致谢：

首先要感谢我的指导老师张信明教授，从毕业设计题目的选定，到中期检查时候的方向的微调和后期的论文完成过程中，张老师的悉心教导令我受益匪浅。

其次要感谢中国科大超级计算中心，在前期的工作中，由于自己的笔记本性能不足，在学习过程中实践难度很大，超算中心提供了平台的支持。

还要感谢我的好朋友好同学，前期的实践不仅仅依靠超算中心，也依靠他，借用他的笔记本进行了不少运算。

更要感谢的是我的父母，养育之恩终身难以回报，更是提供了资金更换笔记本电脑，让我得以在自己的计算机上继续毕业设计，免去了超算中心的排队和借用同学电脑的不便之处。

最后感谢这个城市，这所学校，还有身边的同学、朋友，提供了一个合适的平台环境，让我学习成长。

摘要：

1. 绪论
   1. 背景
   2. 对抗生成网络
   3. 本文章节安排

本文对内容做如下安排：

第一章 绪论。介绍对抗生成网络的相关背景以及本人所做毕设的方向。

第二章 对抗生成网络基础。从神经网络入手，介绍神经网络的思想和部分算法，接着是卷积神经网络的算法原理，以及生成对抗网络的基础。

第三章 Wasserstein GAN的关键技术。首先介绍的是本人学习调研的一些生成对抗网络的改进思想，其次着重介绍Wasserstein GAN的改进方法，包括Wasserstein距离的优势和Wasserstein GAN的优势。

第四章 Wasserstein GAN在图像生成上的改进。本章所述为毕设中期的工作，是在图像生成方面对Wasserstein GAN的修改。

第五章 Improved Training of Wasserstein GANs 所做改进。本章所述是提出Wasserstein GAN的作者Martin Arjovsky于三月31日提出的gradient penalty改进。以及结合本人毕业设计之后的网络设计方案。

1. 对抗生成网络基础
   1. 神经网络基础

神经网络是一种模拟生物神经元结构和功能的模型，典型的神经网络模型包含结构、激活函数和学习规则三部分，其作用是通过简单的矩阵运算和非线性的激活运算来对数据进行处理。

下面所述的是一个简单的神经网络。



图1-1 神经网络示意图

如图1-1所示，这是一个简单的神经网络，包含两个输入变量、，三个中间变量、、，一个输出变量。其中所在为输入层，所在为隐藏层，所在为输出层。和被称为权重。计算方式为：

， (1.1)

例如：

(1.2)

为了表达和计算上的方便，用矩阵来表示如上运算过程：

 (1.3)

在实际的运用中，往往还会加上一个偏置项bias，则1.1、1.2、1.3式转变为如下：

， (1.4)

(1.5)

.  (1.6)

对于图1-1，没有表现出偏置项，在实现中，偏置项有多种方式可以添加进去，一种是如式1.6加上一个向量，还可以将偏置项视作1\*w，将偏置项的加法运算放入矩阵相乘中，式1.6转化为：

 (1.7)

其中。这种方式在计算上省去了一步向量相加，但是多出了一步向量的拼接操作，将拼接为了。如图1-2：



图1-2 神经网络示意图，带偏置项bias

在实际的运算中，拼接操作比向量相加消耗更多的计算。于是出现了又一种计算方法。考虑到每次拼接的开销，不妨考虑只在输入增加一个值为1的神经元，之后的除了最后一层的每一层，都附加一个神经元，但是该神经元的值由之前的计算得到，如图1-3：



图1-3 神经网络示意图，转变后的偏置项

相对应的计算公式为：

.  (1.8)

其中，作为隐藏层的偏置项参与运算。

神经网络里面一个举足轻重的反向传播算法back propagation，其作用在于调整网络使之成为我们所需要的状态，简单的来说就是复合函数的链式求导，用梯度下降的方法更新网络，有兴趣的可以参考！！！

* 1. 卷积神经网络

在后文中用到了另一个重要的网络层是卷积层，熟悉图像处理的人知道，卷积操作在图像中十分常见，对图像的平滑、模糊、锐化、边缘检测、浮雕等操作都可以通过特定的卷积算子来实现。下面简略的介绍卷积在图像中的操作方式。

