.constant(0.0, shape=[OUTPUT_NODE]))

# 计算在当前参数下神经网络前向传播的结果。
    layer1 = tf.nn.relu(tf.matmul(input_tensor, weights1) + bias1)
    y = tf.matmul(layer1, weights2) + bias2

# 定义存储训练轮数的变量。
    global_step = tf.Variable(0, trainable=False)

# 计算交叉熵作为刻画预测值和真实值之间差距的损失函数。
```

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
    labels=y_, logits=y)
    loss = tf.reduce_mean(cross_entropy)

    # 使用tf.train.GradientDescentOptimizer优化算法来优化损失函数。注意这里损失
    # 函数包含了交叉熵损失和L2正则化损失。
    train_op=tf.train.GradientDescentOptimizer(LEARNING_RATE)\
                .minimize(loss, global_step=global_step)

# 检验神经网络的正确率。
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # 初始化会话并开始训练过程。
    with tf.Session() as sess:
    tf.initialize_all_variables().run()
    # 准备验证数据。一般在神经网络的训练过程中会通过验证数据来大致判断停止的
    # 条件和评判训练的效果。
        validate_feed = {x: mnist.validation.images,
                             y_: mnist.validation.labels}

    # 准备测试数据。在真实的应用中，这部分数据在训练时是不可见的，这个数据只是作为
    # 模型优劣的最后评价标准。
        test_feed = {x: mnist.test.images, y_: mnist.test.labels}

        # 迭代地训练神经网络。
        for i in range(TRAINING_STEPS):
            # 每1000轮输出一次在验证数据集上的测试结果。
            if i % 1000 == 0:
    validate_acc = sess.run(accuracy, feed_dict=validate_feed)
                print("After %d training step(s), validation accuracy "
                      "using average model is %g " % (i, validate_acc))

     # 产生这一轮使用的一个batch的训练数据，并运行训练过程。
```

```
            xs, ys = mnist.train.next_batch(BATCH_SIZE)
            sess.run(train_op, feed_dict={x: xs, y_: ys})

    # 在训练结束之后，在测试数据上检测神经网络模型的最终正确率。
    test_acc = sess.run(accuracy, feed_dict=test_feed)
    print("After %d training step(s), test accuracy using average "
          "model is %g" % (TRAINING_STEPS, test_acc))

# 主程序入口
def main(argv=None):
    # 声明处理MNIST数据集的类，这个类在初始化时会自动下载数据。
    mnist = input_data.read_data_sets("/tmp/data", one_hot=True)
    train(mnist)

# TensorFlow提供的一个主程序入口，tf.app.run会调用上面定义的main函数。
if __name__ == '__main__':
tf.app.run()
```

运行上面代码可以得到结果：

```
After 0 training step(s), validation accuracy using average model is 0.103
After 1000 training step(s), validation accuracy using average model is 0.9044
After 2000 training step(s), validation accuracy using average model is 0.9174
After 3000 training step(s), validation accuracy using average model is 0.9258
After 4000 training step(s), validation accuracy using average model is 0.93
After 5000 training step(s), validation accuracy using average model is 0.9346
After 6000 training step(s), validation accuracy using average model is 0.94
After 7000 training step(s), validation accuracy using average model is 0.9422
After 8000 training step(s), validation accuracy using average model is 0.9472
After 9000 training step(s), validation accuracy using average model is 0.9498
After 10000 training step(s), test accuracy using average model is 0.9475
```

通过该程序可以将 MNIST 数据集的准确率达到 ~95%。

训练神经网络模型的常用方法

本文将介绍优化训练神经网络模型的一些常用方法，并给出使用 TensorFlow 实现深度学习的最佳实践样例代码。为了更好的介绍优化神经网络训练过程，我们将首先介绍优化神经网络的算法——梯度下降算法。然后在后面的部分中，我们将围绕该算法中的一些元素来优化模型训练过程。

梯度下降算法

梯度下降算法主要用于优化单个参数的取值，而反向传播算法给出了一个高效的方式在所有参数上使用梯度下降算法，从而使神经网络模型在训练数据上的损失函数尽可能小。反向传播算法是训练神经网络的核心算

法，它可以根据定义好的损失函数优化神经网络中参数的取值，从而使神经网络模型在训练数据集上的损失函数达到一个较小值。神经网络模型中参数的优化过程直接决定了模型的质量，是使用神经网络时非常重要的一步。

假设用 θ 表示神经网络中的参数，J(θ) 表示在给定的参数取值下，训练数据集上损失函数的大小，那么整个优化过程可以抽象为寻找一个参数 θ，使得 J(θ) 最小。因为目前没有一个通用的方法可以对任意损失函数直接求解最佳的参数取值，所以在实践中，梯度下降算法是最常用的神经网络优化方法。梯度下降算法会迭代式更新参数 θ，不断沿着梯度的反方向让参数朝着总损失更小的方向更新。图 1 展示了梯度下降算法的原理。

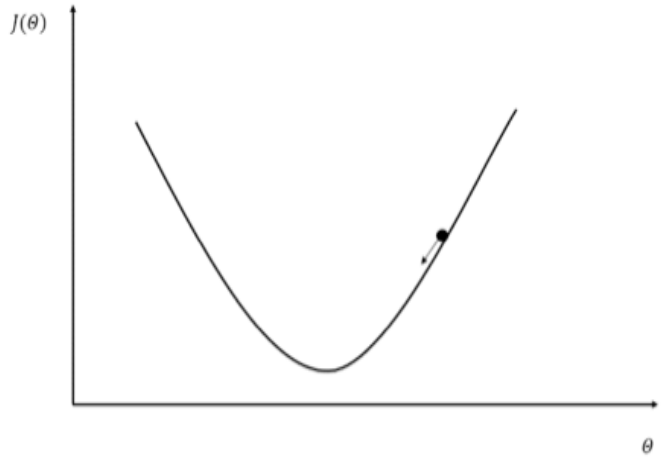


图1 梯度下降算法思想示意图

图 1 中 x 轴表示参数 θ 的取值，y 轴表示损失函数 J(θ) 的值。图 1 的曲线表示了在参数 θ 取不同值时，对应损失函数 J(θ) 的大小。假设当前的参数和损失值对应图 1 中小圆点的位置，那么梯度下降算法会将参数向 x 轴左侧移动，从而使得小圆点朝着箭头的方向移动。参数的梯度可以通过求偏导的方式计算，对于参数 θ，其梯度为 ∂/∂θ J(θ)。有了梯度，还需要定义一个学习率 η（learning rate）来定义每次参数更新的幅度。从直观上理解，可以认为学习率定义的就是每次参数移动的幅度。通过参数的梯度和学习率，参数更新的公式见下页。

下面给出了一个具体的例子来说明梯度下降算法是如何工作的。

$$\Theta_{n+1} = \Theta_n - \eta \frac{\partial}{\partial \Theta_n} J(\Theta_n)$$

假设要通过梯度下降算法来优化参数 x，使得损失函数 J(x)=x^2 的值尽量小。梯度下降算法的第一步需要随机产生一个参数 x 的初始值，然后再通过梯度和学习率来更新参数 x 的取值。在这个样例中，参数 x 的梯度为 =(∂J(x))/∂x=2x，那么使用梯度下降算法每次对参数 x 的更新公式为 x_(n+1)=x_n-η_n。假设参数的初始值为 5，学习率为 0.3，那么这个优化过程可以总结为表 1。

表1 使用梯度下降算法优化函数J(x)=x^2。

轮数	当前轮参数值	梯度×学习率	更新后参数值
1	5	2*5*0.3=3	5−3=2
2	2	2*2*0.3=1.2	2−1.2=0.8
3	0.8	2*0.8*0.3=0.48	0.8−0.48=0.32
4	0.32	2*0.32*0.3=0.192	0.8−0.192=0.128
5	0.128	2*0.128*0.3=0.0768	0.8−0.0768=0.0512

从表 1 中可以看出，经过 5 次迭代之后，参数 x 的值变成了 0.0512，这个和参数最优值 0 已经比较接近了。虽然这里给出的是一个非常简单的样例，但是神经网络的优化过程也是可以类推的。神经网络的优化过程可以分为两个阶段，第一个阶段先通过前向传播算法计算得到预测值，并将预测值和真实值做对比得出两者之间的差距。然后在第二个阶段通过反向传播算法计算损失函数对每一个参数的梯度，再根据梯度和学习率使用梯度下降算法更新每一个参数。本书将略去反向传播算法具体的实现方法和数学证明，有兴趣的读者可以参考论文 Learning representations by back-propagating errors。

因为梯度下降算法要在全部训练数据上最小化损失，所以损失函数 J(Θ) 是在所有训练数据上的损失和。这样在每一轮迭代中都需要计算在全部训练数据上的损失函数。在海量训练数据下，要计算所有训练数据的损失函数是非常消耗时间的。为了加速训练过程，可以使用随机梯度下降的算法（stochastic gradient descent）。这个算法优化的不是在全部训练数据上的损失函数，而是在每一轮迭代中，随机优化某一条训练数据上的损失

函数。这样每一轮参数更新的速度就大大加快了。因为随机梯度下降算法每次优化的只是某一条数据上的损失函数，所以它的问题也非常明显：在某一条数据上损失函数更小并不代表在全部数据上损失函数更小，于是使用随机梯度下降优化得到的神经网络甚至可能无法达到局部最优。

为了综合梯度下降算法和随机梯度下降算法的优缺点，在实际应用中一般采用这两个算法的折中——每次计算一小部分训练数据的损失函数。这一小部分数据被称之为一个 batch。通过矩阵运算，每次在一个 batch 上优化神经网络的参数并不会比单个数据慢太多。另一方面，每次使用一个 batch 可以大大减小收敛所需要的迭代次数，同时可以使收敛到的结果更加接近梯度下降的效果。

学习率的设置

上面提到在优化神经网络时，需要设置学习率（learning rate）控制参数更新的速度。学习率决定了参数每次更新的幅度。如果幅度过大，那么可能导致参数在极优值的两侧来回移动。还是以优化 J(x)=x2 函数为样例。如果在优化中使用的学习率为 1，那么整个优化过程将会如表 2 所示。

表2 当学习率过大时，梯度下降算法的运行过程

轮数	当前轮参数值	梯度学习率	更新后参数值
1	5	2*5*1=10	5−10=−5
2	−5	2*−5*1=−10	−5−(−10)=5
3	5	2*5*1=10	5−10=−5

从上面的样例可以看出，无论进行多少轮迭代，参数将在 5 和 -5 之间摇摆，而不会收敛到一个极小值从上面的样例可以看出，无论进行多少轮迭代，参数将在 5 和 -5 之间摇摆，而不会收敛到一个极小值。相反，当学习率过小时，虽然能保证收敛性，但是这会大大降低优化速度。我们会需要更多轮的迭代才能达到一个比较理想的优化效果。比如当学习率为 0.001 时，迭代 5 次之后，x 的值将为 4.95。要将 x 训练到 0.05 需要大约 2300 轮；而当学习率为 0.3 时，只需要 5 轮就可以达到。综上所述，学习率既不能过大，也不能过小。为了解决设定学习率的问题，TensorFlow 提供了一种更加灵活的学习率设置方法——指数衰减法。

tf.train.exponential_decay 函数实现了指数衰减学习率。通过这个函数，可以先使用较大的学习率来快速得到一个比较优的解，然后随着迭代的继续逐步减小学习率，使得模型在训练后期更加稳定。exponential_decay 函数会指数级地减小学习率，它实现了以下代码的功能：

```
decayed_learning_rate =
    learning_rate * decay_rate ^ (global_step / decay_ steps)
```

其中 decayed_learning_rate 为每一轮优化时使用的学习率，learning_rate 为事先设定的初始学习率，decay_rate 为衰减系数，decay_steps 为衰减速度。下面给出了一段代码来示范如何在 TensorFlow 中使用 tf.train.exponential_decay 函数。

```
# 通过exponential_decay函数生成学习率。
learning_rate = tf.train.exponential_decay(
learning_rate_base, global_step, decay_step, decay_rate)


 # 使用指数衰减的学习率。在minimize函数中传入global_step将自动更新
 # global_step参数，从而使得学习率也得到相应更新。
learning_step =
    tf.train.GradientDescentOptimizer(learning_rate)\
      .minimize(...my loss..., global_step=global_step)
```

过拟合问题

在使用梯度下降优化神经网络时，被优化的函数就是神经网络的损失函数。这个损失函数刻画了在训练数据集上预测结果和真实结果之间的差距。然而在真实的应用中，我们想要的并不是让模型尽量模拟训练数据的行为，而是希望通过训练出来的模型对未知的数据给出判断。模型在训练数据上的表现并不一定代表了它在未知数据上的表现。过拟合问题就是可以导致这个差距的一个很重要因素。所谓过拟合，指的是当一个模型过为复杂之后，它可以很好地“记忆”每一个训练数据中随机噪音的部分而忘记了要去“学习”训练数据中通用的趋势。举一个极端的例子，如果一个模型中的参数比训练数据的总数还多，那么只要训练数据不冲突，这个模型完

全可以记住所有训练数据的结果从而使得损失函数为 0。可以直观地想象一个包含 n 个变量和 n 个等式的方程组，当方程不冲突时，这个方程组是可以通过数学的方法来求解的。然而，过度拟合训练数据中的随机噪音虽然可以得到非常小的损失函数，但是对于未知数据可能无法做出可靠的判断。

图 2 显示了模型训练的三种不同情况。在第一种情况下，由于模型过于简单，无法刻画问题的趋势。第二个模型是比较合理的，它既不会过于关注训练数据中的噪音，又能够比较好地刻画问题的整体趋势。第三个模型就是过拟合了，虽然第三个模型完美地划分了灰色和黑色的点，但是这样的划分并不能很好地对未知数据做出判断，因为它过度拟合了训练数据中的噪音而忽视了问题的整体规律。比如图中浅色方框更有可能和“X”属于同一类，而不是根据图上的划分和“O”属于同一类。

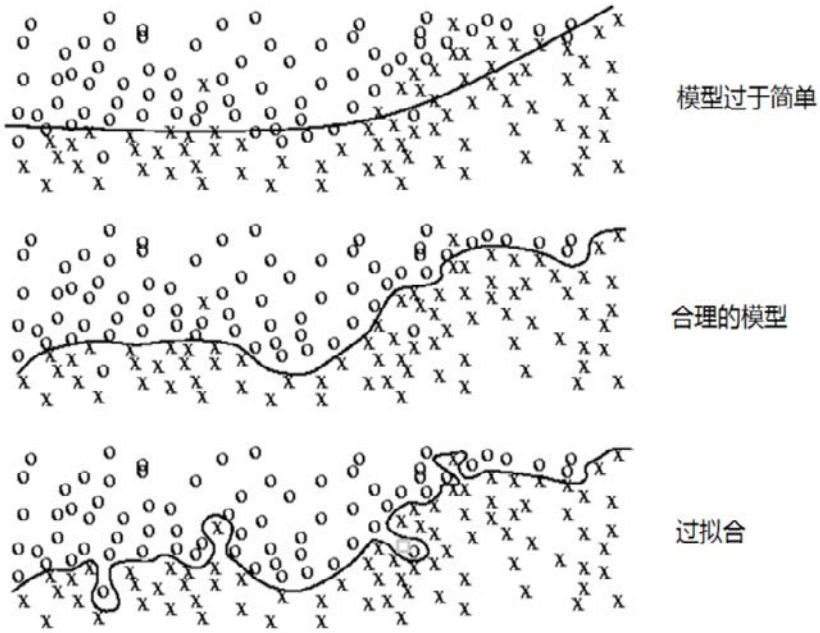


图 2 神经网络模型训练的三种情况

为了避免过拟合问题，一个非常常用的方法是正则化（regularization）。正则化的思想就是在损失函数中加入刻画模型复杂程度的指标。假设用于刻画模型在训练数据上表现的损失函数为 J(Θ)，那么在优化时不是直接优

化 J(θ)，而是优化 J(θ)+λR(w)。其中 R(w) 刻画的是模型的复杂程度，而 λ 表示模型复杂损失在总损失中的比例。注意这里 θ 表示的是一个神经网络中所有的参数，它包括边上的权重 w 和偏置项 b。一般来说模型复杂度只由权重 w 决定。常用的刻画模型复杂度的函数 R(w) 有两种，一种是 L1 正则化，计算公式是：

无论是哪一种正则化方式，基本的思想都是希望通过限制权重的大小，使得模型不能任意拟合训练数据中的随机噪音。但这两种正则化的方法也有很大的区别。首先，L1 正则化会让参数变得更稀疏，而 L2 正则化不会。所谓参数变得更稀疏是指会有更多的参数变为 0，这样可以达到类似特征选取的功能。之所以 L2 正则化不会让参数变得稀疏的原因是当参数很小时，比如 0.001，这个参数的平方基本上就可以忽略了，于是模型不会进一步将这个参数调整为 0。其次，L1 正则化的计算公式不可导，而 L2 正则化公式可导。因为在优化时需要计算损失函数的偏导数，所以对含有 L2 正则化损失函数的优化要更加简洁。优化带 L1 正则化的损失函数要更加复杂，而且优化方法也有很多种。以下代码给出了一个简单的带 L2 正则化的损失函数定义：

```
w= tf.Variable(tf.random_normal([2, 1], stddev=1, seed=1))
y = tf.matmul(x, w)

loss = tf.reduce_mean(tf.square(y_ - y)) +
        tf.contrib.layers.l2_regularizer(lambda)(w)
```

滑动平均模型

在采用随机梯度下降算法训练神经网络时，使用滑动平均模型在很多应用中都可以在一定程度提高最终模型在测试数据上的表现。滑动平均模型可以有效的减小训练数据中的噪音对模型带来的影响。在 TensorFlow 中提供了 tf.train.ExponentialMovingAverage 来实现滑动平均模型。在初始化 ExponentialMovingAverage 时，需要提供一个衰减率（decay）。这个

