**Tensorflow实验：cifar-10数据集分类**

20354240 帅灿宇

**前言**

本实验是使用tensorflow对cifar-10数据集进行分类的任务。在实验中利用tensorflow设计网络模型构建计算流图实现对数据集的分类，并对其进行优化操作提升网络性能。通过对cifar-10数据集的分类实验深刻理解tensorflow的原理，掌握其核心部分即计算流图的构建，并在此基础上探究提升网络性能的方式。

在实验中，利用tensorflow构建的初始神经网络模型框架如图1，网络由两层卷积层加池化层构成，再由三层全连接层处理得到一个1\*10的输出，最后再经过softmax函数对数据进行分类。

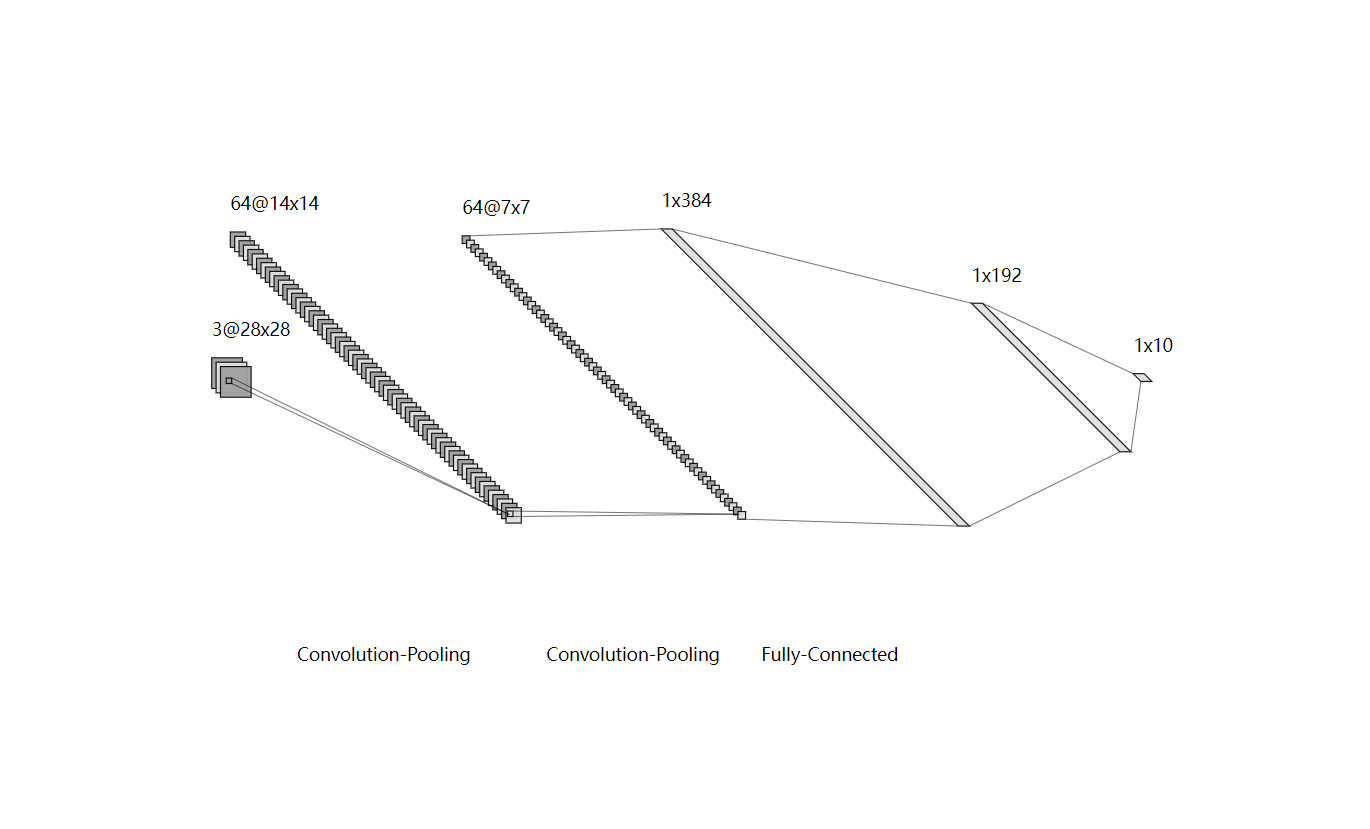
****

图 1tensorflow--cifar10分类神经网络框架

实验结果显示初始神经网络的分类测试准确率为**76.7%**。在初始神经网络基础上，又探索了对网络层加入**批次正则化（Batch\_Normal）**，网络训练参数更新过程中加入**梯度裁剪**方法，两种优化方式对网络进行训练并测试，得到最终结果如Table1，可见在不同的优化情况下，同时使用批正则化及梯度裁剪的方式得到的模型性能最优，测试准确率达**82.0%**。