卷积在图像中的操作方式往往是一个2D卷积，对于一张图像和一个卷积核，如图1-4所示，左边是一个5x5的图像，右边是一个3x3的卷积核。



图1-4 图像卷积示意(1)

有一个很形象的说法，将卷积核视为窗口，在图像上滑动，得到卷积后的结果。举个例子，在图1-4中，将卷积核置于图像的左上角，那么运算方式为，每个像素点的值乘以卷积核上与像素位置对应的值，最后累加得到结果。



图1-5 图像卷积示意(2)

运算的式子为：



然后将卷积核向右移动一个，如图1-6：



图1-6 图像卷积示意

对应的运算为：



将卷积核在图像上全部滑动一遍后，得到一个3x3的新图像。这里图像变小了，若是想维持图像大小不变，则在原图像四周补上数据，补充的数据的方式有多种，最简单的是用0填充，卷积操作连边缘都扫过，则可以得到和原图一样大小的图像。上述的卷积的滑动向左和向下的步长都是1，实际上步长还可以为更大的值，比如，步长为2，边缘补上数据，则卷积后的图像大小的长和宽都是原图像的一半。

在实际的运算中，卷积运算会转换成矩阵相乘来实现，首先对原图像和卷积核都做展开，上述的例子则可以转变成：



对结果改变形状成3x3的就得到了最后的结果。

那么，卷积的好处是什么，第一点就是节省空间，若是将一个5x5的图像通过全连接的方式转变成3x3的图像，所谓的全连接，是指上下两层每两个结点都有关联，那么需要权重25x9=225个，如果使用卷积，那么只需要9个，为了充分的获取图像的特征，可以用更多的卷积核来处理同一个图像，即使是这样，也比全连接节约了很多空间。第二点，图像具体局部性，我们在关注图像的时候，通过某一小块区域得到一个信息，这个信息和距离该区域很远的区域关系并不密切，卷积操作很好的关注了局部性的特征。

简略的提一下反卷积操作，反卷积往往用来数据的扩充，卷积和反卷积的输入输出过程刚好相反，卷积的前向计算过程是反卷积的反向传播过程，卷积的反向传播过程是卷积的前向计算过程。

* 1. 生成对抗网络

三年前蒙特利尔大学 Ian Goodfellow 等学者提出“生成对抗网络“(Generative Adversarial Networks，GANs)的概念，去年在学界、业界，人们对GANs的关注突然增加。

什么是生成对抗网络，Ian Goodfellow给出的描述是“生成对抗网络是一种生成模型（Generative Model），其背后基本思想是从训练库里获取很多训练样本，从而学习这些训练案例生成的概率分布。”

对抗生成网络包含两个网络，一个是判别器Discriminator，一个是生成器Generator。生成器将输入的噪声生成和样本一样格式的数据，判别器则是要区分真实数据和生成器生成的伪造数据。传统的GAN通过如下的方式训练：

判别器接收一组真实数据real data，进行计算之后的结果和1进行比较，计算差距，记为判别器对真实数据的损失函数dis\_loss\_real。生成器生成一组伪造数据fake data，传入判别器进行计算，得到的结果和0进行比较，计算差距，记为判别器对伪造数据的损失函数dis\_loss\_fake；得到的结果和1进行比较，计算差距，记为生成器的损失函数。直观上的理解则是判别器想尽可能的区分真实数据和伪造数据，所以想得到对真实数据判定为1、对伪造数据判定为0的结果；生成器则是希望自己生成的数据被认定为真实数据1。



图1-7 生成对抗网络示意

由上述的描述可以得知，生成器需要联合判别器一起才能训练生成器，判别器也需要生成器给提供fake data。训练生成器的时候，判别网络不做更新；训练判别器的时候，生成网络不做更新。两者交替训练，达到生成器生成的数据可以以假乱真的地步。

关于生成对抗网络，已经有很多的改进的版本，比如ali-bi gan、boundary equilibrium gan、conditional gan、infogan等等，它们大都是在损失函数的计算方式和网络结构上做了一些改动，本质上没有改变GAN的性质。