衰减率将用于控制模型更新的速度。ExponentialMovingAverage 对每一个变量会维护一个影子变量（shadow variable），这个影子变量的初始值就是相应变量的初始值，而每次运行变量更新时，影子变量的值会更新为：

```
shadow_variable = decay × shadow_variable + (1 - decay)× variable
```

其中 shadow_variable 为影子变量，variable 为待更新的变量，decay 为衰减率。从公式中可以看到，decay 决定了模型更新的速度，decay 越大模型越趋于稳定。在实际应用中，decay 一般会设成非常接近 1 的数（比如 0.999 或 0.9999）。为了使得模型在训练前期可以更新得更快，ExponentialMovingAverage 还提供了 num_updates 参数来动态设置 decay 的大小。如果在 ExponentialMovingAverage 初始化时提供了 num_updates 参数，那么每次使用的衰减率将是：

$$\min\{\text{decay}, \frac{1 + \text{num\_updates}}{10 + \text{num\_updates}}\}$$

下面通过一段代码来解释 ExponentialMovingAverage 是如何被使用的。

```
import tensorflow as tf
v1 = tf.Variable(0.0, dtype=tf.float32)
step = tf.Variable(0, trainable=False)
# 定义一个滑动平均的类（class）。初始化时给定了衰减率（0.99）和控制衰减率的变量step。
ema = tf.train.ExponentialMovingAverage(0.99, step)
# 定义一个更新变量滑动平均的操作。这里需要给定一个列表，每次执行这个操作时
# 这个列表中的变量都会被更新。
maintain_averages_op = ema.apply([v1])
with tf.Session() as sess:
    init_op = tf.initialize_all_variables()
sess.run(init_op)
    # 更新变量v1的值到5。
```

```
sess.run(tf.assign(v1, 5))
    # 更新v1的滑动平均值。衰减率为min{0.99,(1+step)/(10+step)= 0.1}=0.1,
    # 所以v1的滑动平均会被更新为0.10+0.95=4.5。
sess.run(maintain_averages_op)
    print sess.run([v1, ema.average(v1)])        #  输出[5.0, 4.5]



    # 更新step的值为10000。
    sess.run(tf.assign(step, 10000))
    # 更新v1的值为10。
sess.run(tf.assign(v1, 10))
    # 更新v1的滑动平均值。衰减率为min{0.99,(1+step)/(10+step)  0.999}=0.99,
    # 所以v1的滑动平均会被更新为0.994.5+0.0110=4.555。
    sess.run(maintain_averages_op)
print sess.run([v1, ema.average(v1)])      #  输出[10.0, 4.5549998]
```

TensorFlow 最佳实践样例程序

将训练和测试分成两个独立的程序，这可以使得每一个组件更加灵活。比如训练神经网络的程序可以持续输出训练好的模型，而测试程序可以每隔一段时间检验最新模型的正确率，如果模型效果更好，则将这个模型提供给产品使用。除了可以将不同功能模块分开，本节还将前向传播的过程抽象成一个单独的库函数。因为神经网络的前向传播过程在训练和测试的过程中都会用到，所以通过库函数的方式使用起来既可以更加方便，又可以保证训练和测试过程中使用的前向传播方法一定是一致的。下面我们将给出 TensorFlow 模型训练的一个最佳实践，它使用了上文中提到的所有优化方法来解决 MNIST 问题。在这儿最佳实践中总共有三个程序，第一个是 mnist_inference.py，它定义了前向传播的过程以及神经网络中的参数。第二个是 mnist_train.py，它定义了神经网络的训练过程。第三个是 mnist_eval.py，它定义了测试过程。以下代码给出了 mnist_inference.py 中的内容。

```
# -*- coding: utf-8 -*-
```

```
import tensorflow as tf
# 定义神经网络结构相关的参数。
INPUT_NODE = 784
OUTPUT_NODE = 10
LAYER1_NODE = 500
# 通过tf.get_variable函数来获取变量。在训练神经网络时会创建这些变量；在测试时会通
# 过保存的模型加载这些变量的取值。而且更加方便的是，因为可以在变量加载时将滑动平均变量
# 重命名，所以可以直接通过同样的名字在训练时使用变量自身，而在测试时使用变量的滑动平
# 均值。在这个函数中，将变量的正则化损失加入损失集合。
def get_weight_variable(shape, regularizer):
   weights = tf.get_variable(
       "weights", shape,
        initializer=tf.truncated_normal_initializer(stddev=0.1))
   # 当给出了正则化生成函数时，将当前变量的正则化损失加入名字为losses的集合。在这里
   # 使用了add_to_collection函数将一个张量加入一个集合，而这个集合的名称为losses。
   # 这是自定义的集合，不在TensorFlow自动管理的集合列表中。
   if regularizer != None:
        tf.add_to_collection('losses', regularizer(weights))
   return weights
# 定义神经网络的前向传播过程。
def inference(input_tensor, regularizer):
   # 声明第一层神经网络的变量并完成前向传播过程。
   with tf.variable_scope('layer1'):
       # 这里通过tf.get_variable或tf.Variable没有本质区别，因为在训练或是测试中
       # 没有在同一个程序中多次调用这个函数。如果在同一个程序中多次调用，在第一次调用
       # 之后需要将reuse参数设置为True。
      weights = get_weight_variable(
           [INPUT_NODE, LAYER1_NODE], regularizer)
      biases = tf.get_variable(
           "biases", [LAYER1_NODE],
           initializer=tf. constant_initializer(0.0))
      layer1 = tf.nn.relu(tf.matmul(input_tensor, weights) + biases)
   # 类似的声明第二层神经网络的变量并完成前向传播过程。
   with tf.variable_scope('layer2'):
```

```
        weights = get_weight_variable(
              [LAYER1_NODE, OUTPUT_NODE], regularizer)
        biases = tf.get_variable(
             "biases", [OUTPUT_NODE],
             initializer=tf. constant_initializer(0.0))
        layer2 = tf.matmul(layer1, weights) + biases
    # 返回最后前向传播的结果。
    return layer2
```

在这段代码中定义了神经网络的前向传播算法。无论是训练时还是测试时，都可以直接调用 inference 这个函数，而不用关心具体的神经网络结构。使用定义好的前向传播过程，以下代码给出了神经网络的训练程序 mnist_train.py。

```
# -*- coding: utf-8 -*-
import os
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
# 加载mnist_inference.py中定义的常量和前向传播的函数。
import mnist_inference
# 配置神经网络的参数。
BATCH_SIZE = 100
LEARNING_RATE_BASE = 0.8
LEARNING_RATE_DECAY = 0.99
REGULARAZTION_RATE = 0.0001
TRAINING_STEPS = 30000
MOVING_AVERAGE_DECAY = 0.99
# 模型保存的路径和文件名。
MODEL_SAVE_PATH = "/path/to/model/"
MODEL_NAME = "model.ckpt"
def train(mnist):
    # 定义输入输出placeholder。
    x = tf.placeholder(
         tf.float32, [None, mnist_inference.INPUT_NODE], name='x-input')
    y_ = tf.placeholder(
```

```
        tf.float32, [None, mnist_inference.OUTPUT_NODE], name='y-input')

    regularizer = tf.contrib.layers.l2_regularizer(REGULARAZTION_RATE)
     # 直接使用mnist_inference.py中定义的前向传播过程。
    y = mnist_inference.inference(x, regularizer)
    global_step = tf.Variable(0, trainable=False)

    # 定义损失函数、学习率、滑动平均操作以及训练过程。
    variable_averages = tf.train.ExponentialMovingAverage(
        MOVING_AVERAGE_DECAY, global_step)
    variables_averages_op = variable_averages.apply(
        tf.trainable_variables())
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits=y, labels=tf.argmax(y_, 1))
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
    loss = cross_entropy_mean + tf.add_n(tf.get_collection('losses'))
    learning_rate = tf.train.exponential_decay(
        LEARNING_RATE_BASE,
         global_step,
         mnist.train.num_examples / BATCH_SIZE, LEARNING_RATE_DECAY)
    train_step = tf.train.GradientDescentOptimizer(learning_rate)\
                        .minimize(loss, global_step=global_step)
    with tf.control_dependencies([train_step, variables_averages_op]):
        train_op = tf.no_op(name='train')

     # 初始化TensorFlow持久化类。
    saver = tf.train.Saver()
    with tf.Session() as sess:
        tf.global_variables_initializer().run()

         # 在训练过程中不再测试模型在验证数据上的表现，验证和测试的过程将会有一个独
         # 立的程序来完成。
        for i in range(TRAINING_STEPS):
            xs, ys = mnist.train.next_batch(BATCH_SIZE)
```

```
            _, loss_value, step = sess.run([train_op, loss, global_step],
                                           feed_dict={x: xs, y_: ys})
            # 每1000轮保存一次模型。
            if i % 1000 == 0:
                # 输出当前的训练情况。这里只输出了模型在当前训练batch上的损失函
                # 数大小。通过损失函数的大小可以大概了解训练的情况。在验证数据集上的
                # 正确率信息会有一个单独的程序来生成。
                print("After %d training step(s), loss on training "
                      "batch is %g." % (step, loss_value))
                # 保存当前的模型。注意这里给出了global_step参数，这样可以让每个被
                # 保存模型的文件名末尾加上训练的轮数，比如“model.ckpt-1000”表示
                # 训练1000轮之后得到的模型。
                saver.save(
                    sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME),
                    global_step=global_step)

def main(argv=None):
    mnist = input_data.read_data_sets("/tmp/data", one_hot=True)
    train(mnist)

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

运行上面的程序，可以得到类似下面的结果。

```
~/mnist$ python mnist_train.py
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
After 1 training step(s), loss on training batch is 3.46893.
After 1001 training step(s), loss on training batch is 0.172291.
After 2001 training step(s), loss on training batch is 0.197483.
After 3001 training step(s), loss on training batch is 0.153582.
After 4001 training step(s), loss on training batch is 0.117219.
After 5001 training step(s), loss on training batch is 0.121872.
```

```
After 6001 training step(s), loss on training batch is 0.0976607.
```

在新的训练代码中，不再将训练和测试跑在一起。训练过程中，每1000轮输出一次在当前训练batch上损失函数的大小来大致估计训练的效果。在上面的程序中，每1000轮保存一次训练好的模型，这样可以通过一个单独的测试程序，更加方便地在滑动平均模型上做测试。以下代码给出了测试程序mnist_eval.py。

```
# -*- coding: utf-8 -*-
import time
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

# 加载mnist_inference.py和mnist_train.py中定义的常量和函数。
import mnist_inference
import mnist_train

# 每10秒加载一次最新的模型，并在测试数据上测试最新模型的正确率。
EVAL_INTERVAL_SECS = 10
def evaluate(mnist):
   with tf.Graph().as_default() as g:
       # 定义输入输出的格式。
      x = tf.placeholder(
           tf.float32, [None, mnist_inference.INPUT_NODE], name='x-input')
      y_ = tf.placeholder(
           tf.float32, [None, mnist_inference.OUTPUT_NODE], name='y-input')
      validate_feed = {x: mnist.validation.images,
                       y_:mnist.validation. labels}
       # 直接通过调用封装好的函数来计算前向传播的结果。因为测试时不关注正则化损失的值，
       # 所以这里用于计算正则化损失的函数被设置为None。
       y = mnist_inference.inference(x, None)
        # 使用前向传播的结果计算正确率。如果需要对未知的样例进行分类，那么使用
        # tf.argmax(y, 1)就可以得到输入样例的预测类别了。
       correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
       accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)
```

```
        # 通过变量重命名的方式来加载模型，这样在前向传播的过程中就不需要调用求滑动平均
        # 的函数来获取平均值了。这使得我们可以完全共用mnist_inference.py中定义的
        # 前向传播过程。
        variable_averages = tf.train.ExponentialMovingAverage(
              mnist_train.MOVING_AVERAGE_DECAY)
        variables_to_restore = variable_averages.variables_to_restore()
        saver = tf.train.Saver(variables_to_restore)
        # 每隔EVAL_INTERVAL_SECS秒调用一次计算正确率的过程以检测训练过程中正确率的# 变
化。
        while True:
            with tf.Session() as sess:
                  # tf.train.get_checkpoint_state函数会通过checkpoint文件自动
                  # 找到目录中最新模型的文件名。
                ckpt = tf.train.get_checkpoint_state(
                       mnist_train.MODEL_SAVE_PATH)
                if ckpt and ckpt.model_checkpoint_path:
                       # 加载模型。
                    saver.restore(sess, ckpt.model_checkpoint_path)
                       # 通过文件名得到模型保存时迭代的轮数。
                    global_step = ckpt.model_checkpoint_path
                                              .split('/')[-1].split('-')[-1]
                    accuracy_score = sess.run(accuracy,feed_dict=validate_feed)
                     print("After %s training step(s), validation "
                                "accuracy = %g" % (global_step, accuracy_score))
                 else:
                     print('No checkpoint file found')
                     return
            time.sleep(EVAL_INTERVAL_SECS)
    def main(argv=None):
    mnist = input_data.read_data_sets("/tmp/data", one_hot=True)
    evaluate(mnist)
if __name__ == '__main__':
    tf.app.run()
```

上面给出的 mnist_eval.py 程序会每隔 10 秒运行一次，每次运行都是

读取最新保存的模型，并在 MNIST 验证数据集上计算模型的正确率。如果需要离线预测未知数据的类别（比如这个样例程序可以判断手写体数字图片中所包含的数字），只需要将计算正确率的部分改为答案输出即可。运行 mnist_eval.py 程序可以得到类似下面的结果。注意因为这个程序每 10 秒自动运行一次，而训练程序不一定每 10 秒输出一个新模型，所以在下面的结果中会发现有些模型被测试了多次。一般在解决真实问题时，不会这么频繁地运行评测程序。