|  |  |  |
| --- | --- | --- |
| **Table1在设置不同批正则化及梯度修剪下的网络分类准确度** | | |
|  | Batch\_Normal=False | Batch\_Normal=True |
| Gvs\_capped=False | 0.767 | 0.809 |
| Gvs\_capped=True | 0.797 | **0.820** |

**实验流程**

**1数据准备**

本程序使用Cifar-10的二进制数据集进行训练，在读取数据时，使用到了tensorflow官方对Cifar-10数据集进行读取的程序[cifar10\_input.py](to_teacher/cifar10_input.py)，分别读取并存储训练集、测试集的图片信息及对应的标签。

在加载数据时，特别的是，对训练数据加入了一系列**数据增强**操作，例如对图片随机裁剪、翻转、更改饱和度、对比度等。对输入数据如此操作实现了人为添加数据训练量，可以有效提高模型的**泛化能力**。

**2网络的构建及训练**

**2.1 网络构建**

分类网络的整体架构为：



图 2 网络整体框架

Tensorflow的重要部分是数据流图的构建，即构建算法的数学计算部分，搭建出算法框架，再代入数据进行计算。

由于tensorflow数据流图的构建时并未代入真实数据进行计算，因此在搭建算法框架时，输入数据部分则需要构建**占位符**进行填充，构建的占位符与真实的数据输入需要保持一致，在算法框架构建完成后再将真实数据填喂如占位符中进行运算。

本程序在神经网络构建时使用到了两层卷积网络，三层全连接层，激活函数为**ReLU**，在经过最后的三层全连接层处理后得到一个logits值，再经由softmax函数处理得到分类值。

在网络参数中，设置了计算的迭代次数为10000为了获得更佳的分类效果，而又为防止网络性能过拟合，又分别在两层卷积操作后加入了LRN层（局部响应归一化层，Local Response Normalization），大致网络构建结构如下。

****

图 3 神经网络结构

**2.2 网络训练**

网络训练中用到的损失函数是**交叉熵损失**，使用in\_top\_k函数判断预测正确数并计算准确率。本程序在网络训练中使用**Adam**优化子，求导器则是将损失函数降到最小。填喂数据时，cifar-10数据集已划分好了测试集和训练集，可直接调用填喂进网络中进行训练及测试。

**3训练结果**

在本程序中，设置了训练迭代次数为10000次，为能更清晰得看出网络训练过程中性能的提升，每迭代1000次输出训练的时间、损失及准确率，如Table2所示，可以看到训练过程中准确率在不断提升，最后的测试集准确率也达到了**76.7%**。

|  |  |  |  |
| --- | --- | --- | --- |
| **Table2 批正则化及梯度裁剪均未设置** | | | |
| Step | Time | loss | Accuracy |
| 1000 | 29.332 | 297.289 | 0.133 |
| 2000 | 131.997 | 121.365 | 0.656 |
| 3000 | 119.574 | 138.239 | 0.625 |
| 4000 | 132.786 | 106.238 | 0.703 |
| 5000 | 116.331 | 123.081 | 0.703 |
| 6000 | 136.348 | 107.926 | 0.688 |
| 7000 | 135.818 | 102.310 | 0.695 |
| 8000 | 127.426 | 90.466 | 0.734 |
| 9000 | 125.297 | 97.760 | 0.742 |
| 10000 | 169.077 | 98.088 | 0.742 |
| **Test** |  |  | **0.767** |