1. Wasserstein GAN的关键技术
   1. 以往的生成对抗网络的改进方案

先看对损失函数的改进方案。考虑原始的GAN，判别器最后的输出是一个数据对应一个输出值，对这个值做sigmoid激活，结果和0或者1比较计算交叉熵，在tensorflow中，使用的是sigmoid\_cross\_entropy\_with\_logits函数，传入的两个参数，logits是判别器输出的结果（不需要sigmoid激活，因为这个函数自带sigmoid激活），labels是同等规模的0或者1。

在调研中，发现对此的改进主要有三个方向，一种是使用sigmoid激活之后用其他函数来处理，第二种是对输出的结果直接做处理，第三种，引入其他的度量标准，修改损失函数。

第一种以ali bigan为例，sigmoid激活后的结果在0~1之间，用log函数可以拉开差距，设sigmoid激活后的结果为X，判别器用的是log(X\_real)+log(1-X\_fake)，生成器用的是log(X\_fake)+log(1-X\_real)，对结果计算均值之后取负号，最小化损失函数，则可以起到对抗训练的效果，判别器想让X\_real向1靠近，让X\_fake向0靠近，生成器则做相反的事情。在Least Square GAN中，对判别器输出的sigmoid激活后的结果，做的操作是：判别器对真实结果减去1再平方，对伪造结果直接平方，生成器对伪造结果减去1再平方，最后的思想还是在于判别器让真实数据的结果逼近1，伪造数据的结果逼近0，生成器让伪造的结果逼近1。

第二种，对输出的结果直接处理的，有Auxiliary Classifier GAN，真实数据是有不同的标签的，对标签进行独热码编码，将判别器输出的结果进行softmax之后，再与编码后的标签计算交叉熵。判别器对真实数据和伪造数据以及对应的标签，都希望得到正确的分类，但是，这样一来无法区分真实数据和伪造数据，于是还需要最小化伪造数据经过判别器之后的结果，这就要求判别器有两组输出，一组是分类，长度为独热码的长度，另一组是一个值，经sigmoid激活之后和前面用相似的方法，判别器需要让伪造数据的第二种输出尽可能接近0，生成器则是希望分类正确并且输出接近1。

至于第三种，方法比较杂乱，有像MaGAN那样给一个阈值m，来对fake data的判断结果做限定，也有像Mode Regularized GAN那样给出通过比较真实数据和伪造数据的差距，来优化网络的。因为方法繁多，并且还使用了其他的优化，在此不做详细说明。

除了改进损失函数的计算方法，另一大类是对网络做修改，并且两种方法往往结合使用。在修改网络的优化中，不得不提的是Conditional GAN，即条件生成对抗网络。一大特点就是将数据的标签也引入到判别器和生成器中，与Auxiliary Classifier GAN中使用标签的方法不同的是，Condition GAN在数据的输入中就加入标签，甚至会在网络的内部层次中将标签数据拼接上去，达到引入数据和控制网络方向的作用。举一个具体的例子，在手写数字的生成中，对于判别器，输入的是图像和标签，对标签往往会进行独热码标签，一方面是加强标签的作用，另一方面是便于数据的拼接操作。在图像和标签的共同作用下，判别器把真实数据判定为1。对于生成器，噪声和标签作为输入，得到的是图像，再将图像和对应的标签放入判别器，训练生成器让判断结果接近1，训练判别器则让结果接近0。DCGAN在手写数字生成上效果很好，生成的数字肉眼几乎无法判断是真是假，其中有个优化就是引入了标签。

* 1. Wasserstein距离的优势

Wasserstein距离又叫Earth-Mover（EM）距离，定义如下：

 (3.1)

看起来很复杂的定义需要解释，表示和组合的所有可能的联合分布的集合，意味着中，每一个分布的边缘分布都是和。对于每一个可能的联合分布来说，可以从中采样得到一个真实样本和一个生成样本，并计算这对样本的距离，因此可以计算该联合分布下样本对距离的期望值。在所有可能的联合分布中对这个期望取下界，定义成Wasserstein距离。