```
~/mnist$ python mnist_eval.py
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
After 1 training step(s), validation accuracy = 0.0616
After 1001 training step(s), validation accuracy = 0.9764
After 2001 training step(s), validation accuracy = 0.9834
After 2001 training step(s), validation accuracy = 0.9834
After 3001 training step(s), validation accuracy = 0.9852
After 4001 training step(s), validation accuracy = 0.9854
After 5001 training step(s), validation accuracy = 0.986
After 6001 training step(s), validation accuracy = 0.9854
```

上面的程序可以将 MNIST 正确率达到 ~98.4%。

卷积神经网络

本文为第四篇文章，将会介绍卷积神经网络，通过与传统算法的对比、卷积神经网络的结构分析等方面来介绍卷积神经网络模型，并给出通过 TensorFlow 在 MNIST 数据集上实现卷积神经网络的方法。

卷积神经网络简介

斯坦福大学（Stanford University）李飞飞（Feifei Li）教授带头整理的 ImageNet 是图像识别领域非常有名的数据集。在 ImageNet 中，将近 1500 万图片被关联到了 WordNet 的大约 20000 个名词同义词集上。ImageNet 每年都举办图像识别相关的竞赛（ImageNet Large Scale Visual Recognition Challenge，ILSVRC），其中最有名的就是 ILSVRC2012 图像

分类数据集。ILSVRC2012 图像分类数据集包含了来自 1000 个类别的 120 万张图片，其中每张图片属于且只属于一个类别。因为 ILSVRC2012 图像分类数据集中的图片是直接从互联网上爬取得到的，图片的大小从几千字节到几百万字节不等。

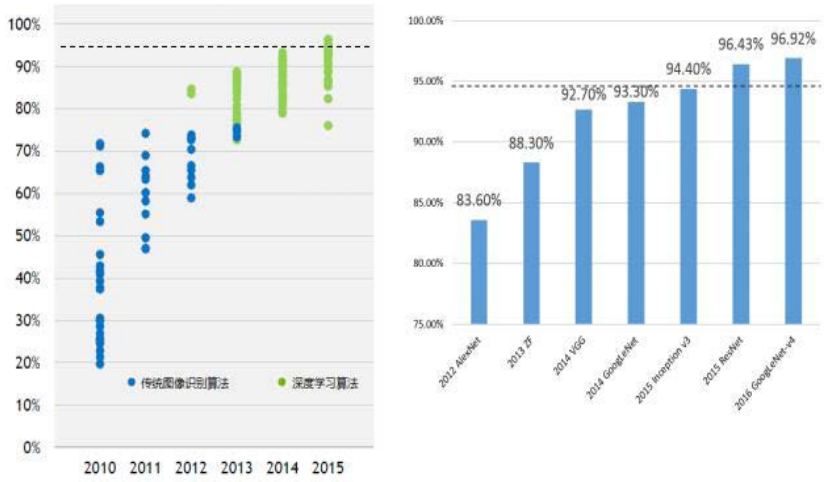


图1 不同算法在ImageNet ILSVRC2012图像分类数据集上的正确率

图 1 给出了不同算法在 ImageNet 图像分类数据集上的 top-5 正确率。top-N 正确率指的是图像识别算法给出前 N 个答案中有一个是正确的概率。在图像分类问题上，很多学术论文都将前 N 个答案的正确率作为比较的方法，其中 N 的取值一般为 3 或 5。从图 1 中可以看出，在 ImageNet 问题上，基于卷积神经网络的图像识别算法可以远远超过人类的表现。在图 1 的左侧对比了传统算法与深度学习算法的正确率。从图中可以看出，深度学习，特别是卷积神经网络，给图像识别问题带来了质的飞跃。2013 年之后，基本上所有的研究都集中到了深度学习算法上。

在前面的文章中所介绍的神经网络每两层之间的所有结点都是有边相连的，所以我们称这种网络结构为全连接层网络结构。图 2 显示了全连接神经网络与卷积神经网络的结构对比图。虽然图 2 中显示的全连接神经网络结构和卷积神经网络的结构直观上差异比较大，但实际上它们的整体架构是非常相似的。从图 2 中可以看出，卷积神经网络也是通过一层一层的节点组织起来的。和全连接神经网络一样，卷积神经网络中的每一个节点

都是一个神经元。在全连接神经网络中，每相邻两层之间的节点都有边相连，于是一般会将每一层全连接层中的节点组织成一列，这样方便显示连接结构。而对于卷积神经网络，相邻两层之间只有部分节点相连，为了展示每一层神经元的维度，一般会将每一层卷积层的节点组织成一个三维矩阵。

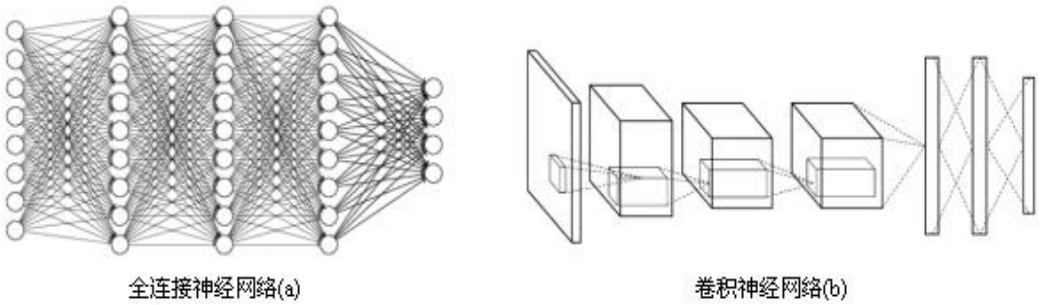


图2 全连接神经网络与卷积神经网络结构示意图

除了结构相似，卷积神经网络的输入输出以及训练流程与全连接神经网络也基本一致。以图像分类为例，卷积神经网络的输入层就是图像的原始像素，而输出层中的每一个节点代表了不同类别的可信度。这和全连接神经网络的输入输出是一致的。在 TensorFlow 中训练一个卷积神经网络的流程和训练一个全连接神经网络没有任何区别。卷积神经网络和全连接神经网络的唯一区别就在于神经网络中相邻两层的连接方式。

使用全连接神经网络处理图像的最大问题在于全连接层的参数太多。对于 MNIST 数据，每一张图片的大小是 28×28×1，其中 28×28 为图片的大小，*1 表示图像是黑白的，只有一个色彩通道。假设第一层隐藏层的节点数为 500 个，那么一个全链接层的神经网络将有 28×28×500+500=392500 个参数。当图片更大时，比如在 Cifar-10 数据集中，图片的大小为 32×32×3，其中 32×32 表示图片的大小，×3 表示图片是通过红绿蓝三个色彩通道（channel）表示的。这样输入层就有 3072 个节点，如果第一层全连接层仍然是 500 个节点，那么这一层全链接神经网络将有 3072×500+50 约等于 150 万个参数。参数增多除了导致计算速度减慢，还很容易导致过拟合问题。所以需要一个更合理的神经网络结构

来有效地减少神经网络中参数个数。卷积神经网络就可以达到这个目的。

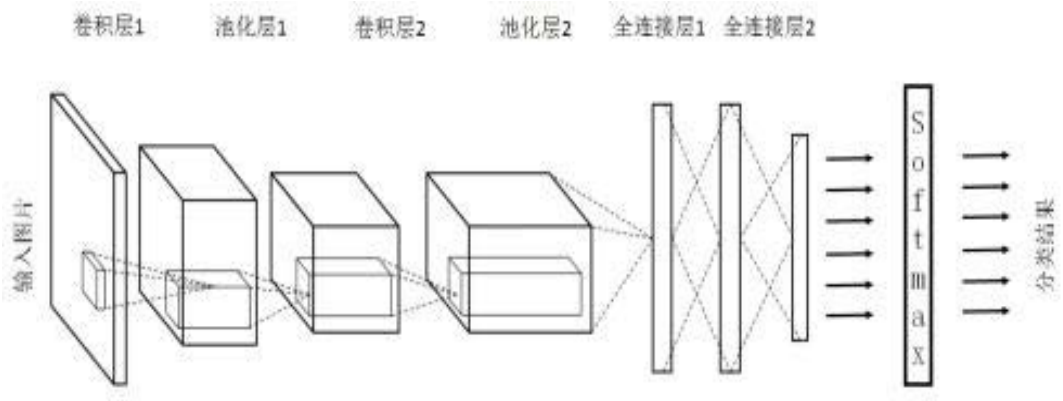


图 3 用于图像分类问题的一种卷积神经网络架构图

图 3 给出了一个更加具体的卷积神经网络架构图。在卷积神经网络的前几层中，每一层的节点都被组织成一个三维矩阵。比如处理 Cifar-10 数据集中的图片时，可以将输入层组织成一个 32*32*3 的三维矩阵。图 3 中虚线部分展示了卷积神经网络的一个连接示意图，从图中可以看出卷积神经网络中前几层中每一个节点只和上一层中部分的节点相连。卷积神经网络的具体连接方式将在下文中介绍。

卷积层网络结构

图 4 中显示了卷积层神经网络结构中最重要的部分，这个部分被称之为过滤器（filter）或者内核（kernel）。因为 TensorFlow 文档中将这个结构称之为过滤器（filter），所以我们将统称这个结构为过滤器。如图 4 所示，过滤器可以将当前层神经网络上的一个子节点矩阵转化为下一层神经网络上的一个单位节点矩阵。单位节点矩阵指的是一个长和宽都为 1，但深度不限的节点矩阵。

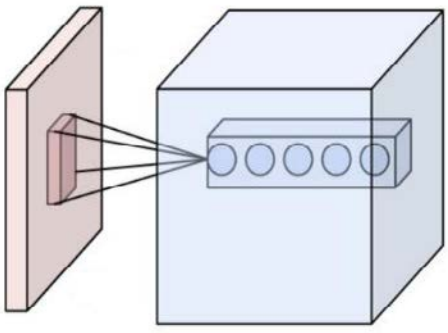


图 4 卷积层过滤器结构示意图

在一个卷积层中，过滤器所处理的节点矩阵的长和宽都是由人工指定的，这个节点矩阵的尺寸也被称之为过滤器的尺寸。常用的过滤器尺寸有3×3 或 5×5。因为过滤器处理的矩阵深度和当前层神经网络节点矩阵的深度是一致的，所以虽然节点矩阵是三维的，但过滤器的尺寸只需要指定两个维度。过滤器中另外一个需要人工指定的设置是处理得到的单位节点矩阵的深度，这个设置称为过滤器的深度。注意过滤器的尺寸指的是一个过滤器输入节点矩阵的大小，而深度指的是输出单位节点矩阵的深度。如图 4 所示，左侧小矩阵的尺寸为过滤器的尺寸，而右侧单位矩阵的深度为过滤器的深度。

如图 4 所示，过滤器的前向传播过程就是通过左侧小矩阵中的节点计算出右侧单位矩阵中节点的过程。为了直观地解释过滤器的前向传播过程，在下面的篇幅中将给出一个具体的样例。在这个样例中将展示如何通过过滤器将一个 2×2×3 的节点矩阵变化为一个 1×1×5 的单位节点矩阵。一个过滤器的前向传播过程和全连接层相似，它总共需要 2×2×3×5+5=65 个参数，其中最后的 +5 为偏置项参数的个数。假设使用 $w_{(x,y,z)}^{i}$ 来表示对于输出单位节点矩阵中的第 i 个节点，过滤器输入节点 (x,y,z) 的权重，使用 b^{i} 表示第 i 个输出节点对应的偏置项参数，那么单位矩阵中的第 i 个节点的取值 g(i) 为：

$$g(i)=f\left(\sum_{x=1}^{2}\sum_{y=1}^{2}\sum_{z=1}^{3}a_{x,y,z}\times w_{x,y,z}^{i}+b^{i}\right)$$

其中 $a_{(x,y,z)}$ 为过滤器中节点 (x,y,z) 的取值，f 为激活函数。如果将 a 和 w^{i} 组织成两个向量，那么一个过滤器的计算过程完全可以通过向量乘法来完成。卷积层结构的前向传播过程就是通过将一个过滤器从神经网络当前层的左上角移动到右下角，并且在移动中计算每一个对应的单位矩阵得到的。图 5 展示了卷积层结构前向传播的过程。为了更好地可视化过滤器的移动过程，图 5 中使用的节点矩阵深度都为 1。在图 5 中，展示了在 3×3

矩阵上使用 2×2 过滤器的卷积层前向传播过程。在这个过程中，首先将这个过滤器用于左上角子矩阵，然后移动到左下角矩阵，再到右上角矩阵，最后到右下角矩阵。过滤器每移动一次，可以计算得到一个值（当深度为 k 时会计算出 k 个值）。将这些数值拼接成一个新的矩阵，就完成了卷积层前向传播的过程。图 5 的右侧显示了过滤器在移动过程中计算得到的结果与新矩阵中节点的对应关系。

在图 5 中，只讲解了移动过滤器的方式，没有涉及到过滤器中的参数如何设定。在卷积神经网络中，每一个卷积层中使用的过滤器中的参数都是一样的。这是卷积神经网络一个非常重要的性质。从直观上理解，共享过滤器的参数可以使得图像上的内容不受位置的影响。以 MNIST 手写体数字识别为例，无论数字“1”出现在左上角还是右下角，图片的种类都是不变的。因为在左上角和右下角使用的过滤器参数相同，所以通过卷积层之后无论数字在图像上的哪个位置，得到的结果都一样。

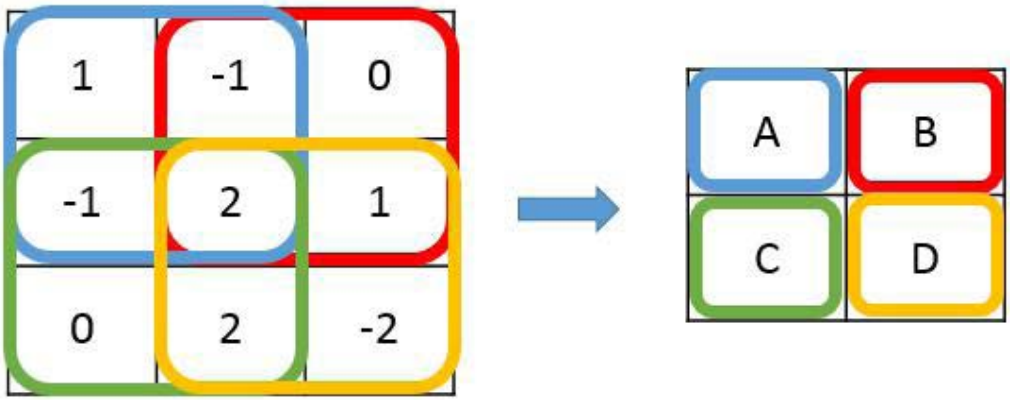


图5 卷积层前向传播过程示意图

共享每一个卷积层中过滤器中的参数可以巨幅减少神经网络上的参数。以 Cifar-10 问题为例，输入层矩阵的维度是 32×32×3。假设第一层卷积层使用尺寸为 5×5，深度为 16 的过滤器，那么这个卷积层的参数个数为 5×5×3×16+16=1216 个。上文提到过，使用 500 个隐藏节点的全连接层将有 150 万个参数。相比之下，卷积层的参数个数要远远小于全连接层。而且卷积层的参数个数和图片的大小无关，它只和过滤器的尺寸、深

度以及当前层节点矩阵的深度有关。这使得卷积神经网络可以很好地扩展到更大的图像数据上。

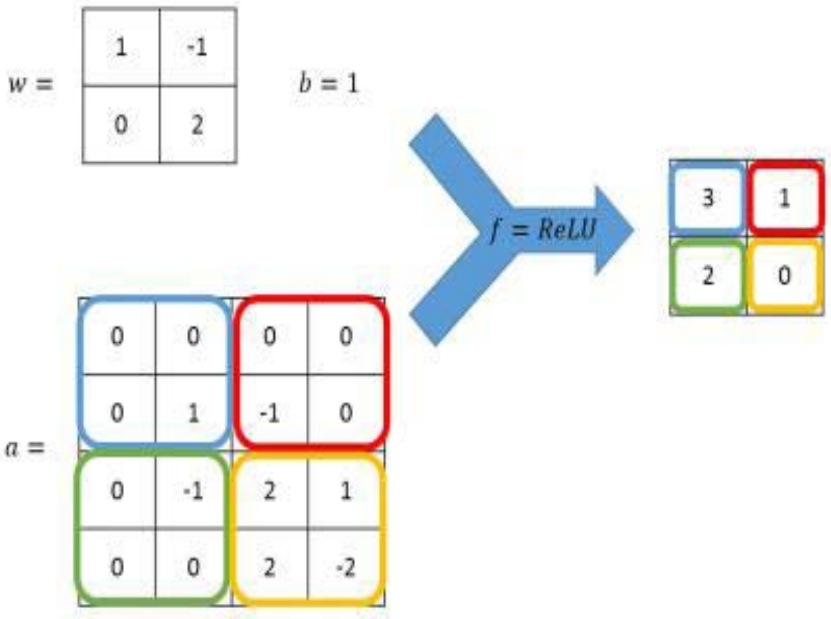


图6 卷积层前向传播过程样例图

结合过滤器的使用方法和参数共享的机制，图 6 给出了使用了全 0 填充、移动步长为 2 的卷积层前向传播的计算流程。下面的公式给出了左上角格子取值的计算方法，其他格子可以依次类推。

```
ReLu（0×1+0×（-1）+0×0+1×2+1）=ReLu（3）=3
```

TensorFlow 对卷积神经网络提供了非常好的支持，下面的程序实现了一个卷积层的前向传播过程。从以下代码可以看出，通过 TensorFlow 实现卷积层是非常方便的。

```
# 通过tf.get_variable的方式创建过滤器的权重变量和偏置项变量。上面介绍了卷积层
# 的参数个数只和过滤器的尺寸、深度以及当前层节点矩阵的深度有关，所以这里声明的参数变
# 量是一个四维矩阵，前面两个维度代表了过滤器的尺寸，第三个维度表示当前层的深度，第四
# 个维度表示过滤器的深度。
filter_weight = tf.get_variable(
    'weights', [5, 5, 3, 16],
    initializer=tf.truncated_normal_initializer(stddev=0.1))
# 和卷积层的权重类似，当前层矩阵上不同位置的偏置项也是共享的，所以总共有下一层深度个不
# 同的偏置项。本样例代码中16为过滤器的深度，也是神经网络中下一层节点矩阵的深度。
```

```
    biases = tf.get_variable(
        'biases', [16], initializer=tf.constant_initializer(0.1))
    # tf.nn.conv2d提供了一个非常方便的函数来实现卷积层前向传播的算法。这个函数的第一个输
    # 入为当前层的节点矩阵。注意这个矩阵是一个四维矩阵，后面三个维度对应一个节点矩阵，第一
    # 维对应一个输入batch。比如在输入层，input[0,:,:,:]表示第一张图片，input[1,:,:,:]
    # 表示第二张图片，以此类推。tf.nn.conv2d第二个参数提供了卷积层的权重，第三个参数为不
    # 同维度上的步长。虽然第三个参数提供的是一个长度为4的数组，但是第一维和最后一维的数字
    # 要求一定是1。这是因为卷积层的步长只对矩阵的长和宽有效。最后一个参数是填充（padding）
    # 的方法，TensorFlow中提供SAME或是VALID两种选择。其中SAME表示添加全0填充，
    # “VALID”表示不添加。
    conv = tf.nn.conv2d(
        input, filter_weight, strides=[1, 1, 1, 1], padding='SAME')
    # tf.nn.bias_add提供了一个方便的函数给每一个节点加上偏置项。注意这里不能直接使用加
    # 法，因为矩阵上不同位置上的节点都需要加上同样的偏置项。虽然下一层神经网络的大小为
    # 2×2，但是偏置项只有一个数（因为深度为1），而2×2矩阵中的每一个值都需要加上这个
    # 偏置项。
    bias = tf.nn.bias_add(conv, biases)
    # 将计算结果通过ReLU激活函数完成去线性化。
    actived_conv = tf.nn.relu(bias)
```

池化层网络结构

在卷积神经网络中，卷积层之间往往会加上一个池化层（pooling layer）。池化层可以非常有效地缩小矩阵的尺寸，从而减少最后全连接层中的参数。使用池化层既可以加快计算速度也有防止过拟合问题的作用。和卷积层类似，池化层前向传播的过程也是通过移动一个类似过滤器的结构完成的。不过池化层过滤器中的计算不是节点的加权和，而是采用更加简单的最大值或者平均值运算。使用最大值操作的池化层被称之为最大池化层（max pooling），这是被使用得最多的池化层结构。使用平均值操作的池化层被称之为平均池化层（average pooling）。

与卷积层的过滤器类似，池化层的过滤器也需要人工设定过滤器的尺寸、是否使用全 0 填充以及过滤器移动的步长等设置，而且这些设置的意

义也是一样的。卷积层和池化层中过滤器移动的方式是相似的，唯一的区别在于卷积层使用的过滤器是横跨整个深度的，而池化层使用的过滤器只影响一个深度上的节点。所以池化层的过滤器除了在长和宽两个维度移动之外，它还需要在深度这个维度移动。下面的 TensorFlow 程序实现了最大池化层的前向传播算法。