**4 网络优化**

在本实验中，还探究了在网络层构建中使用**批次正则化**，以及在网络参数的更新过程中加入**梯度裁剪**方式对网络分类性能的影响。

**4.1批次正则化**

**批次正则化（Batch\_Normalization）**是为了避免中间层变量的数值进入到后续激活函数的饱和区域，添加在每个中间层变量和激活函数之间，作用是让数据分布靠近激活函数的中段区间。

批次正则化对输入层和中间层做标准化处理，使得输出服从**正态分布**，从而避免分布偏移问题。在训练期间，我们仅通过计算当前层的一小批数据的均值和方差来标准化每一层的输入，因此称之为批次正则化。

在程序中，实现批次正则化的部分函数如下。

|  |
| --- |
| hidden2\_w = weight\_variable(shape=[5, 5, 64, 64], std=5e-2, w1=0) hidden2\_b = tf.Variable(tf.constant(0.1, dtype=tf.float32, shape=[64])) hidden2 = conv\_layer(hidden1, hidden2\_w, [1, 1, 1, 1], padding="SAME") hidden2 = tf.nn.bias\_add(hidden2, hidden2\_b) # BatchNormal,批正则化 hidden2 = tf.layers.batch\_normalization(hidden2, training=is\_train, trainable=True) |

批次正则化的优点是可以避免梯度消失问题，加快训练速度，提高模型泛化能力。本实验在网络层中加入了批次正则化再次训练得到的结果如Table3，最终的测试准确率达到**80.9%**，比为使用批次正则化准确率高出**4.2%**，可见对网络性能有一定提升。

|  |  |  |  |
| --- | --- | --- | --- |
| **Table3 设置批正则化，未设置梯度裁剪** | | | |
| Step | Time | loss | Accuracy |
| 1000 | 38.930 | 296.295 | 0.133 |
| 2000 | 141.166 | 132.515 | 0.695 |
| 3000 | 158.506 | 108.411 | 0.672 |
| 4000 | 146.047 | 105.697 | 0.727 |
| 5000 | 138.467 | 83.987 | 0.797 |
| 6000 | 143.497 | 89.616 | 0.805 |
| 7000 | 133.672 | 93.342 | 0.781 |
| 8000 | 182.625 | 90.454 | 0.773 |
| 9000 | 188.047 | 83.025 | 0.789 |
| 10000 | 153.216 | 79.139 | 0.820 |
| **Test** |  |  | **0.809** |

**4.2 梯度裁剪**

梯度裁剪是一种在网络反向传播过程中，将误差导数改变或剪裁到阈值，并利用裁剪后的梯度来更新权值的方法。

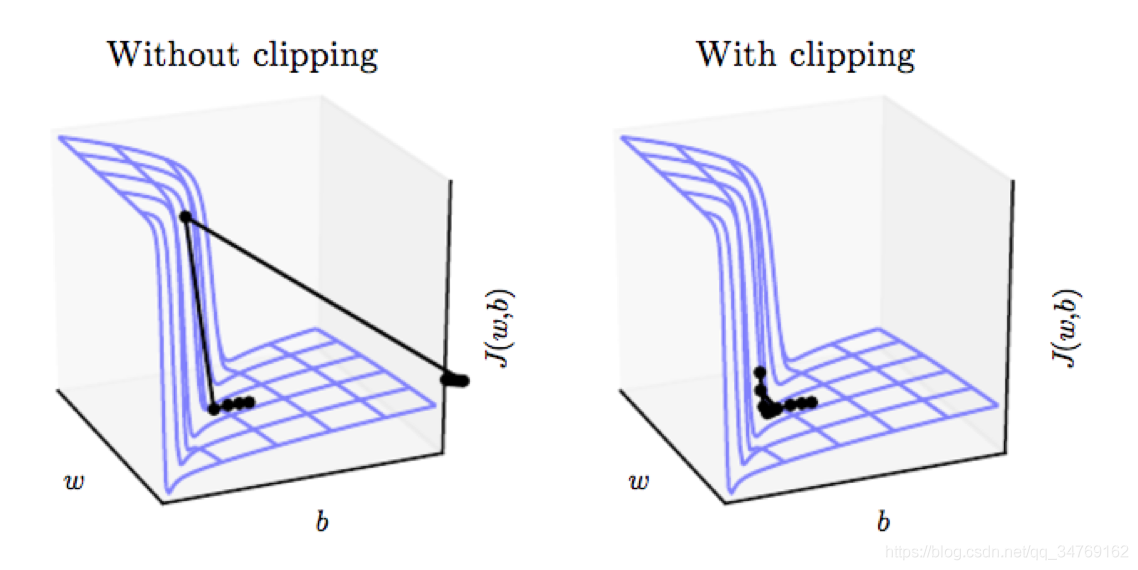


图 4 梯度裁剪

在模型训练过程中梯度过大导致模型不稳定，梯度消失则会使训练过程停止，两种情况都导致了网络参数的更新受阻。梯度裁剪则是在模型梯度过大时调节其保持较小的状态，以防止网络参数更新时梯度爆炸带来的影响。