上面的说法确实晦涩难懂，关于EM距离，有个形象的说明，在这个“路径规划”下，把这堆“沙土”挪到位置所需要的消耗看作，那么则是“最优路径规划”下的“最小消耗”，故此成为Earth-Mover（推土机）距离。

在原始的对抗生成网络中，当判别器近似最优的时候，最小化生成器的损失函数等价于最小化和之间的JS散度。问题是，和之间几乎不可能有不可忽略的重叠，所以JS散度为常数log2，对常数的求导有个显然的结果，这样以来生成器的梯度近似为0了，出现梯度消失问题，无法继续有效的训练下去。

原始的GAN不稳定，原因是要么判别器训练的太好，生成器梯度消失严重，生成器的loss降不下去；要么是判别器训练的不够，生成器梯度不准确，导致梯度下降的方向乱跑。故此需要判别器不好不坏的，才能让生成器有个稳定并且不消失的梯度，找到这个不好不坏的判别器很困难，训练的前期和后期都可能不一样，也就导致设想很好的GAN训练起来很困难。

那么Wasserstein距离的优势在于，相比JS散度，即使和分布没有重叠，它都可以反应它们的远近。并且，相比JS散度的突变，Wasserstein距离是平滑的，能够提供梯度。

尽管Wasserstein距离很有优势，但是实际的计算中，没法直接求解，作者用一个已有的定理将Wasserstein距离转变为了如下形式：

 (3.2)

式(3.2)表示，在保证函数的Lipschitz常数（存在常熟，使得在定义域内，连续函数，任取，满足）不超过K的条件下，对所有可能满足条件的函数取到的上界，然后再除以。式(3.2)变成：

 (3.3)

利用神经网络强大的拟合能力，找到一系列的足以高度近似公式3.2要求的不是难事。至于，对权重做范围的限制，即令所有的参数不超过范围[-c,c]即可保证关于输入样本的导数也不会超过某个范围。在tensorflow中，使用clip\_by\_value函数对权重进行限制。

* 1. Wasserstein GAN的优势

之前提到的，原始GAN的问题有：判别器越好，生成器梯度消失越严重。最小化生成器的损失函数loss，会等价于最小化一个不合理的距离衡量，导致两个问题，梯度不稳定和多样性不足。这里的多样性指的是和真实数据相比，生成的结果可能偏向于某一种或者少量的几种，例如手写数字的生成中，原始的GAN在没有标签控制生成方向的时候，结果会向着特征简单的方向生成，比如生成的很多都是1，而那些特征复杂的数字，生成的就很少。

关于手写数字mnist的数据集，已经有效果较好的深度卷积生成对抗网络DCGAN来生成数据，但是由于网络相对复杂，训练的耗时较长。在下一章将会讲到本人在一些网络结构上进行的实验以及优化。

Wasserstein GAN与原始的GAN相比，只修改了四点：

* 判别器的最后一层去掉sigmoid
* 生成器和判别器的loss不取log
* 每次更新判别器的参数之后把它们的绝对值截断到不超过一个固定常熟c
* 不要用基于动量的优化算法（包括momentum和Adam），推荐使用RMSProp，SGD也行。

那么Wasserstein GAN的优势在于什么呢，在实验中发现，对于已有的较为成熟有效的生成对抗网络模型中，在不改变网络结构的情况下，使用Wasserstein GAN可以达到相同的效果。但是，较为成熟的生成对抗网络模型中，使用了很多的优化方法，诸如归一化等操作。比如DCGAN中，使用了反卷积和归一化，并且引入标签来补充数据，去掉这些，模型将会无法正常工作，但是，使用Wasserstein GAN，即使很简单的全链接网络，没有归一化，简单的leaky relu激活，就可以生成较好的图片。此外，在没有标签参与的情况下，多数GAN都出现了多样性不足的情况，Wasserstein GAN却避开了这一点。此外，Wasserstein GAN的损失函数可以指示训练进度，传统的GAN的损失函数上下抖动的较为剧烈，无法作为指示。