```
# tf.nn. max_pool实现了最大池化层的前向传播过程，它的参数和tf.nn.conv2d函数类似。
# ksize提供了过滤器的尺寸，strides提供了步长信息，padding提供了是否使用全0填充。
pool = tf.nn.max_pool(actived_conv, ksize=[1, 3, 3, 1],
                      strides=[1, 2, 2, 1], padding='SAME')
```

对比池化层和卷积层前向传播在 TensorFlow 中的实现，可以发现函数的参数形式是相似的。在 tf.nn.max_pool 函数中，首先需要传入当前层的节点矩阵，这个矩阵是一个四维矩阵，格式和 tf.nn.conv2d 函数中的第一个参数一致。第二个参数为过滤器的尺寸。虽然给出的是一个长度为 4 的一维数组，但是这个数组的第一个和最后一个数必须为 1。这意味着池化层的过滤器是不可以跨不同输入样例或者节点矩阵深度的。在实际应用中使用得最多的池化层过滤器尺寸为 [1,2,2,1] 或者 [1,3,3,1]。

tf.nn.max_pool 函数的第三个参数为步长，它和 tf.nn.conv2d 函数中步长的意义是一样的，而且第一维和最后一维也只能为 1。这意味着在 TensorFlow 中，池化层不能减少节点矩阵的深度或者输入样例的个数。tf.nn.max_pool 函数的最后一个参数指定了是否使用全 0 填充。这个参数也只有两种取值——VALID 或者 SAME，其中 VALID 表示不使用全 0 填充，SAME 表示使用全 0 填充。TensorFlow 还提供了 tf.nn.avg_pool 来实现平均池化层。tf.nn.avg_pool 函数的调用格式和 tf.nn.max_pool 函数是一致的。

LeNet-5 模型

LeNet-5 模型是 Yann LeCun 教授于 1998 年在论文 Gradient-based learning applied to document recognition 中提出的，它是第一个成功应用于数字识别问题的卷积神经网络。在 MNIST 数据集上，LeNet-5 模型可以达到大约 99.2% 的正确率。LeNet-5 模型总共有 7 层，图 7 展示了 LeNet-5

模型的架构。

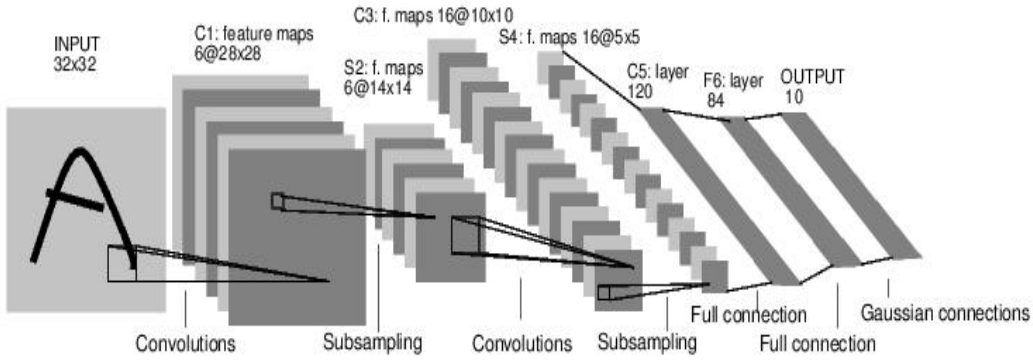


图7 LeNet-5模型结构图

在下面的篇幅中将详细介绍LeNet-5模型每一层的结构。论文GradientBased Learning Applied to Document Recognition提出的LeNet-5模型中，卷积层和池化层的实现与上文中介绍的TensorFlow的实现有细微的区别，这里不过多的讨论具体细节，而是着重介绍模型的整体框架。

第一层，卷积层

这一层的输入就是原始的图像像素，LeNet-5模型接受的输入层大小为32×32×1。第一个卷积层过滤器的尺寸为5×5，深度为6，不使用全0填充，步长为1。因为没有使用全0填充，所以这一层的输出的尺寸为32-5+1=28，深度为6。这一个卷积层总共有5×5×1×6+6=156个参数，其中6个为偏置项参数。因为下一层的节点矩阵有28×28×6=4704个节点，每个节点和5×5=25个当前层节点相连，所以本层卷积层总共有4704×（25+1）=122304个连接。

第二层，池化层

这一层的输入为第一层的输出，是一个28×28×6的节点矩阵。本层采用的过滤器大小为2×2，长和宽的步长均为2，所以本层的输出矩阵大小为14×14×6。原始的LeNet-5模型中使用的过滤器和本文中介绍的有些细微差别，这里不做具体介绍。

第三层，卷积层

本层的输入矩阵大小为 14×14×6，使用的过滤器大小为 5×5，深度为 16。本层不使用全 0 填充，步长为 1。本层的输出矩阵大小为 10×10×16。按照标准的卷积层，本层应该有 5×5×6×16+16=2416 个参数，10×10×16×（25+1）=41600 个连接。

第四层，池化层

本层的输入矩阵大小为 10×10×16，采用的过滤器大小为 2×2，步长为 2。本层的输出矩阵大小为 5×5×16。

第五层，全连接层

本层的输入矩阵大小为 5×5×16，在 LeNet-5 模型的论文中将这一层称为卷积层，但是因为过滤器的大小就是 5×5，所以和全连接层没有区别，在之后的 TensorFlow 程序实现中也会将这一层看成全连接层。如果将 5×5×16 矩阵中的节点拉成一个向量，那么这一层和在第四章中介绍的全连接层输入就一样了。本层的输出节点个数为 120，总共有 5×5×16×120+120=48120 个参数。

第六层，全连接层

本层的输入节点个数为 120 个，输出节点个数为 84 个，总共参数为 120×84+84=10164 个。

第七层，全连接层

本层的输入节点个数为 84 个，输出节点个数为 10 个，总共参数为 84×10+10=850 个。

上面介绍了 LeNet-5 模型每一层结构和设置，下面给出一个 TensorFlow 的程序来实现一个类似 LeNet-5 模型的卷积神经网络来解决 MNIST 数字识别问题。通过 TensorFlow 训练卷积神经网络的过程和第五章中介绍的训练全连接神经网络是完全一样的。损失函数的计算、反向传播过程的实现都可以复用上一篇中给出的 mnist_train.py 程序。唯一的区别在于因为卷积神经网络的输入层为一个三维矩阵，所以需要调整一下输

入数据的格式：

```
# 调整输入数据placeholder的格式，输入为一个四维矩阵。
x = tf.placeholder(tf.float32, [
                BATCH_SIZE,                        # 第一维表示一个batch中样例的个数。
                mnist_inference.IMAGE_SIZE,     # 第二维和第三维表示图片的尺寸。
                mnist_inference.IMAGE_SIZE,
                mnist_inference.NUM_CHANNELS],  name='x-input')
                                                # 第四维表示图片的深度，对于RBG格
                                                #式的图片，深度为5。
# 类似地将输入的训练数据格式调整为一个四维矩阵，并将这个调整后的数据传入sess.run过程。
reshaped_xs = np.reshape(xs, (BATCH_SIZE,
                                    mnist_inference.IMAGE_SIZE,
                                    mnist_inference.IMAGE_SIZE,
                                    mnist_inference.NUM_CHANNELS))
```

在调整完输入格式之后，只需要在程序 mnist_inference.py 中实现类似 LeNet-5 模型结构的前向传播过程即可。下面给出了修改后的 mnist_infernece.py 程序。

```
# -*- coding: utf-8 -*-
import tensorflow as tf
# 配置神经网络的参数。
INPUT_NODE = 784
OUTPUT_NODE = 10
IMAGE_SIZE = 28
NUM_CHANNELS = 1
NUM_LABELS = 10
# 第一层卷积层的尺寸和深度。
CONV1_DEEP = 32
CONV1_SIZE = 5
# 第二层卷积层的尺寸和深度。
CONV2_DEEP = 64
CONV2_SIZE = 5
# 全连接层的节点个数。
FC_SIZE = 512
```

```
# 定义卷积神经网络的前向传播过程。这里添加了一个新的参数train，用于区分训练过程和测试
# 过程。在这个程序中将用到dropout方法，dropout可以进一步提升模型可靠性并防止过拟合，
# dropout过程只在训练时使用。
def inference(input_tensor, train, regularizer):
 # 声明第一层卷积层的变量并实现前向传播过程。这个过程和6.3.1小节中介绍的一致。
 # 通过使用不同的命名空间来隔离不同层的变量，这可以让每一层中的变量命名只需要
 # 考虑在当前层的作用，而不需要担心重名的问题。和标准LeNet-5模型不大一样，这里
 # 定义的卷积层输入为28×28×1的原始MNIST图片像素。因为卷积层中使用了全0填充，
 # 所以输出为28×28×32的矩阵。
        with tf.variable_scope('layer1-conv1'):
       conv1_weights = tf.get_variable(
           "weight",  [CONV1_SIZE, CONV1_SIZE, NUM_CHANNELS, CONV1_DEEP],
           initializer=tf.truncated_normal_initializer(stddev=0.1))
       conv1_biases = tf.get_variable(
           "bias", [CONV1_DEEP], initializer=tf. constant_initializer(0.0))
 # 使用边长为5，深度为32的过滤器，过滤器移动的步长为1，且使用全0填充。
       conv1 = tf.nn.conv2d(
           input_tensor, conv1_weights,
           strides=[1, 1, 1, 1], padding='SAME')
       relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases))
   # 实现第二层池化层的前向传播过程。这里选用最大池化层，池化层过滤器的边长为2，
   # 使用全0填充且移动的步长为2。这一层的输入是上一层的输出，也就是28×28×32
   # 的矩阵。输出为14×14×32的矩阵。
   with tf.name_scope('layer2-pool1'):
       pool1 = tf.nn.max_pool(
           relu1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
   # 声明第三层卷积层的变量并实现前向传播过程。这一层的输入为14×14×32的矩阵。
   # 输出为14×14×64的矩阵。
   with tf.variable_scope('layer3-conv2'):
       conv2_weights = tf.get_variable(
           "weight", [CONV2_SIZE, CONV2_SIZE, CONV1_DEEP, CONV2_DEEP],
           initializer=tf.truncated_normal_initializer(stddev=0.1))
       conv2_biases = tf.get_variable(
           "bias", [CONV2_DEEP],
```

```
            initializer=tf. constant_initializer(0.0))
        # 使用边长为5，深度为64的过滤器，过滤器移动的步长为1，且使用全0填充。
        conv2 = tf.nn.conv2d(
            pool1, conv2_weights, strides=[1, 1, 1, 1], padding='SAME')
        relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_biases))
    # 实现第四层池化层的前向传播过程。这一层和第二层的结构是一样的。这一层的输入为
    # 14×14×64的矩阵，输出为7×7×64的矩阵。
    with tf.name_scope('layer4-pool2'):
        pool2 = tf.nn.max_pool(
            relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # 将第四层池化层的输出转化为第五层全连接层的输入格式。第四层的输出为7×7×64的矩阵，
    # 然而第五层全连接层需要的输入格式为向量，所以在这里需要将这个7×7×64的矩阵拉直成一
    # 个向量。pool2.get_shape函数可以得到第四层输出矩阵的维度而不需要手工计算。注意
    # 因为每一层神经网络的输入输出都为一个batch的矩阵，所以这里得到的维度也包含了一个
    # batch中数据的个数。
    pool_shape = pool2.get_shape().as_list()
    # 计算将矩阵拉直成向量之后的长度，这个长度就是矩阵长宽及深度的乘积。注意这里
    # pool_shape[0]为一个batch中数据的个数。
    nodes = pool_shape[1] * pool_shape[2] * pool_shape[3]
    # 通过tf.reshape函数将第四层的输出变成一个batch的向量。
    reshaped = tf.reshape(pool2, [pool_shape[0], nodes])
    # 声明第五层全连接层的变量并实现前向传播过程。这一层的输入是拉直之后的一组向量，
    # 向量长度为3136，输出是一组长度为512的向量。这一层和之前在第五章中介绍的基本
    # 一致，唯一的区别就是引入了dropout的概念。dropout在训练时会随机将部分节点的
    # 输出改为0。dropout可以避免过拟合问题，从而使得模型在测试数据上的效果更好。
    # dropout一般只在全连接层而不是卷积层或者池化层使用。
    with tf.variable_scope('layer5-fc1'):
        fc1_weights = tf.get_variable(
            "weight", [nodes, FC_SIZE],
             initializer=tf.truncated_normal_initializer(stddev=0.1))
        # 只有全连接层的权重需要加入正则化。
        if regularizer != None:
            tf.add_to_collection('losses', regularizer(fc1_weights))
        fc1_biases = tf.get_variable(
```

```
            "bias", [FC_SIZE], initializer=tf.constant_initializer(0.1))
        fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_weights) + fc1_biases)
        if train: fc1 = tf.nn.dropout(fc1, 0.5)
    # 声明第六层全连接层的变量并实现前向传播过程。这一层的输入为一组长度为512的向量，
    # 输出为一组长度为10的向量。这一层的输出通过Softmax之后就得到了最后的分类结果。
    with tf.variable_scope('layer6-fc2'):
        fc2_weights = tf.get_variable(
            "weight", [FC_SIZE, NUM_LABELS],
        initializer=tf.truncated_normal_initializer(stddev=0.1))
        if regularizer != None:
            tf.add_to_collection('losses', regularizer (fc2_weights))
        fc2_biases = tf.get_variable(
            "bias", [NUM_LABELS],
            initializer=tf. constant_initializer(0.1))
       logit = tf.matmul(fc1, fc2_weights) + fc2_biases