在程序中实现梯度裁剪的部分代码如下。

|  |
| --- |
| # 获取损失函数 loss = loss\_func(logits, labels\_placeholder) # 设置优化算法使得成本最小 optimizer = tf.train.AdamOptimizer(1e-3) # 梯度裁剪 if gvs\_capped:  gvs, var = optimizer.compute\_gradients(loss)  capped\_gvs = [(tf.clip\_by\_value(grad, -1., 1.), var) for grad, var in gvs]  train\_step = optimizer.apply\_gradients(capped\_gvs) |

本实验在梯度更新过程中加入梯度裁剪操作后的网络训练及测试结果Table4，测试准确率达**79.7%**，相比初始网络准确率高出**3.0%**。

|  |  |  |  |
| --- | --- | --- | --- |
| **Table4设置梯度裁剪，未设置批正则化** | | | |
| Step | Time | loss | Accuracy |
| 1000 | 31.437 | 297.346 | 0.039 |
| 2000 | 160.407 | 164.662 | 0.539 |
| 3000 | 135.188 | 133.078 | 0.617 |
| 4000 | 125.859 | 122.782 | 0.656 |
| 5000 | 148.724 | 102.345 | 0.703 |
| 6000 | 160.912 | 79.817 | 0.789 |
| 7000 | 138.696 | 84.332 | 0.797 |
| 8000 | 122.696 | 101.746 | 0.711 |
| 9000 | 39.1031 | 71.033 | 0.805 |
| 10000 | 120.771 | 99.324 | 0.727 |
| **Test** |  |  | **0.797** |

**4.3 批次正则化+梯度裁剪**

最后在模型中同时设置批次正则化及梯度裁剪，可以得出测试结果为**82.0%**，在设置的所有模型中取得最高的准确率，且比初始模型高出**5.3%**，训练结果如Table5。

|  |  |  |  |
| --- | --- | --- | --- |
| **Table5 批正则化及梯度裁剪均设置** | | | |
| Step | Time | loss | Accuracy |
| 1000 | 40.510 | 296.628 | 0.102 |
| 2000 | 133.344 | 134.531 | 0.672 |
| 3000 | 122.875 | 87.107 | 0.797 |
| 4000 | 107.201 | 95.629 | 0.734 |
| 5000 | 114.446 | 117.882 | 0.688 |
| 6000 | 128.215 | 86.317 | 0.805 |
| 7000 | 125.349 | 86.748 | 0.789 |
| 8000 | 127.034 | 93.702 | 0.766 |
| 9000 | 123.808 | 75.918 | 0.820 |
| 10000 | 123.692 | 98.269 | 0.766 |
| **Test** |  |  | **0.820** |

**5 总结**

经过本实验的探究，充分理解了tensorflow的工作原理，其核心组成即数据流图的构建，即在构造部分将网络中的运算过程构建为数据流图的形式，在执行数据流图时，才会将数据代入到计算图中进行计算。实验过程中深刻体会到tensorflow较强的代码可读性，这体现在网络构建及训练过程、张量运算过程、数据代入过程清晰明了，是一个很强大且易用的深度学习工具。

除对tensorflow原理的理解之外，还对网络模型训练过程中，加入批次正则化的作用，以及在网络参数更新过程中梯度的裁剪对网络性能的提升有了深刻理解。

**批次正则化**作用是对输入层和中间层做标准化处理，使得输出服从正态分布，从而避免分布偏移问题，可以避免梯度消失问题，加快训练速度，提高模型泛化能力。**梯度裁剪**作用是在网络梯度过大时减小到一定值避免出现梯度爆炸问题，两种优化方式都能提升网络性能，且在同时设置时对网络性能提升最大。