1. Wasserstein GAN在图像生成上的改进

由于神经网络网络结构和损失函数以及参数的复杂性，下述在进行对比实验的时候，均是经过大量调参，找到尽可能适合当前网络模型的参数。在使用不同的损失函数时，对学习率相同的限制没有意义。对于GAN生成的图像效果，尚未找到合适的衡量标准，将以本人和同学的主观感受作为衡量标准，判断生成图像的质量。

* 1. 深度卷积神经网络识别验证码

正如程序员的第一课是hello world，神经网络的第一课是手写数字识别，使用的数据集是mnist，本人在实践了caffe训练并识别mnist数据集之后，修改网络结构，做学校教务系统验证码的识别，并移植到tensorflow上，因为python语言的灵活性，以下全部用tensorflow作为框架来描述。

1. 网络构建

首先先描述数据集的特征，验证码图片从<http://mis.teach.ustc.edu.cn/randomImage.do>这个地址抓取，这个地址是原先学校教务系统本科生的地址，现在已经弃用，先抓了一百张，每一张验证码由四个字母组成，图像大小是20x80的，水平四等分之后差不多能把验证码分隔开。验证码包含数字和大写字母，除了数字0、1和字母I、O，也就是一共32种字符，用python将验证码转化为灰度图做阈值分割并且四等分之后，手动给分割后的验证码分类，并制作标签，用python转化成mat文件格式（matlab的文件格式）。

C:\Users\monk\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tmp.jpg

图4-1 验证码原图示例

C:\Users\monk\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1.jpg C:\Users\monk\AppData\Local\Microsoft\Windows\INetCache\Content.Word\2.jpg C:\Users\monk\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3.jpg



图4-2 验证码初步处理并做分割后的图片

参考lenet对手写数字做分类的网络模型，我自己的网络模型如下：

1. 数据输入：20\*20\*1的图像，20\*20是图像尺寸，1表示图像是单通道的灰度图
2. 第一层卷积层，卷积核大小是5\*5，卷积核个数20，卷积步长长和宽两个方向上都为1，卷积的结果加上偏置项，偏置项的长度和卷积核个数相同。一张图像经过20个卷积核之后，得到20张图像，每一张图像都加上各自的偏置项，一张图的所有像素点的数据所加的值相同。卷积的设置是不填充边界的，因此卷积之后的结果规模是16\*16\*20。
3. 第二层激活层，对数据进行relu激活，即只保留正数部分，负数部分置为0，数据规模不变化。
4. 第三层池化层，使用最大化池化，池化的核大小为2\*2，步长为2，意味着图像缩小为原来的四分之一，并且每个像素点对应的是原来4个像素点中的最大值。此时数据规模是8\*8\*20。
5. 第四层卷积层，卷积核大小仍是5\*5，卷积核个数50，步长为1，得到的结果仍旧加上偏置项。每个核都对20个从上一层接收的8\*8的图像做处理，将20张图像按权重线性累加得到当前卷积核输出的结果，累加操作由tensorflow框架内部实现。卷积图像不填充边界，得到的数据结果规模是4\*4\*50。
6. 第五层激活层，relu激活，数据规模仍是4\*4\*50。
7. 第六层池化层，池化方式和第四层相同，输出的数据规模结果是2\*2\*50。
8. 第七层全连接层，将第六层得到的数据转化为一个规模为2\*2\*50=200的一维向量，叉乘一个200\*500的矩阵，加上长为500的偏置项，得到长为500的一维向量。
9. 第八层激活层，relu激活，只保留正数，负数置为0。输出的数据规模是一维的长500的向量。
10. 第九层全连接层，上层结果叉乘500\*32的矩阵之后加上长32的偏置项，最后得到长32的一维向量。
11. 第十层softmax层，对上层结果做softmax回归，得到类似独热码的结果，其中一个数较大，接近1，其他数较小，接近0，数据都在[0,1]之间。
12. 接下来计算损失函数，将进行独热码编码后的真实标签与取对数后的上一层结果相乘并累加，取相反数作为损失函数。
13. 以上是训练的网络，训练过程中计算准确率，将第十层的结果中最大的数的索引和独热码编码后的真实标签中为1的数的索引进行比较，统计相等的个数。
14. 关键程序实现