    # 返回第六层的输出。
    return logit
```

运行修改后的 mnist_train.py 和 mnist_eval.py，可以得到一下测试结果:

```
~/mnist$ python mnist_train.py
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
After 1 training step(s), loss on training batch is 6.45373.
After 1001 training step(s), loss on training batch is 0.824825.
After 2001 training step(s), loss on training batch is 0.646993.
After 3001 training step(s), loss on training batch is 0.759975.
After 4001 training step(s), loss on training batch is 0.68468.
After 5001 training step(s), loss on training batch is 0.630368.
```

上面的程序可以将 MNIST 正确率达到 ~99.4%。

循环神经网络简介

循环神经网络（recurrent neural network，RNN）源自于1982年由Saratha Sathasivam提出的霍普菲尔德网络。霍普菲尔德网络因为实现困难，在其提出的时候并且没有被合适地应用。该网络结构也于1986年后被全连接神经网络以及一些传统的机器学习算法所取代。然而，传统的机器学习算法非常依赖于人工提取的特征，使得基于传统机器学习的图像识别、语音识别以及自然语言处理等问题存在特征提取的瓶颈。而基于全连接神经网络的方法也存在参数太多、无法利用数据中时间序列信息等问题。随着更加有效的循环神经网络结构被不断提出，循环神经网络挖掘数据中的时序信息以及语义信息的深度表达能力被充分利用，并在语音识别、语言模型、机器翻译以及时序分析等方面实现了突破。

循环神经网络的主要用途是处理和预测序列数据。在之前介绍的全连接神经网络或卷积神经网络模型中，网络结构都是从输入层到隐含层再到输出层，层与层之间是全连接或部分连接的，但每层之间的节点是无连接的。考虑这样一个问题，如果要预测句子的下一个单词是什么，一般需要用到当前单词以及前面的单词，因为句子中前后单词并不是独立的。比如，当前单词是“很”，前一个单词是“天空”，那么下一个单词很大概率是“蓝”。循环神经网络的来源就是为了刻画一个序列当前的输出与之前信息的关系。从网络结构上，循环神经网络会记忆之前的信息，并利用之前的信息影响后面结点的输出。也就是说，循环神经网络的隐藏层之间的结点是有连接的，隐藏层的输入不仅包括输入层的输出，还包括上一时刻隐藏层的输出。

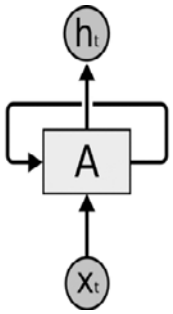


图1 循环神经网络经典结构示意图

图 1 展示了一个典型的循环神经网络。对于循环神经网络，一个非常重要的概念就是时刻。循环神经网络会对于每一个时刻的输入结合当前模型的状态给出一个输出。从图 1 中可以看到，循环神经网络的主体结构 A 的输入除了来自输入层 Xt，还有一个循环的边来提供当前时刻的状态。在每一个时刻，循环神经网络的模块 A 会读取 t 时刻的输入 Xt，并输出一个值 ht。同时 A 的状态会从当前步传递到下一步。因此，循环神经网络理论上可以被看作是同一神经网络结构被无限复制的结果。但出于优化的考虑，目前循环神经网络无法做到真正的无限循环，所以，现实中一般会将循环体展开，于是可以得到图 2 所展示的结构。

在图 2 中可以更加清楚的看到循环神经网络在每一个时刻会有一个输入 Xt，然后根据循环神经网络当前的状态 At 提供一个输出 Ht。从而神经网络当前状态 At 是根据上一时刻的状态 At-1 和当前输入 Xt 共同决定的。从循环神经网络的结构特征可以很容易地得出它最擅长解决的问题是与时间序列相关的。循环神经网络也是处理这类问题时最自然的神经网络

结构。对于一个序列数据，可以将这个序列上不同时刻的数据依次传入循环神经网络的输入层，而输出可以是对序列中下一个时刻的预测。循环神经网络要求每一个时刻都有一个输入，但是不一定每个时刻都需要有输出。在过去几年中，循环神经网络已经被广泛地应用在语音识别、语言模型、机器翻译以及时序分析等问题上，并取得了巨大的成功。

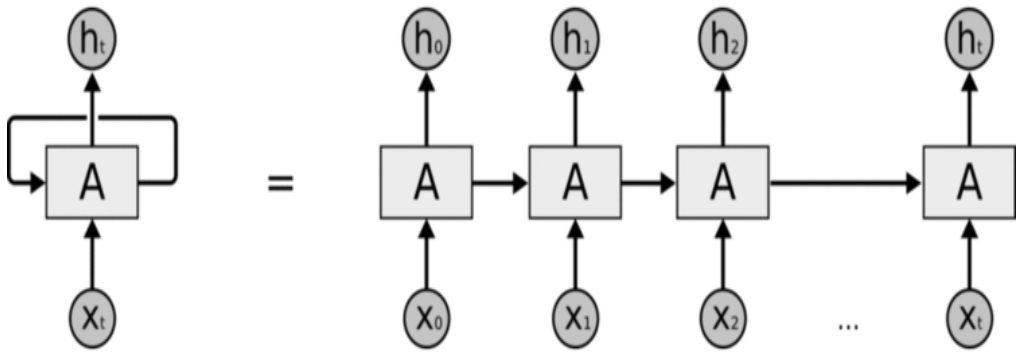


图 2 循环神经网络按时间展开后的结构

以机器翻译为例来介绍循环神经网络是如何解决实际问题的。循环神经网络中每一个时刻的输入为需要翻译的句子中的单词。如图 3 所示，需要翻译的句子为 ABCD，那么循环神经网络第一段每一个时刻的输入就分别是 A、B、C 和 D，然后用“”作为待翻译句子的结束符。在第一段中，循环神经网络没有输出。从结束符“”开始，循环神经网络进入翻译阶段。该阶段中每一个时刻的输入是上一个时刻的输出，而最终得到的输出就是句子 ABCD 翻译的结果。从图 3 中可以看到句子 ABCD 对应的翻译结果

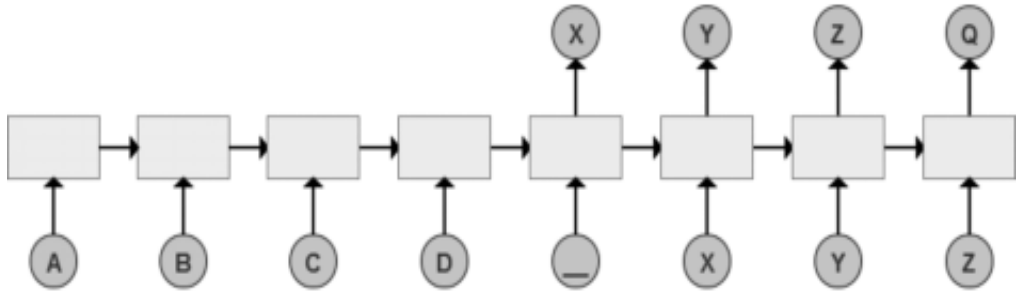


图 3

就是 XYZ，而 Q 是代表翻译结束的结束符。

如之前所介绍，循环神经网络可以被看做是同一神经网络结构在时间序列上被复制多次的结果，这个被复制多次的结构被称之为循环体。如何设计循环体的网络结构是循环神经网络解决实际问题的关键。和卷积神经网络过滤器中参数是共享的类似，在循环神经网络中，循环体网络结构中的参数在不同时刻也是共享的。

图 4 展示了一个使用最简单的循环体结构的循环神经网络，在这个循环体中只使用了一个类似全连接层的神经网络结构。下面将通过图 4 中所展示的神经网络来介绍循环神经网络前向传播的完整流程。循环神经网络中的状态是通过一个向量来表示的，这个向量的维度也称为循环神经网络隐藏层的大小，假设其为 h。从图 4 中可以看出，循环体中的神经网络的输入有两部分，一部分为上一时刻的状态，另一部分为当前时刻的输入样本。对于时间序列数据来说（比如不同时刻商品的销量），每一时刻的输入样例可以是当前时刻的数值（比如销量值）；对于语言模型来说，输入样例可以是当前单词对应的单词向量（word embedding）。

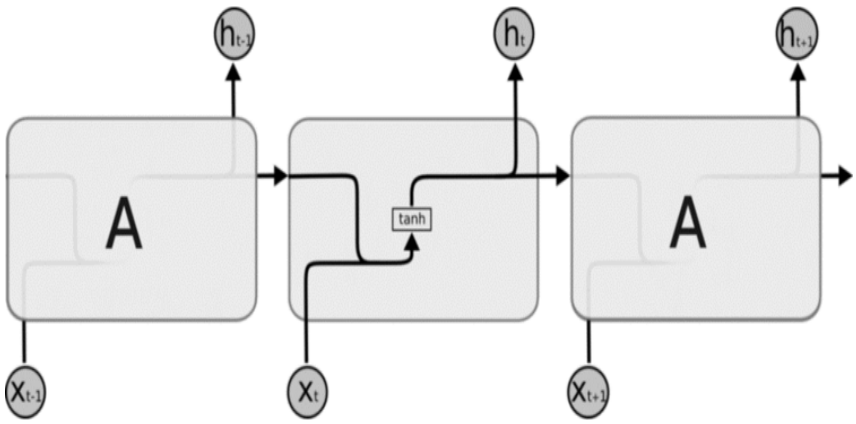


图 4 使用单层全连接神经网络作为循环体的循环神经网络结构图（图中中间标有 tanh 的小方框表示一个使用了 tanh 作为激活函数的全连接神经网络）

长短时记忆网络（LTSM）结构

循环神经网络工作的关键点就是使用历史的信息来帮组当前的决策。

例如使用之前出现的单词来加强对当前文字的理解。循环神经网络可以更好地利用传统神经网络结构所不能建模的信息，但同时，这也带来了更大的技术挑战——长期依赖（long-term dependencies）问题。

在有些问题中，模型仅仅需要短期内的信息来执行当前的任务。比如预测短语“大海的颜色是蓝色”中的最后一个单词“蓝色”时，模型并不需要记忆这个短语之前更长的上下文信息——因为这一句话已经包含了足够的信息来预测最后一个词。在这样的场景中，相关的信息和待预测的词的位置之间的间隔很小，循环神经网络可以比较容易地利用先前信息。

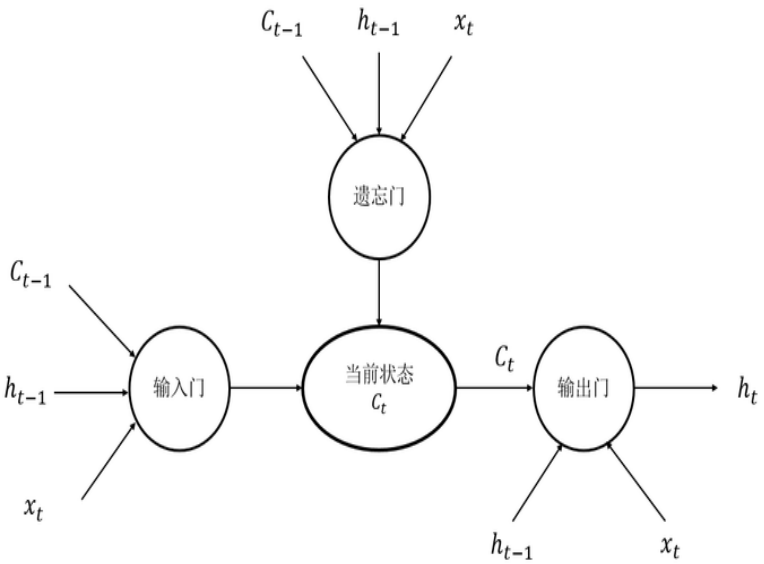


图 5 LSTM 单元结构示意图

但同样也会有一些上下文场景更加复杂的情况。比如当模型试着去预测段落“某地开设了大量工厂，空气污染十分严重 … 这里的天空都是灰色的”的最后一个单词时，仅仅根据短期依赖就无法很好的解决这种问题。因为只根据最后一小段，最后一个词可以是“蓝色的”或者“灰色的”。但如果模型需要预测清楚具体是什么颜色，就需要考虑先前提到但离当前位置较远的上下文信息。因此，当前预测位置和相关信息之间的文本间隔就有可能变得很大。当这个间隔不断增大时，类似图 4 中给出的简单循环神经网络有可能会丧失学习到距离如此远的信息的能力。或者在复杂语言场景中，

有用信息的间隔有大有小、长短不一，循环神经网络的性能也会受到限制。

长短时记忆网络（long short term memory, LSTM）的设计就是为了解决这个问题，而循环神经网络被成功应用的关键就是 LSTM。在很多的任务上，采用 LSTM 结构的循环神经网络比标准的循环神经网络表现更好。在下文中将重点介绍 LSTM 结构。LSTM 结构是由 Sepp Hochreiter 和 Jürgen Schmidhuber 于 1997 年提出的，它是一种特殊的循环体结构。如图 5 所示，与单一 tanh 循环体结构不同，LSTM 是一种拥有三个“门”结构的特殊网络结构。

LSTM 靠一些“门”的结构让信息有选择性地影响每个时刻循环神经网络中的状态。所谓“门”的结构就是一个使用 sigmoid 神经网络和一个按位做乘法的操作，这两个操作合在一起就是一个“门”的结构。之所以该结构叫做“门”是因为使用 sigmoid 作为激活函数的全连接神经网络层会输出一个 0 到 1 之间的数值，描述当前输入有多少信息量可以通过这个结构。于是这个结构的功能就类似于一扇门，当门打开时（sigmoid 神经网络层输出为 1 时），全部信息都可以通过；当门关上时（sigmoid 神经网络层输出为 0 时），任何信息都无法通过。本节下面的篇幅将介绍每一个“门”是如何工作的。

为了使循环神经网更有效的保存长期记忆，图 5 中“遗忘门”和“输入门”至关重要，它们是 LSTM 结构的核心。“遗忘门”的作用是让循环神经网络“忘记”之前没有用的信息。比如一段文章中先介绍了某地原来是绿水蓝天，但后来被污染了。于是在看到被污染了之后，循环神经网络应该“忘记”之前绿水蓝天的状态。这个工作是通过“遗忘门”来完成的。“遗忘门”会根据当前的输入 xt、上一时刻状态 ct-1 和上一时刻输出 ht-1 共同决定哪一部分记忆需要被遗忘。在循环神经网络“忘记”了部分之前的状态后，它还需要从当前的输入补充最新的记忆。这个过程就是“输入门”完成的。如图 5 所示，“输入门”会根据 xt、ct-1 和 ht-1 决定哪些部分将进入当前时刻的状态 ct。比如当看到文章中提到环境被污染之后，模型需要将这个信息写入新的状态。通过“遗忘门”和“输入门”，LSTM 结构可以更加有效的决定哪些信息

应该被遗忘，哪些信息应该得到保留。

LSTM 结构在计算得到新的状态 ct 后需要产生当前时刻的输出，这个过程是通过“输出门”完成的。“输出们”会根据最新的状态 ct、上一时刻的输出 ht-1 和当前的输入 xt 来决定该时刻的输出 ht。比如当前的状态为被污染，那么“天空的颜色”后面的单词很可能就是“灰色的”。

相比图 4 中展示的循环神经网络，使用 LSTM 结构的循环神经网络的前向传播是一个相对比较复杂的过程。具体 LSTM 每个“门”中的公式可以参考论文 Long short-term memory。在 TensorFlow 中，LSTM 结构可以被很简单地实现。以下代码展示了在 TensorFlow 中实现使用 LSTM 结构的循环神经网络的前向传播过程。

```
#定义一个LSTM结构。在TensorFlow中通过一句简单的命令就可以实现一个完整LSTM结构。
# LSTM中使用的变量也会在该函数中自动被声明。
lstm = rnn_cell.BasicLSTMCell(lstm_hidden_size)
# 将LSTM中的状态初始化为全0数组。和其他神经网络类似，在优化循环神经网络时，每次也
# 会使用一个batch的训练样本。以下代码中，batch_size给出了一个batch的大小。
# BasicLSTMCell类提供了zero_state函数来生成全领的初始状态。
state = lstm.zero_state(batch_size, tf.float32)
# 定义损失函数。
loss = 0.0
# 在8.1节中介绍过，虽然理论上循环神经网络可以处理任意长度的序列，但是在训练时为了
# 避免梯度消散的问题，会规定一个最大的序列长度。在以下代码中，用num_steps
# 来表示这个长度。
for i in range(num_steps):
# 在第一个时刻声明LSTM结构中使用的变量，在之后的时刻都需要复用之前定义好的变量。
if i > 0: tf.get_variable_scope().reuse_variables()
     # 每一步处理时间序列中的一个时刻。将当前输入（current_input）和前一时刻状态
     # （state）传入定义的LSTM结构可以得到当前LSTM结构的输出lstm_output和更新后
      # 的状态state。
    lstm_output, state = lstm(current_input, state)
   # 将当前时刻LSTM结构的输出传入一个全连接层得到最后的输出。
    final_output = fully_connected(lstm_output)
   # 计算当前时刻输出的损失。
    loss += calc_loss(final_output, expected_output)
```

通过上面这段代码看到，通过 TensorFlow 可以非常方便地实现使用

LSTM 结构的循环神经网络，而且并不需要用户对 LSTM 内部结构有深入的了解。

自然语言建模

简单地说，语言模型的目的是为了计算一个句子的出现概率。在这里把句子看成是单词的序列，于是语言模型需要计算的就是 p(w1,w2,w3,…,wn)。利用语言模型，可以确定哪个单词序列的可能性更大，或者给定若干个单词，可以预测下一个最可能出现的词语。举个音字转换的例子，假设输入的拼音串为“xianzaiquna”，它的输出可以是“西安在去哪”，也可以是“现在去哪”。根据语言常识，我们知道转换成第二个的概率更高。语言模型就可以告诉我们后者的概率大于前者，因此在大多数情况下转换成后者比较合理。

语言模型效果好坏的常用评价指标是复杂度（perplexity）。简单来说，perplexity 值刻画的就是通过某一个语言模型估计的一句话出现的概率。比如当已经知道（w1，w2，w3 · · · wm）这句话出现在语料库之中，那么通过语言模型计算得到的这句话的概率越高越好，也就是 perplexity 值越小越好。计算 perplexity 值的公式如下：

$$
\begin{aligned}
Perplexity(S) &= p(w_1,w_2,w_3,\ldots,w_m)^{-\frac{1}{m}} \\
&= \sqrt[m]{\frac{1}{p(w_1,w_2,w_3,\ldots,w_m)}} \\
&= \sqrt[m]{\prod_{i=1}^{m}\frac{1}{p(w_i|w_1,w_2,\ldots,w_{i-1})}}
\end{aligned}
$$

复杂度 perplexity 表示的概念其实是平均分支系数（average branch factor），即模型预测下一个词时的平均可选择数量。例如，考虑一个由

0~9 这 10 个数字随机组成的长度为 m 的序列。由于这 10 个数字出现的概率是随机的，所以每个数字出现的概率是 1/10。因此，在任意时刻，模型都有 10 个等概率的候选答案可以选择，于是 perplexity 就是 10（有 10 个合理的答案）。perplexity 的计算过程如下：

$$Perplexity(S) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{\frac{1}{10}}} = 10$$

因此，如果一个语言模型的 perplexity 是 89，就表示，平均情况下，模型预测下一个词时，有 89 个词等可能地可以作为下一个词的合理选择。

PTB （Penn Treebank Dataset）文本数据集是语言模型学习中目前最被广泛使用数据集。本小节将在 PTB 数据集上使用循环神经网络实现语言模型。在给出语言模型代码之前将先简单介绍 PTB 数据集的格式以及 TensorFlow 对于 PTB 数据集的支持。首先，需要下载来源于 Tomas Mikolov 网站上的 PTB 数据。数据的下载地址。

将下载下来的文件解压之后可以得到如下文件夹列表

```
1-train/
2-nbest-rescore/
3-combination/
4-data-generation/
5-one-iter/
6-recovery-during-training/
7-dynamic-evaluation/
8-direct/
9-char-based-lm/
data/
models/
rnnlm-0.2b/
```

在本文中只需要关心 data 文件夹下的数据，对于其他文件不再一一介绍，感兴趣的读者可以自行参考 README 文件。在 data 文件夹下总共有 7 个文件，但本文中将只会用到以下三个文件：

```
ptb.test.txt    #测试集数据文件
ptb.train.txt   #训练集数据文件
ptb.valid.txt   #验证集数据文件
```

这三个数据文件中的数据已经经过了预处理，包含了 10000 个不同的词语和语句结束标记符（在文本中就是换行符）以及标记稀有词语的特殊符号。下面展示了训练数据中的一行：

```
mr. <unk> is chairman of <unk> n.v. the dutch publishing group
```

为了让使用 PTB 数据集更加方便，TensorFlow 提供了两个函数来帮助实现数据的预处理。首先，TensorFlow 提供了 ptb_raw_data 函数来读取 PTB 的原始数据，并将原始数据中的单词转化为单词 ID。以下代码展示了如何使用这个函数。

```
from tensorflow.models.rnn.ptb import reader
# 存放原始数据的路径。
DATA_PATH = "/path/to/ptb/data"
train_data, valid_data, test_data, _ = reader.ptb_raw_data(DATA_PATH)
# 读取数据原始数据。
print len(train_data)
print train_data[:100]
'''
```

运行以上程序可以得到输出：

```
929589
[9970, 9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984, 9986, 9987,
9988, 9989, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 2, 9256, 1,
3, 72, 393, 33, 2133, 0, 146, 19, 6, 9207, 276, 407, 3, 2, 23, 1, 13, 141, 4,
1, 5465, 0, 3081, 1596, 96, 2, 7682, 1, 3, 72, 393, 8, 337, 141, 4, 2477, 657,
2170, 955, 24, 521, 6, 9207, 276, 4, 39, 303, 438, 3684, 2, 6, 942, 4, 3150,
496, 263, 5, 138, 6092, 4241, 6036, 30, 988, 6, 241, 760, 4, 1015, 2786, 211, 6,
96, 4]
'''
```

从输出中可以看出训练数据中总共包含了 929589 个单词，而这些单词被组成了一个非常长的序列。这个序列通过特殊的标识符给出了每句话结束的位置。在这个数据集中，句子结束的标识符 ID 为 2。

虽然循环神经网络可以接受任意长度的序列，但是在训练时需要将序列按照某个固定的长度来截断。为了实现截断并将数据组织成 batch，TensorFlow 提供了 ptb_iterator 函数。以下代码展示了如何使用 ptb_iterator 函数。

```
from tensorflow.models.rnn.ptb import reader
# 类似地读取数据原始数据。
DATA_PATH = "/path/to/ptb/data"
train_data, valid_data, test_data, _ = reader.ptb_raw_data(DATA_PATH)
# 将训练数据组织成batch大小为4、截断长度为5的数据组。
result = reader.ptb_iterator(train_data, 4, 5)
# 读取第一个batch中的数据，其中包括每个时刻的输入和对应的正确输出。
x, y = result.next()
print "X:", x
print "y:", y
'''
```

运行以上程序可以得到输出：

```
X: [[9970 9971 9972 9974 9975]
[ 332 7147  328 1452 8595]
[1969    0   98   89 2254]
[   3    3    2   14   24]]
y: [[9971 9972 9974 9975 9976]
[7147  328 1452 8595   59]
[   0   98   89 2254    0]
[   3    2   14   24  198]]
'''
```

图 6 展示了 ptb_iterator 函数实现的功能。ptb_iterator 函数会将一个长序列划分为 batch_size 段，其中 batch_size 为一个 batch 的大小。每次调用 ptb_iterator 时，该函数会从每一段中读取长度为 num_step 的子序列，其中 num_step 为截断的长度。从上面代码的输出可以看到，在第一个 batch 的第一行中，前面 5 个单词的 ID 和整个训练数据中前 5 个单词的 ID 是对应的。ptb_iterator 在生成 batch 时可以会自动生成每个 batch

对应的正确答案，这个对于每一个单词，它对应的正确答案就是该单词的后面一个单词。

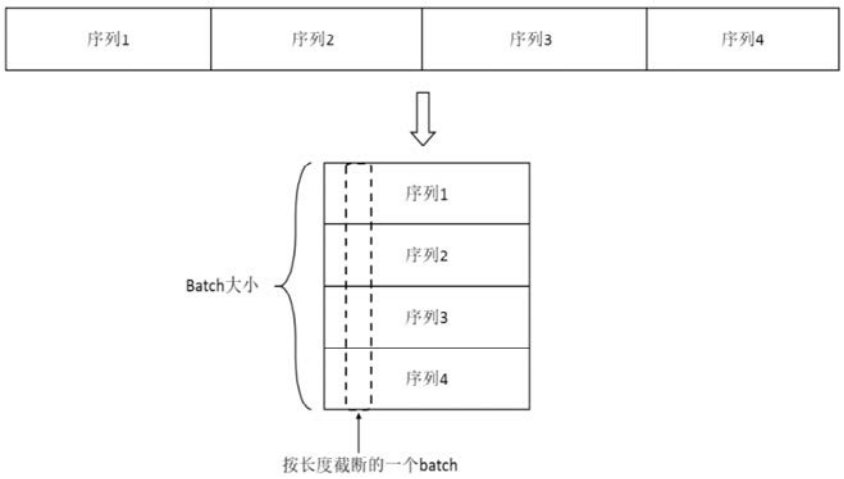


图 6 将一个长序列分成 batch 并截断的操作示意图

在介绍了语言模型的理论和使用到的数据集之后，下面给出了一个完成的 TensorFlow 样例程序来通过循环神经网络实现语言模型。

```
# -*- coding: utf-8 -*-
import numpy as np
import tensorflow as tf
from tensorflow.models.rnn.ptb import reader
DATA_PATH = "/path/to/ptb/data"  # 数据存放的路径。
HIDDEN_SIZE = 200           # 隐藏层规模。
NUM_LAYERS = 2               # 深层循环神经网络中LSTM结构的层数。
VOCAB_SIZE = 10000          # 词典规模，加上语句结束标识符和稀有
                               # 单词标识符总共一万个单词。
LEARNING_RATE = 1.0          # 学习速率。
TRAIN_BATCH_SIZE = 20       # 训练数据batch的大小。
TRAIN_NUM_STEP = 35         # 训练数据截断长度。
# 在测试时不需要使用截断，所以可以将测试数据看成一个超长的序列。
EVAL_BATCH_SIZE = 1         # 测试数据batch的大小。
EVAL_NUM_STEP = 1            # 测试数据截断长度。
NUM_EPOCH = 2                 # 使用训练数据的轮数。
KEEP_PROB = 0.5               # 节点不被dropout的概率。
```

```
MAX_GRAD_NORM = 5            # 用于控制梯度膨胀的参数。
# 通过一个PTBModel类来描述模型，这样方便维护循环神经网络中的状态。
class PTBModel(object):
    def __init__(self, is_training, batch_size, num_steps):
        # 记录使用的batch大小和截断长度。
        self.batch_size = batch_size
        self.num_steps = num_steps
        # 定义输入层。可以看到输入层的维度为batch_size × num_steps，这和
        # ptb_iterator函数输出的训练数据batch是一致的。
        self.input_data = tf.placeholder(tf.int32, [batch_size, num_steps])
        # 定义预期输出。它的维度和ptb_iterator函数输出的正确答案维度也是一样的。
        self.targets = tf.placeholder(tf.int32, [batch_size, num_steps])
        # 定义使用LSTM结构为循环体结构且使用dropout的深层循环神经网络。
        lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE)
        if is_training :
            lstm_cell = tf.nn.rnn_cell.DropoutWrapper(
                lstm_cell, output_keep_prob=KEEP_PROB)
        cell = tf.nn.rnn_cell.MultiRNNCell([lstm_cell] * NUM_LAYERS)
        # 初始化最初的状态，也就是全零的向量。
        self.initial_state = cell.zero_state(batch_size, tf.float32)
        # 将单词ID转换成为单词向量。因为总共有VOCAB_SIZE个单词，每个单词向量的维度
        # 为HIDDEN_SIZE，所以embedding参数的维度为VOCAB_SIZE × HIDDEN_SIZE。
        embedding = tf.get_variable("embedding", [VOCAB_SIZE, HIDDEN_SIZE])
        # 将原本batch_size × num_steps个单词ID转化为单词向量，转化后的输入层维度
        # 为batch_size × num_steps × HIDDEN_SIZE。
        inputs = tf.nn.embedding_lookup(embedding, self.input_data)
        # 只在训练时使用dropout。
        if is_training: inputs = tf.nn.dropout(inputs, KEEP_PROB)
        # 定义输出列表。在这里先将不同时刻LSTM结构的输出收集起来，再通过一个全连接
        # 层得到最终的输出。
        outputs = []
       # state 存储不同batch中LSTM的状态，将其初始化为0。
        state = self.initial_state
        with tf.variable_scope("RNN"):
```

```
        for time_step in range(num_steps):
            if time_step > 0: tf.get_variable_scope().reuse_variables()
            # 从输入数据中获取当前时刻获的输入并传入LSTM结构。
            cell_output, state = cell(inputs[:, time_step, :], state)
            # 将当前输出加入输出队列。
            outputs.append(cell_output)
    # 把输出队列展开成[batch, hidden_size*num_steps]的形状，然后再
    # reshape成[batch*numsteps, hidden_size]的形状。
    output = tf.reshape(tf.concat(1, outputs), [-1, HIDDEN_SIZE])
    # 将从LSTM中得到的输出再经过一个全链接层得到最后的预测结果，最终的预测结果在
    # 每一个时刻上都是一个长度为VOCAB_SIZE的数组，经过softmax层之后表示下一个
    # 位置是不同单词的概率。
    weight = tf.get_variable("weight", [HIDDEN_SIZE, VOCAB_SIZE])
    bias = tf.get_variable("bias", [VOCAB_SIZE])
    logits = tf.matmul(output, weight) + bias

    # 定义交叉熵损失函数。TensorFlow提供了sequence_loss_by_example函数来计
    # 算一个序列的交叉熵的和。
    loss = tf.nn.seq2seq.sequence_loss_by_example(
        [logits],                                   # 预测的结果。
        [tf.reshape(self.targets, [-1])],    # 期待的正确答案，这里将
                                             # [batch_size, num_steps]
                                             # 二维数组压缩成一维数组。
        # 损失的权重。在这里所有的权重都为1，也就是说不同batch和不同时刻
        # 的重要程度是一样的。
        [tf.ones([batch_size * num_steps], dtype=tf.float32)])

    # 计算得到每个batch的平均损失。
    self.cost = tf.reduce_sum(loss) / batch_size
    self.final_state = state

    # 只在训练模型时定义反向传播操作。
    if not is_training: return
```

```
        trainable_variables = tf.trainable_variables()
        # 通过clip_by_global_norm函数控制梯度的大小，避免梯度膨胀的问题。
        grads, _ = tf.clip_by_global_norm(
            tf.gradients(self.cost, trainable_variables), MAX_GRAD_NORM)

        # 定义优化方法。
        optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE)
        # 定义训练步骤。
        self.train_op = optimizer.apply_gradients(
             zip(grads, trainable_variables))

# 使用给定的模型model在数据data上运行train_op并返回在全部数据上的perplexity值。
def run_epoch(session, model, data, train_op, output_log):
    # 计算perplexity的辅助变量。
    total_costs = 0.0
    iters = 0
    state = session.run(model.initial_state)
    # 使用当前数据训练或者测试模型。
for step, (x, y) in enumerate(
        reader.ptb_iterator(data, model.batch_size, model.num_steps)):
        # 在当前batch上运行train_op并计算损失值。交叉熵损失函数计算的就是下一个单
        # 词为给定单词的概率。
        cost, state, _ = session.run(
            [model.cost, model.final_state, train_op],
            {model.input_data: x, model.targets: y,
             model.initial_state: state})
        # 将不同时刻、不同batch的概率加起来就可以得到第二个perplexity公式等号右
        # 边的部分，再将这个和做指数运算就可以得到perplexity值。
        total_costs += cost
        iters += model.num_steps
        # 只有在训练时输出日志。
        if output_log and step % 100 == 0:
            print("After %d steps, perplexity is %.3f" % (
                  step, np.exp(total_costs / iters)))
```

```
    # 返回给定模型在给定数据上的perplexity值。
    return np.exp(total_costs / iters)
    def main(_):
    # 获取原始数据。
    train_data, valid_data, test_data, _ = reader.ptb_raw_data(DATA_PATH)
    # 定义初始化函数。
    initializer = tf.random_uniform_initializer(-0.05, 0.05)
    # 定义训练用的循环神经网络模型。
with tf.variable_scope("language_model",
                           reuse=None, initializer=initializer):
        train_model = PTBModel(True, TRAIN_BATCH_SIZE, TRAIN_NUM_STEP)
    # 定义评测用的循环神经网络模型。
with tf.variable_scope("language_model",
                           reuse=True, initializer=initializer):
        eval_model = PTBModel(False, EVAL_BATCH_SIZE, EVAL_NUM_STEP)
    with tf.Session() as session:
        tf.initialize_all_variables().run()
        # 使用训练数据训练模型。
        for i in range(NUM_EPOCH):
            print("In iteration: %d" % (i + 1))
            # 在所有训练数据上训练循环神经网络模型。
            run_epoch(session, train_model,
                      train_data, train_model.train_op, True)
            # 使用验证数据评测模型效果。
            valid_perplexity = run_epoch(
                   session, eval_model, valid_data, tf.no_op(), False)
            print("Epoch: %d Validation Perplexity: %.3f" % (
                i + 1, valid_perplexity))
        # 最后使用测试数据测试模型效果。
        test_perplexity = run_epoch(
             session, eval_model, test_data, tf.no_op(), False)
        print("Test Perplexity: %.3f" % test_perplexity)
    if __name__ == "__main__":
        tf.app.run()
```

运行以上程序可以得到类似如下的输出：

```
In iteration: 1
After 0 steps, perplexity is 10003.783
After 100 steps, perplexity is 1404.742
After 200 steps, perplexity is 1061.458
After 300 steps, perplexity is 891.044
After 400 steps, perplexity is 782.037
…
After 1100 steps, perplexity is 228.711
After 1200 steps, perplexity is 226.093
After 1300 steps, perplexity is 223.214
Epoch: 2 Validation Perplexity: 183.443
Test Perplexity: 179.420
```

从输出可以看出，在迭代开始时 perplexity 值为 10003.783，这基本相当于从一万个单词中随机选择下一个单词。而在训练结束后，在训练数据上的 perplexity 值降低到了 179.420。这表明通过训练过程，将选择下一个单词的范围从一万个减小到了大约 180 个。通过调整 LSTM 隐藏层的节点个数和大小以及训练迭代的轮数还可以将 perplexity 值降到更低。

TensorFlow 高层封装

在前面的几篇文章已经介绍了如何使用原生态 TensorFlow API 来实现各种不同的神经网络结构。虽然原生态的 TensorFlow API 可以很灵活的支持不同的神经网络结构，但是其代码相对比较冗长，写起来比较麻烦。为了让 TensorFlow 用起来更加方便，可以使用一些 TensorFlow 的高层封装。

目前对 TensorFlow 的主要封装有 4 个：

- 第一个是TensorFlow-Slim；
- 第二个是tf.contrib.learn（之前也被称为skflow）；
- 第三个是TFLearn；
- 最后一个是Keras。

本文将大致介绍这几种不同的高层封装的使用方法，并通过其中常用的三种方式在 MNIST 数据集上实现卷积神经网络。

TensorFlow-Slim

TensorFlow-Slim 是一个相对轻量级的 TensorFlow 高层封装。通过 TensorFlow-Slim，定义网络结构的代码可以得到很大程度的简化，使得整个代码更加可读。下面的代码对比了使用原生态 TensorFlow 实现卷积层和使用 TensorFlow-Slim 实现卷积层的代码：

```
# 直接使用TensorFlow原生态API实现卷积层。
with tf.variable_scope(scope_name):
    weights = tf.get_variable("weight", …)
    biases = tf.get_variable("bias", …)
    conv = tf.nn.conv2d(…)
relu = tf.nn.relu(tf.nn.bias_add(conv, biases))
# 使用TensorFlow-Slim实现卷积层。通过TensorFlow-Slim可以在一行中实现一个卷积层的
# 前向传播算法。slim.conv2d函数的有3个参数是必填的。第一个参数为输入节点矩阵，第二参数
# 是当前卷积层过滤器的深度，第三个参数是过滤器的尺寸。可选的参数有过滤器移动的步长、
# 是否使用全0填充、激活函数的选择以及变量的命名空间等。
net = slim.conv2d(input, 32, [3, 3])
```

从上面的代码可以看出，使用 TensorFlow-Slim 可以大幅减少代码量。省去很多与网络结构无关的变量声明的代码。虽然 TensorFlow-Slim 可以起到简化代码的作用，但是在实际应用中，使用 TensorFlow-Slim 定义网络结构的情况相对较少，因为它既不如原生态 TensorFlow 的灵活，也不如下面将要介绍的其他高层封装简洁。但除了简化定义神经网络结构的代码量，使用 TensorFlow-Slim 的一个最大好处就是它直接实现了一些经典的卷积神经网络，并且 Google 提供了这些神经网络在 ImageNet 上训练好的模型。下表总结了通过 TensorFlow-Slim 可以直接实现的神经网络模型。

Google 提供的训练好的模型可以在 github 上 tensorflow/models/slim 目录下找到。在该目录下也提供了迁移学习的案例和代码。

模型	ImageNet上 Top 1 正确率	ImageNet上 Top5 正确率
Inception V1	69.8	89.6
Inception V2	73.9	91.8
Inception V3	78.0	93.9
Inception V4	80.2	95.2
Inception-ResNet-v2	80.4	95.3
ResNet V1 50	75.2	92.2
ResNet V1 101	76.4	92.9
ResNet V1 152	76.8	93.2
ResNet V2 50	75.6	92.8
ResNet V2 101	77.0	93.7
ResNet V2 152	77.8	94.1
VGG 16	71.5	89.8
VGG 19	71.1	89.8

tf.contrib.learn

tf.contrib.learn是TensorFlow官方提供的另外一个对TensorFlow的高层封装，通过这个封装，用户可以和使用sklearn类似的方法使用TensorFlow。通过tf.contrib.learn训练模型时，需要使用一个Estimator对象。Estimator对象是tf.contrib.learn 进行模型训练（train/fit）和模型评估（evaluation）的入口。

tf.contrib.learn模型提供了一些预定义的 Estimator，例如线性回归（tf.contrib.learn.LinearRegressor）、逻辑回归（tf.contrib.learn.LogisticRegressor）、线性分类（tf.contrib.learn.LinearClassifier）以及一些完全由全连接层构成的深度神经网络回归或者分类模型（tf.contrib.learn.DNNClassifier、tf.contrib.learn.DNNRegressor）。

除了可以使用预先定义好的模型，tf.contrib.learn也支持自定义模型，下面的代码给出了使用tf.contrib.learn在MNIST数据集上实现卷积神经网络的过程。更多关于tf.contrib.learn的介绍可以参考Google官方文档。

```
import tensorflow as tf
from sklearn import metrics
# 使用tf.contrib.layers中定义好的卷积神经网络结构可以更方便的实现卷积层。
layers = tf.contrib.layers
learn = tf.contrib.learn
# 自定义模型结构。这个函数有三个参数，第一个给出了输入的特征向量，第二个给出了
# 该输入对应的正确输出，最后一个给出了当前数据是训练还是测试。该函数的返回也有
# 三个指，第一个为定义的神经网络结构得到的最终输出节点取值，第二个为损失函数，第
# 三个为训练神经网络的操作。
def conv_model(input, target, mode):
    # 将正确答案转化成需要的格式。
    target = tf.one_hot(tf.cast(target, tf.int32), 10, 1, 0)
```