训练模型的输入为图像数据和标签。以下按照本节a）中所述展示关键代码。

1. x = tf.placeholder(tf.float32, [**None**, 400], **"x"**)

y = tf.placeholder(tf.float32, [**None**, 32], **"y"**)

h0 = tf.reshape(x, shape=[-1, 20, 20, 1], name=**"reshape"**)

1. h1 = tf.nn.conv2d(h0, filter=W1, strides=[1, 1, 1, 1], padding=**"VALID"**) + B1
2. h2 = tf.nn.relu(h1)
3. h3 = tf.nn.max\_pool(h2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding=**"VALID"**)
4. h4 = tf.nn.conv2d(h3, filter=W2, strides=[1, 1, 1, 1], padding=**"VALID"**) + B2
5. h5 = tf.nn.relu(h4)
6. h6 = tf.nn.max\_pool(h5, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding=**"VALID"**)
7. h7 = tf.reshape(h6, shape=[-1, 2 \* 2 \* self.dim\_2], name=**"reshape"**)

h8 = tf.matmul(h7, W3) + B3

1. h9 = tf.nn.relu(h8)
2. h10 = tf.matmul(h9, W4) + B4
3. y\_ = tf.nn.softmax(h10)
4. loss = -tf.reduce\_sum(y \* tf.log(y\_))
5. accuarcy = tf.reduce\_mean(tf.cast(tf.equal(tf.argmax(y\_, axis=1), tf.argmax(y, axis=1)), tf.float32))

训练使用AdamOptimizer算法梯度下降最小化loss。

train\_operate = tf.train.AdamOptimizer(learning\_rate).minimize(loss)

训练过程如下

sess.run(train\_operate, feed\_dict={x: input\_x, y: input\_y})

1. 实验结果

使用tensorflow自带的工具输出了loss和准确率：



图4-3 训练过程中的损失函数loss和准确率

可见，本次的训练结果非常好，损失函数接近0，准确率近乎为1。

保存模型并另抓取验证码图片测试：



图4-4 抓取的验证码图片和识别结果

实际上，在实验中，最初获取的数据较少，约400张，进行手工的分类，用手动分类之后的结果做训练，准确率大概为92%。抓取更多的数据进行分类，再人工检查，修正一些错误，用新的数据进行训练，准确率继续上升。在准确率达到97%左右的时候，用模型去检测数据集，又找出了之前没有发现的数据分类的错误。直到准确率达到近乎100%，已经超过人的识别准确度。

* 1. 深度卷积网络生成验证码

上一节详细的讲述识别验证码所用的卷积神经网络CNN的结构和具体细节，为本节的深度卷积DCGAN生成对抗网络打了一个良好的基础。

在CNN的识别过程中，用到了池化层，将2x2的像素缩减为了1x1的像素，只保留最大值，故此损失了许多数据。为了减少数据的损失，将使用步长为2的卷积代替CNN中的卷积和池化操作。

对于对抗生成网络GAN，判别器和生成器没有规定要结构相逆，但是，实际操作中我们构建的判别器和生成器结构相逆，不仅仅是拥有更清晰的逻辑，视生成器为判别器的逆过程，而且，效果往往较好。原因是一个良好的判别器既然能够做到良好的识别，则意味着对特征的捕捉很到位，那么和它结构相逆的网络在生成上一定程度上会还原相应的特征。

基于上述，首先所做的是DCGAN的实现。

1. 网络构建

判别器的网络结构：

1. 数据输入和上一节中相同，但是不包含标签
2. 第一层卷积层，卷积核大小5\*5，个数64，步长为2，用0填充边界，得到规模为10\*10\*64的数据。
3. 第二层激活层，使用leaky relu激活，正数部分保持不变，负数部分乘以一个微小量，本程序设置为0.2
4. 第三层卷积层，卷积核大小5\*5，个数128，步长为2，用0填充边界，得到规模为5\*5\*128的数据。
5. 第四层激活层，使用leaky relu激活。
6. 第五层全连接层，将上层得到的数据排列成长为5\*5\*128=3200的一维向量，叉乘3200\*1024的矩阵，得到长1024的一维向量。
7. 第六层归一化层，对数据进行归一化操作。
8. 第七层激活层，使用leaky relu激活。
9. 第八层全连接层，上层得到的数据叉乘1024\*1的矩阵，得到长为1的向量。
10. 第九层激活层，使用sigmoid函数激活