```
    # 定义神经网络结构，首先需要将输入转化为一个三维举证，其中第一维表示一个batch中的
    # 样例数量。
    network = tf.reshape(input, [-1, 28, 28, 1])
    # 通过tf.contrib.layers来定义过滤器大小为5*5的卷积层。
     network = layers.convolution2d(network, 32, kernel_size=[5, 5], activation_
fn=tf.nn.relu)
    # 实现过滤器大小为2*2，长和宽上的步长都为2的最大池化层。
     network = tf.nn.max_pool(network, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')
    # 类似的定义其他的网络层结构。
     network = layers.convolution2d(network, 64, kernel_size=[5, 5], activation_
fn=tf.nn.relu)
     network = tf.nn.max_pool(network, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')
    # 将卷积层得到的矩阵拉直成一个向量，方便后面全连接层的处理。
    network = tf.reshape(network, [-1, 7 * 7 * 64])
    # 加入dropout。注意dropout只在训练时使用。
    network = layers.dropout(
        layers.fully_connected(network, 500, activation_fn=tf.nn.relu),
        keep_prob=0.5,
        is_training=(mode == tf.contrib.learn.ModeKeys.TRAIN))
    # 定义最后的全连接层。
    logits = layers.fully_connected(network, 10, activation_fn=None)
    # 定义损失函数。
    loss = tf.losses.softmax_cross_entropy(target, logits)
    # 定义优化函数和训练步骤。
    train_op = layers.optimize_loss(
        loss,
              tf.contrib.framework.get_global_step(),
        optimizer='SGD',
        learning_rate=0.01)
    return tf.argmax(logits, 1), loss, train_op
# 加载数据。
mnist = learn.datasets.load_dataset('mnist')
```

```
# 定义神经网络结构，并在训练数据上训练神经网络。
classifier = learn.Estimator(model_fn=conv_model)
classifier.fit(mnist.train.images, mnist.train.labels, batch_size=100,
steps=20000)
# 在测试数据上计算模型准确率。
score = metrics.accuracy_score(mnist.test.labels, list(classifier.predict(mnist.
test.images)))
print('Accuracy: {0:f}'.format(score))
'''
```

运行上面的程序，可以得到类似如下的

```
Accuracy: 0.9901
'''
```

TFLearn

TensorFlow 的另外一个高层封装 TFLearn 进一步简化了 tf.contrib.learn 中对模型定义的方法，并提供了一些更加简洁的方法来定义神经网络的结构。和上面两个高层封装不一样，使用 TFLearn 需要单独安装，安装的方法为：

```
pip install tflearn
```

下面的代码介绍了如何通过 TFLearn 来实现卷积神经网络。更多关于 TFLearn 的用法介绍可以参考 TFLearn 的官方网站（http://tflearn.org/）

```
import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.estimator import regression
import tflearn.datasets.mnist as mnist
# 读取MNIST数据。
trainX, trainY, testX, testY = mnist.load_data(one_hot=True)
# 将图像数据resize成卷积卷积神经网络输入的格式。
trainX = trainX.reshape([-1, 28, 28, 1])
testX = testX.reshape([-1, 28, 28, 1])
# 构建神经网络。input_data定义了一个placeholder来接入输入数据。
```

```
network = input_data(shape=[None, 28, 28, 1], name='input')
# 定义一个深度为5，过滤器为5*5的卷积层。从这个函数可以看出，它比tf.contrib.learn
# 中对卷积层的抽象要更加简洁。
network = conv_2d(network, 32, 5, activation='relu')
# 定义一个过滤器为2*2的最大池化层。
network = max_pool_2d(network, 2)
# 类似的定义其他的网络结构。
network = conv_2d(network, 64, 5, activation='relu')
network = max_pool_2d(network, 2)
network = fully_connected(network, 500, activation='relu', regularizer="L2")
network = dropout(network, 0.5)
network = fully_connected(network, 10, activation='softmax', regularizer="L2")
# 定义学习任务。指定优化器为sgd，学习率为0.01，损失函数为交叉熵。
network = regression(network, optimizer='sgd', learning_rate=0.01,
                                                    loss='categorical_crossentropy',
name='target')
# 通过定义的网络结构训练模型，并在指定的验证数据上验证模型的效果。
model = tflearn.DNN(network, tensorboard_verbose=0)
model.fit(trainX, trainY, n_epoch=20, validation_set=([testX, testY]), show_
metric=True)
'''
```

运行上面的代码，可以得到类似如下的输出：

```
Run id: UY9GEP
Log directory: /tmp/tflearn_logs/
---------------------------------
Training samples: 55000
Validation samples: 10000
--
Training Step: 860  | total loss: 0.25554 | time: 493.917s
| SGD | epoch: 001 | loss: 0.25554 - acc: 0.9267 |
val_loss: 0.24617 - val_acc: 0.9267 -- iter: 55000/55000
--
Training Step: 1054  | total loss: 0.28228 | time: 110.039s
| SGD | epoch: 002 | loss: 0.28228 - acc: 0.9207 -- iter: 12416/55000
```

Keras

Keras 是一个基于 TensorFlow 或者 Theano 的高层 API，在安装好 TensorFlow 之后可以通过以下命令可以安装：

下面的代码介绍了如何通过 Keras 来实现卷积神经网络。更多关于 Keras 的用法介绍可以参考 Keras 的官方网站（http://tflearn.org/）

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
batch_size = 128
num_classes = 10
epochs = 20
img_rows, img_cols = 28, 28
# 加载MNIST数据。
(trainX, trainY), (testX, testY) = mnist.load_data()
# 根据系统要求设置输入层的格式。
if K.image_data_format() == 'channels_first':
    trainX = trainX.reshape(trainX.shape[0], 1, img_rows, img_cols)
    testX = testX.reshape(testX.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    trainX = trainX.reshape(trainX.shape[0], img_rows, img_cols, 1)
    testX = testX.reshape(testX.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
# 将图像像素转化为0到1之间的实数。
trainX = trainX.astype('float32')
testX = testX.astype('float32')
trainX /= 255.0
testX /= 255.0
# 将标准答案转化为需要的格式。
trainY = keras.utils.to_categorical(trainY, num_classes)
```

```
testY = keras.utils.to_categorical(testY, num_classes)
# 定义模型。
model = Sequential()
# 一层深度为32，过滤器大小为5*5的卷积层。
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', input_shape=input_
shape))
# 一层过滤器大小为2*2的最大池化层。
model.add(MaxPooling2D(pool_size=(2, 2)))
# 一层深度为64，过滤器大小为5*5的卷积层。
model.add(Conv2D(64, (5, 5), activation='relu'))
# 一层过滤器大小为2*2的最大池化层。
model.add(MaxPooling2D(pool_size=(2, 2)))
# 将上层最大池化层的输出在dropout之后提供给全连接层。
model.add(Dropout(0.5))
# 将卷积层的输出拉直后作为下面全连接层的输入。
model.add(Flatten())
# 全连接层，有500个节点。
model.add(Dense(500, activation='relu'))
# 全连接层，得到最后的输出。
model.add(Dense(num_classes, activation='softmax'))

# 定义损失函数、优化函数和评测方法。
model.compile(loss=keras.losses.categorical_crossentropy,
                        optimizer=keras.optimizers.SGD(),
                        metrics=['accuracy'])
# 类似TFLearn中的训练过程，给出训练数据、batch大小、训练轮数和验证数据，
# Keras可以自动完成模型训练过程。
model.fit(trainX, trainY,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(testX, testY))
# 在测试数据上计算准确率。
score = model.evaluate(testX, testY, verbose=0)
```

```
print('Test loss:', score[0])
print('Test accuracy:', score[1])

'''
```

运行上面的代码，可以得到类似如下的输出：

```
60000/60000 [==============================]
- 255s - loss: 1.3127 - acc: 0.5943 - val_loss: 0.3409 - val_acc: 0.9046
Epoch 2/20
60000/60000 [==============================]
- 219s - loss: 0.3827 - acc: 0.8840 - val_loss: 0.2006 - val_acc: 0.9433
Epoch 3/20
35072/60000 [================>.............]
- ETA: 82s - loss: 0.2752 - acc: 0.9163
'''
```

TensorFlow 计算加速

在前面的文章中介绍了使用 TensorFlow 实现各种深度学习的算法。然而要将深度学习应用到实际问题中，一个非常大的问题在于训练深度学习模型需要的计算量太大。比如要将 Inception-v3 模型在单机单卡上训练到 78% 的正确率需要将近半年的时间，这样的训练速度是完全无法应用到实际生产中的。为了加速训练过程，本文将介绍如何通过 TensorFlow 利用 GPU 或 / 和分布式计算进行模型训练。

TensorFlow 使用 GPU

TensorFlow 程序可以通过 tf.device 函数来指定运行每一个操作的设备，这个设备可以是本地的 CPU 或者 GPU，也可以是某一台远程的服务

器。TensorFlow 会给每一个可用的设备一个名称，tf.device 函数可以通过设备的名称来指定执行运算的设备。比如 CPU 在 TensorFlow 中的名称为 /cpu:0。

在默认情况下，即使机器有多个 CPU，TensorFlow 也不会区分它们，所有的 CPU 都使用 /cpu:0 作为名称。而一台机器上不同 GPU 的名称是不同的，第 n 个 GPU 在 TensorFlow 中的名称为 /gpu:n。比如第一个 GPU 的名称为 /gpu:0，第二个 GPU 名称为 /gpu:1，以此类推。

TensorFlow 提供了一个快捷的方式来查看运行每一个运算的设备。在生成会话时，可以通过设置 log_device_placement 参数来打印运行每一个运算的设备。下面的程序展示了如何使用 log_device_placement 这个参数。

```
import tensorflow as tf
a = tf.constant([1.0, 2.0, 3.0], shape=[3], name='a')
b = tf.constant([1.0, 2.0, 3.0], shape=[3], name='b')
c = a + b
# 通过log_device_placement参数来输出运行每一个运算的设备。
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print sess.run(c)
'''
```

在没有 GPU 的机器上运行以上代码可以得到以下输出：

```
Device mapping: no known devices.
add: /job:localhost/replica:0/task:0/cpu:0
b: /job:localhost/replica:0/task:0/cpu:0
a: /job:localhost/replica:0/task:0/cpu:0
[ 2.  4.  6.]
'''
```

在以上代码中，TensorFlow 程序生成会话时加入了参数 log_device_placement=True，所以程序会将运行每一个操作的设备输出到屏幕。于是除了可以看到最后的计算结果之外，还可以看到类似“add:/job:localhost/replica:0/task:0/cpu:0”这样的输出。这些输出显示了执行每一个运算的设备。比如加法操作 add 是通过 CPU 来运行的，因为它的设备名称中包含了 /cpu:0。

在配置好 GPU 环境的 TensorFlow 中，如果操作没有明确地指定运行设备，那么 TensorFlow 会优先选择 GPU。比如将以上代码在亚马逊（Amazon Web Services, AWS）的 g2.8xlarge 实例上运行时，会得到以下运行结果。

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: GRID K520, pci bus id:
0000:00:03.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: GRID K520, pci bus id:
0000:00:04.0
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: GRID K520, pci bus id:
0000:00:05.0
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: GRID K520, pci bus id:
0000:00:06.0
add: /job:localhost/replica:0/task:0/gpu:0
b: /job:localhost/replica:0/task:0/gpu:0
a: /job:localhost/replica:0/task:0/gpu:0
[ 2.  4.  6.]
```

从上面的输出可以看到在配置好 GPU 环境的 TensorFlow 中，TensorFlow 会自动优先将运算放置在 GPU 上。不过，尽管 g2.8xlarge 实例有 4 个 GPU，在默认情况下，TensorFlow 只会将运算优先放到 /gpu:0 上。于是可以看见在上面的程序中，所有的运算都被放在了 /gpu:0 上。如果需要将某些运算放到不同的 GPU 或者 CPU 上，就需要通过 tf.device 来手工指定。下面的程序给出了一个通过 tf.device 手工指定运行设备的样例。

```
import tensorflow as tf
# 通过tf.device将运算指定到特定的设备上。
with tf.device('/cpu:0'):
   a = tf.constant([1.0, 2.0, 3.0], shape=[3], name='a')
   b = tf.constant([1.0, 2.0, 3.0], shape=[3], name='b')
with tf.device('/gpu:1'):
    c = a + b
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print sess.run(c)
```

在 AWS g2.8xlarge 实例上运行上述代码可以得到一下结果：

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: GRID K520, pci bus id:
0000:00:03.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: GRID K520, pci bus id:
0000:00:04.0
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: GRID K520, pci bus id:
0000:00:05.0
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: GRID K520, pci bus id:
0000:00:06.0
add: /job:localhost/replica:0/task:0/gpu:1
b: /job:localhost/replica:0/task:0/cpu:0
a: /job:localhost/replica:0/task:0/cpu:0
[ 2.  4.  6.]
'''
```

在以上代码中可以看到生成常量 a 和 b 的操作被加载到了 CPU 上，而加法操作被放到了第二个 GPU“/gpu:1”上。在 TensorFlow 中，不是所有的操作都可以被放在 GPU 上，如果强行将无法放在 GPU 上的操作指定到 GPU 上，那么程序将会报错。

多 GPU 并行

下面将给出具体的 TensorFlow 代码在一台机器的多个 GPU 上并行训练深度学习模型。因为一般来说一台机器上的多个 GPU 性能相似，所以在这种设置下会更多地采用同步模式训练深度学习模型。下面将给出具体的代码，在多 GPU 上训练深度学习模型解决 MNIST 问题。（该代码可以在 TensorFlow 0.9.0 下运行，对于更新 TensorFlow 的版本，请参考 Github 代码库：https://github.com/caicloud/tensorflow-tutorial）

```
# -*- coding: utf-8 -*-
from datetime import datetime
import os
import time
```

```
import tensorflow as tf
# 定义训练神经网络时需要用到的配置。
BATCH_SIZE = 100
LEARNING_RATE_BASE = 0.001
LEARNING_RATE_DECAY = 0.99
REGULARAZTION_RATE = 0.0001
TRAINING_STEPS = 1000
MOVING_AVERAGE_DECAY = 0.99
N_GPU = 4
# 定义日志和模型输出的路径。
MODEL_SAVE_PATH = "/path/to/logs_and_models/"
MODEL_NAME = "model.ckpt"
# 定义数据存储的路径。因为需要为不同的GPU提供不同的训练数据，所以通过placerholder
# 的方式就需要手动准备多份数据。为了方便训练数据的获取过程，可以采用输入队列的方式从
# TFRecord中读取数据。于是在这里提供的数据文件路径为将MNIST训练数据转化为
# TFRecords格式之后的路径。
DATA_PATH = "/path/to/data.tfrecords"
# 定义输入队列得到训练数据，具体细节可以参考《TensorFlow:实战Google深度学习框架》
# 第七章。
def get_input():
    filename_queue = tf.train.string_input_producer([DATA_PATH])
    reader = tf.TFRecordReader()
_, serialized_example = reader.read(filename_queue)
```

```
# 定义数据解析格式。
features = tf.parse_single_example(
    serialized_example,
    features={
        'image_raw': tf.FixedLenFeature([], tf.string),
        'pixels': tf.FixedLenFeature([], tf.int64),
        'label': tf.FixedLenFeature([], tf.int64),
    })
# 解析图片和标签信息。
```

```
    decoded_image = tf.decode_raw(features['image_raw'], tf.uint8)
    reshaped_image = tf.reshape(decoded_image, [784])
    retyped_image = tf.cast(reshaped_image, tf.float32)
    label = tf.cast(features['label'], tf.int32)
    # 定义输入队列并返回。
    min_after_dequeue = 10000
    capacity = min_after_dequeue + 3 * BATCH_SIZE
return tf.train.shuffle_batch(
        [retyped_image, label],
        batch_size=BATCH_SIZE,
        capacity=capacity,
        min_after_dequeue=min_after_dequeue)
# 定义损失函数。对于给定的训练数据、正则化损失计算规则和命名空间，计算在这个命名空间
# 下的总损失。之所以需要给定命名空间是因为不同的GPU上计算得出的正则化损失都会加入名为
# loss的集合，如果不通过命名空间就会将不同GPU上的正则化损失都加进来。
def get_loss(x, y_, regularizer, scope):
    # 沿用第四篇文章中定义的卷积神经网络计算前向传播结果。
y = inference(x, regularizer)
# 计算交叉熵损失。
cross_entropy = tf.reduce_mean(
    tf.nn.sparse_softmax_cross_entropy_with_logits(y, y_))
# 计算当前GPU上计算得到的正则化损失。
regularization_loss = tf.add_n(tf.get_collection('losses', scope))
# 计算最终的总损失。
    loss = cross_entropy + regularization_loss
    return loss
    # 计算每一个变量梯度的平均值。
def average_gradients(tower_grads):
average_grads = []
# 枚举所有的变量和变量在不同GPU上计算得出的梯度。
for grad_and_vars in zip(*tower_grads):
          # 计算所有GPU上的梯度平均值。
          grads = []
          for g, _ in grad_and_vars:
```

```
                expanded_g = tf.expand_dims(g, 0)
                grads.append(expanded_g)
            grad = tf.concat(0, grads)
            grad = tf.reduce_mean(grad, 0)

            v = grad_and_vars[0][7]
            grad_and_var = (grad, v)
            # 将变量和它的平均梯度对应起来。
            average_grads.append(grad_and_var)
    # 返回所有变量的平均梯度，这将被用于变量更新。
    return average_grads
# 主训练过程。
def main(argv=None):
    # 将简单的运算放在CPU上，只有神经网络的训练过程放在GPU上。
with tf.Graph().as_default(), tf.device('/cpu:0'):
        # 获取训练batch。
        x, y_ = get_input()
        regularizer = tf.contrib.layers.l2_regularizer(REGULARAZTION_RATE)

        # 定义训练轮数和指数衰减的学习率。
    global_step = tf.get_variable(
        'global_step', [], initializer=tf.constant_initializer(0),
         trainable=False)
        learning_rate = tf.train.exponential_decay(
            LEARNING_RATE_BASE, global_step, 60000 / BATCH_SIZE,
            LEARNING_ RATE_DECAY)

        # 定义优化方法。
        opt = tf.train.GradientDescentOptimizer(learning_rate)
        tower_grads = []
        # 将神经网络的优化过程跑在不同的GPU上。
        for i in range(N_GPU):
            # 将优化过程指定在一个GPU上。
            with tf.device('/gpu:%d' % i):
```

```
                with tf.name_scope('GPU_%d' % i) as scope:
                    cur_loss = get_loss(x, y_, regularizer, scope)
                    # 在第一次声明变量之后，将控制变量重用的参数设置为True。这样可以
                    # 让不同的GPU更新同一组参数。注意tf.name_scope函数并不会影响
# tf.get_ variable的命名空间。
                    tf.get_variable_scope().reuse_variables()

                    # 使用当前GPU计算所有变量的梯度。
                    grads = opt.compute_gradients(cur_loss)
                    tower_grads.append(grads)
    # 计算变量的平均梯度，并输出到TensorBoard日志中。
    grads = average_gradients(tower_grads)
    for grad, var in grads:
        if grad is not None:
            tf.histogram_summary(
                'gradients_on_average/%s' % var.op.name, grad)
    # 使用平均梯度更新参数。
    apply_gradient_op = opt.apply_gradients(
        grads, global_step=global_ step)
    for var in tf.trainable_variables():
        tf.histogram_summary(var.op.name, var)
    # 计算变量的滑动平均值。
    variable_averages = tf.train.ExponentialMovingAverage(
        MOVING_AVERAGE_DECAY, global_step)
    variables_averages_op = variable_averages.apply(
        tf.trainable_variables())
    # 每一轮迭代需要更新变量的取值并更新变量的滑动平均值。
    train_op = tf.group(apply_gradient_op, variables_averages_op)
    saver = tf.train.Saver(tf.all_variables())
    summary_op = tf.merge_all_summaries()
    init = tf.initialize_all_variables()
    # 训练过程。
    with tf.Session(config=tf.ConfigProto(
        allow_soft_placement=True,
```

```
log_device_placement=True)) as sess:
# 初始化所有变量并启动队列。
init.run()
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
summary_writer = tf.train.SummaryWriter(
    MODEL_SAVE_PATH, sess.graph)

for step in range(TRAINING_STEPS):
    # 执行神经网络训练操作，并记录训练操作的运行时间。
    start_time = time.time()
    _, loss_value = sess.run([train_op, cur_loss])
    duration = time.time() - start_time
    # 每隔一段时间展示当前的训练进度，并统计训练速度。
    if step != 0 and step % 10 == 0:
        # 计算使用过的训练数据个数。因为在每一次运行训练操作时，每一个GPU
        # 都会使用一个batch的训练数据，所以总共用到的训练数据个数为
        # batch大小×GPU个数。
        num_examples_per_step = BATCH_SIZE * N_GPU

        # num_examples_per_step为本次迭代使用到的训练数据个数，
        # duration为运行当前训练过程使用的时间，于是平均每秒可以处理的训
        # 练数据个数为num_examples_per_step / duration。
        examples_per_sec = num_examples_per_step / duration

        # duration为运行当前训练过程使用的时间，因为在每一个训练过程中，
        # 每一个GPU都会使用一个batch的训练数据，所以在单个batch上的训
        # 练所需要时间为duration / GPU个数。
        sec_per_batch = duration / N_GPU

        # 输出训练信息。
        format_str = ('step %d, loss = %.2f (%.1f examples/ '
                      ' sec; %.3f sec/batch)')
        print(format_str % (step, loss_value,
```

```
                                                examples_per_sec, sec_per_batch))
                        # 通过TensorBoard可视化训练过程。
                        summary = sess.run(summary_op)
                        summary_writer.add_summary(summary, step)

                    # 每隔一段时间保存当前的模型。
                    if step % 1000 == 0 or (step + 1) == TRAINING_STEPS:
                        checkpoint_path = os.path.join(
                            MODEL_SAVE_PATH, MODEL_ NAME)
                        saver.save(sess, checkpoint_path, global_step=step)
                coord.request_stop()
                coord.join(threads)
    if __name__ == '__main__':
        tf.app.run()

    '''
```

在 AWS 的 g2.8xlarge 实例上运行上面这段程序可以得到类似下面的结果：

```
step 10, loss = 71.90 (15292.3 examples/sec; 0.007 sec/batch)
step 20, loss = 37.97 (18758.3 examples/sec; 0.005 sec/batch)
step 30, loss = 9.54 (16313.3 examples/sec; 0.006 sec/batch)
step 40, loss = 11.84 (14199.0 examples/sec; 0.007 sec/batch)
...
step 980, loss = 0.66 (15034.7 examples/sec; 0.007 sec/batch)
step 990, loss = 1.56 (16134.1 examples/sec; 0.006 sec/batch)
'''
```

在 AWS 的 g2.8xlarge 实例上运行以上代码可以同时使用 4 个 GPU 训练神经网络。图 1 显示了运行样例代码时不同 GPU 的使用情况。因为运行的神经网络规模比较小，所以在图 1 中显示的 GPU 使用率不高。如果训练大型的神经网络模型，TensorFlow 将会占满所有用到的 GPU。

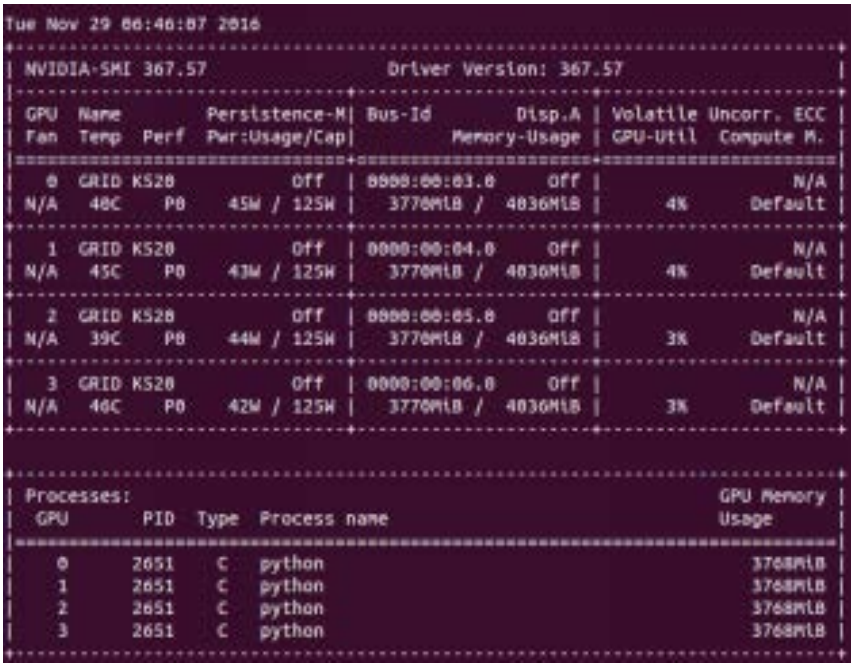


图1 在AWS的g2.8xlarge实例上运行MNIST样例程序时GPU的使用情况

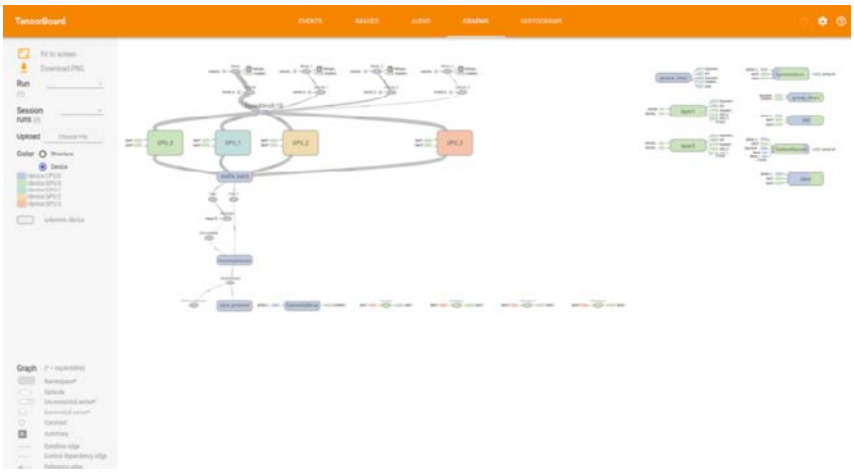


图2 使用了4个GPU的TensorFlow计算图可视化结果

图 2 展示了通过 TensorBoard 可视化得到的样例代码 TensorFlow 计算图，其中节点上的颜色代表了不同的设备，比如黑色代表 CPU、白色代表第一个 GPU，等等。从图 2 中可以看出，训练神经网络的主要过程

被放到了 GPU_0、GPU_1、GPU_2 和 GPU_3 这 4 个模块中，而且每一个模块运行在一个 GPU 上。

分布式 TensorFlow

通过多 GPU 并行的方式可以达到很好的加速效果。然而一台机器上能够安装的 GPU 有限，要进一步提升深度学习模型的训练速度，就需要将 TensorFlow 分布式运行在多台机器上。以下代码展示了如何创建一个最简单的 TensorFlow 集群：

```
import tensorflow as tf
c = tf.constant("Hello, distributed TensorFlow!")
# 创建一个本地TensorFlow集群
server = tf.train.Server.create_local_server()
# 在集群上创建一个会话。
sess = tf.Session(server.target)
# 输出Hello, distributed TensorFlow!
print sess.run(c)
```

在以上代码中，首先通过 tf.train.Server.create_local_server 函数在本地建立了一个只有一台机器的 TensorFlow 集群。然后在该集群上生成了一个会话，并通过生成的会话将运算运行在创建的 TensorFlow 集群上。虽然这只是一个单机集群，但它大致反应了 TensorFlow 集群的工作流程。TensorFlow 集群通过一系列的任务（tasks）来执行 TensorFlow 计算图中的运算。一般来说，不同任务跑在不同机器上。最主要的例外是使用 GPU 时，不同任务可以使用同一台机器上的不同 GPU。TensorFlow 集群中的任务也会被聚合成工作（jobs），每个工作可以包含一个或者多个任务。比如在训练深度学习模型时，一台运行反向传播的机器是一个任务，而所有运行反向传播机器的集合是一种工作。

上面的样例代码是只有一个任务的集群。当一个 TensorFlow 集群有多个任务时，需要使用 tf.train.ClusterSpec 来指定运行每一个任务的机器。比如以下代码展示了在本地运行有两个任务的 TensorFlow 集群。第一个

任务的代码如下：

```
import tensorflow as tf
c = tf.constant("Hello from server1!")
# 生成一个有两个任务的集群，一个任务跑在本地2222端口，另外一个跑在本地2223端口。
cluster = tf.train.ClusterSpec(
    {"local": ["localhost:2222", "localhost: 2223"]})
# 通过上面生成的集群配置生成Server，并通过job_name和task_index指定当前所启动
# 的任务。因为该任务是第一个任务，所以task_index为0。
server = tf.train.Server(cluster, job_name="local", task_index=0)

# 通过server.target生成会话来使用TensorFlow集群中的资源。通过设置
# log_device_placement可以看到执行每一个操作的任务。
sess = tf.Session(
    server.target, config=tf.ConfigProto(log_device_placement=True))
print sess.run(c)
server.join()
```

下面给出了第二个任务的代码：

```
import tensorflow as tf
c = tf.constant("Hello from server2!")
# 和第一个程序一样的集群配置。集群中的每一个任务需要采用相同的配置。
cluster = tf.train.ClusterSpec(
    {"local": ["localhost:2222", "localhost: 2223"]})
# 指定task_index为1，所以这个程序将在localhost:2223启动服务。
server = tf.train.Server(cluster, job_name="local", task_index=1)
# 剩下的代码都和第一个任务的代码一致。
...
```

启动第一个任务后，可以得到类似下面的输出：

```
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:206] Initialize
HostPortsGrpcChannelCache for job local -> {localhost:2222, localhost:2223}
I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:202] Started server
with target: grpc://localhost:2222
E1123 08:26:06.824503525    12232 tcp_client_posix.c:173]      failed to connect
to 'ipv4:127.0.0.1:2223': socket error: connection refused
```

```
E1123 08:26:08.825022513   12232 tcp_client_posix.c:173]     failed to connect
to 'ipv4:127.0.0.1:2223': socket error: connection refused
I tensorflow/core/common_runtime/simple_placer.cc:818] Const: /job:local/
replica:0/task:0/cpu:0
Const: /job:local/replica:0/task:0/cpu:0
Hello from server1!
```

从第一个任务的输出中可以看到，当只启动第一个任务时，程序会停下来等待第二个任务启动，而且持续输出 failed to connect to 'ipv4:127.0.0.1:2223': socket error: connection refused。当第二个任务启动后，可以看到从第一个任务中会输出 Hello from server1! 的结果。第二个任务的输出如下：

```
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:206] Initialize
HostPortsGrpcChannelCache for job local -> {localhost:2222, localhost:2223}
I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:202] Started server
with target: grpc://localhost:2223
Const: /job:local/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:818] Const: /job:local/
replica:0/task:0/cpu:0
Hello from server2!
```

值得注意的是第二个任务中定义的计算也被放在了设备 /job:local/replica:0/task:0/cpu:0 上。也就是说这个计算将由第一个任务来执行。从上面这个样例可以看到，通过 tf.train.Server.target 生成的会话可以统一管理整个 TensorFlow 集群中的资源。

和使用多 GPU 类似，TensorFlow 支持通过 tf.device 来指定操作运行在哪个任务上。比如将第二个任务中定义计算的语句改为以下代码，就可以看到这个计算将被调度到 /job:local/replica:0/task:1/cpu:0 上面。

```
with tf.device("/job:local/task:1"):
    c = tf.constant("Hello from server2!")
```

在上面的样例中只定义了一个工作“local”。但一般在训练深度学习模型时，会定义两个工作。一个工作专门负责存储、获取以及更新变量的取值，这个工作所包含的任务统称为参数服务器（parameter server，ps）。

另外一个工作负责运行反向传播算法来获取参数梯度，这个工作所包含的任务统称为计算服务器（worker）。下面给出了一个比较常见的用于训练深度学习模型的 TensorFlow 集群配置方法。

```
tf.train.ClusterSpec({
    "worker": [
        "tf-worker0:2222",
        "tf-worker1:2222",
        "tf-worker2:2222"
    ],
    "ps": [
        "tf-ps0:2222",
        "tf-ps1:2222"
    ]})
```

使用分布式 TensorFlow 训练深度学习模型一般有两种方式。一种方式叫做计算图内分布式（in-graph replication）。使用这种分布式训练方式时，所有的任务都会使用一个 TensorFlow 计算图中的变量（也就是深度学习模型中的参数），而只是将计算部分发布到不同的计算服务器上。

上面给出的使用多 GPU 样例程序就是这种方式。多 GPU 样例程序将计算复制了多份，每一份放到一个 GPU 上进行运算。但不同的 GPU 使用的参数都是在一个 TensorFlow 计算图中的。因为参数都是存在同一个计算图中，所以同步更新参数比较容易控制。在上面给出的代码也实现了参数的同步更新。然而因为计算图内分布式需要有一个中心节点来生成这个计算图并分配计算任务，所以当数据量太大时，这个中心节点容易造成性能瓶颈。

另外一种分布式 TensorFlow 训练深度学习模型的方式叫计算图之间分布式（between-graph replication）。使用这种分布式方式时，在每一个计算服务器上都会创建一个独立的 TensorFlow 计算图，但不同计算图中的相同参数需要以一种固定的方式放到同一个参数服务器上。TensorFlow 提供了 tf.train.replica_device_setter 函数来帮助完成这一个过程。

因为每个计算服务器的 TensorFlow 计算图是独立的，所以这种方式

的并行度要更高。但在计算图之间分布式下进行参数的同步更新比较困难。

为了解决这个问题，TensorFlow 提供了 tf.train.SyncReplicasOptimizer 函数来帮助实现参数的同步更新。这让计算图之间分布式方式被更加广泛地使用。因为篇幅所限，所以这里不在给出具体代码，具体代码可以在 Github 代码库中找到。

使用 Caicloud TaaS 平台运行分布式 TensorFlow

每次运行分布式 TensorFlow 都需要登录不同的机器来启动集群。这使得使用起来非常不方便。当需要使用 100 台机器运行分布式 TensorFlow 时，需要手动登录到每一台机器并启动 TensorFlow 服务，这个过程十分繁琐。而且，当某个服务器上的程序死掉之后，TensorFlow 并不能自动重启，这给监控工作带来了巨大的难度。

如果类比 TensorFlow 与 Hadoop，可以发现 TensorFlow 只实现了相当于 Hadoop 中 MapReduce 的计算框架，而没有提供类似 Yarn 的集群管理工具以及 HDFS 的存储系统。为了降低分布式 TensorFlow 的使用门槛，才云科技 (Caicloud.io) 基于 Kubernetes 容器云平台提供了一个分布式 TensorFlow 平台 TensorFlow as a Service（TaaS）。

从我们提供的开源代码库中可以看出，编写分布式 TensorFlow 程序需要指定很多与模型训练无关的代码来完成 TensorFlow 集群的设置工作。为了降低分布式 TensorFlow 的学习成本，Caicloud 的 TensorFlow as a Service（TaaS）平台首先对 TensorFlow 集群进行了更高层的封装，屏蔽了其中与模型训练无关的底层细节。

其次，TaaS 平台结合了谷歌开源的容器云平台管理工具 Kubernetes 来实现对分布式 TensorFlow 任务的管理和监控，并支持通过 UI 设置分布式 TensorFlow 任务的节点个数、是否使用 GPU 等信息。

Caicloud TaaS 平台的更多信息可以参考文档。

TaaS 平台的使用文档。

版权声明

InfoQ 中文站出品

出版：北京极客邦科技有限公司
北京市朝阳区洛娃大厦C座1607
欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系editors@cn.infoq.com。
网 址：www.infoq.com.cn

TensorFlow实战

郑泽宇

才云科技联合创始人/首席大数据科学家
前谷歌高级工程师

课程大纲

深度学习与TensorFlow简介
深层神经网络解决MNIST问题
神经网络优化方法
卷积神经网络
循环神经网络
TensorFlow加速

现在报名，立减 300 元
加小助手，咨询课程