生成器的网络结构：

1. 数据输入为长为128的噪声向量。
2. 第一层全连接层，噪声向量叉乘128\*1024的矩阵，得到长为1024的一维向量。
3. 第二层归一化层，对上层得到的数据进行归一化操作。
4. 第三层激活层，使用relu激活
5. 第四层全连接层，上层数据叉乘1024\*3200的矩阵，得到长为3200的一维向量。
6. 第五层归一化层，对上层得到的数据进行归一化操作。
7. 第六层激活层，使用relu激活
8. 第七层反卷积层，将上层得到的数据重新排列成5\*5\*128的三维矩阵，对矩阵进行反卷积操作，卷积核大小5\*5，共有128个卷积核，使用卷积反向传播的算法得到64张图像，卷积步长为2，得到的数据规模为10\*10\*64的三维矩阵。
9. 第八层归一化层，对上层得到的数据进行归一化操作。
10. 第九层激活层，使用relu激活
11. 第十层反卷积层，卷积核大小5\*5，个数64，输出1，步长为2，得到大小为20\*20\*1的数据
12. 第十一层激活层，使用sigmoid激活，使数据范围为[0,1]

损失函数的计算：

将一组噪声向量输入生成器，得到一组生成的图片，将这组图片放入判别器，输出结果记为fake；将一组真实图像放入判别器，输出的结果记为real。

判别器损失函数由两部分组成，第一部分对真实图片的判别准确程度，将real与一组1进行比较计算交叉熵，第二部分对伪造图片的判断准确成都，将fake与一组0进行比较计算交叉熵，两部分加起来得到判别器的损失函数。

生成器的损失函数是将fake与一组1进行比较计算交叉熵，体现了生成器“造假”的程度。

1. 关键程序实现

判别器主要程序：

1. dis\_h0 = tf.nn.conv2d(image, self.dis\_W1, strides=[1, 2, 2, 1], padding=**"SAME"**)
2. dis\_h0 = lrelu(dis\_h0)
3. dis\_h1 = tf.nn.conv2d(dis\_h0, self.dis\_W2, strides=[1, 2, 2, 1], padding=**"SAME"**)
4. dis\_h1 = lrelu(dis\_h1)
5. dis\_h1 = tf.reshape(dis\_h1, [-1, self.dim\_dis\_2 \* (self.shape[0] // 4) \* (self.shape[1] // 4)])  
   dis\_h2 = tf.matmul(dis\_h1, self.dis\_W3)
6. dis\_h2 = batchnormalize(dis\_h2)
7. dis\_h2 = lrelu(dis\_h2)
8. dis\_h3 = tf.matmul(dis\_h2, self.dis\_W4)
9. dis\_h3 = tf.nn.sigmoid(dis\_h3)

生成器主要程序：

1. gen\_h0 = tf.matmul(z, self.gen\_W1)
2. gen\_h0 = batchnormalize(gen\_h0)
3. gen\_h0 = tf.nn.relu(gen\_h0)
4. gen\_h1 = tf.matmul(gen\_h0, self.gen\_W2)
5. gen\_h1 = batchnormalize(gen\_h1)
6. gen\_h1 = tf.nn.relu(gen\_h1)
7. gen\_h1 = tf.reshape(gen\_h1, [-1, self.shape[0] // 4, self.shape[1] // 4, self.dim\_gen\_2])  
   gen\_h2 = tf.nn.conv2d\_transpose(gen\_h1, self.gen\_W3, [batch\_size, self.shape[0] // 2, self.shape[1] // 2, self.dim\_gen\_3], strides=[1, 2, 2, 1])
8. gen\_h2 = batchnormalize(gen\_h2)
9. gen\_h2 = tf.nn.relu(gen\_h2)
10. gen\_h3 = tf.nn.conv2d\_transpose(gen\_h2, self.gen\_W4, [batch\_size, self.shape[0], self.shape[1], self.channel], strides=[1, 2, 2, 1])
11. gen\_h3 = tf.nn.sigmoid(gen\_h3)

损失函数的计算：

real\_image = tf.reshape(x, [-1, self.shape[0], self.shape[1], self.channel])  
fake\_image = self.generate(batch\_size, z)  
real = self.discriminate(real\_image)  
fake = self.discriminate(fake\_image)  
dis\_loss\_real = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=tf.clip\_by\_value(real, 1e-7, 1. - 1e-7), labels=tf.ones\_like(real)))  
dis\_loss\_fake = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=tf.clip\_by\_value(fake, 1e-7, 1. - 1e-7), labels=tf.zeros\_like(fake)))  
dis\_loss = dis\_loss\_real + dis\_loss\_fake  
gen\_loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=tf.clip\_by\_value(fake, 1e-7, 1. - 1e-7), labels=tf.ones\_like(fake)))

训练过程使用AdamOptimizer梯度更新的方法最小化判别器和生成器的损失函数。

train\_op\_dis = tf.train.AdamOptimizer(learning\_rate, beta1=0.5).minimize(dis\_loss, var\_list=dis\_vars)  
train\_op\_gen = tf.train.AdamOptimizer(learning\_rate, beta1=0.5).minimize(gen\_loss, var\_list=gen\_vars)

训练过程如下：

sess.run(train\_op\_dis, feed\_dict={x: input\_x, z: input\_z})

sess.run(train\_op\_gen, feed\_dict={z: input\_z})

经过实验得到的结果是判别器和生成器的训练比例是1:5，只代表此网络的训练判别器的生成器的训练比例。

1. 实验结果



图4-5 DCGAN的损失函数状态

从损失函数中并不能看出训练的效果，此时耗时184秒，在第800次迭代的结果如图4-6：



图4-6 DCGAN生成验证码示例

可见此时已经可以看出大部分的验证码，效果尚可，倘若继续训练下去，则会因为判别器训练的过好，生成器梯度消失，无法生成验证码。800之后的判别器的损失函数变换不明显，图像生成不准确，图4-7表示了在第1260次迭代（左）和第1480次（右）的结果。



图4-7 DCGAN生成验证码示例

* 1. Wasserstein GAN生成验证码

在3.3节中提到，即使是使用简单的全连接网络，Wasserstein GAN（以下简称WGAN）都有很好的效果。下面的实验包含两个，一个是使用简单的全连接网络进行验证码的生成，另一个是基于4.2中的DCGAN进行修改后的网络进行验证码的生成。

首先是全连接网络。

判别器的网络结构：第一层将图像转变形状为一个长400的一维向量，叉乘400\*500的矩阵之后加上一个长500的偏置项，得到长500的一维向量，然后使用relu激活；第二层全连接到长为1的一维向量并加上偏置项，作为判别器的输出。

生成器的网络结构：第一层将噪声向量全连接成长500的一维向量并加偏置项，进行relu激活；然后全连接成长400的一维向量并加偏置项；最后使用sigmoid激活。

整个网络十分简单，没有复杂的卷积、反卷积，没有归一化操作，生成器和判别器只有简单的三四层，因此每次网络的迭代都非常快。

损失函数的计算是WGAN的重点，记real为真实数据经过判别器之后输出的结果，记fake为伪造数据经过判别器之后输出的结果。那么判别器的损失函数为fake的均值减去real的均值，生成器的损失函数为fake的均值的相反数。使用的tensorflow的程序为：

dis\_loss = tf.reduce\_mean(fake) - tf.reduce\_mean(real)  
gen\_loss = -tf.reduce\_mean(fake)

同时为了满足式3.3中关于Lipschitz常数的限定，，对判别器网络中的参数进行了范围限制，其中限制的范围为0.05，这个数值也是需要多次实验才能得到的。程序为：

clip\_dis = [p.assign(tf.clip\_by\_value(p, -0.05, 0.05)) **for** p **in** dis\_vars]

其中dis\_vars表示的是判别器中参与计算的参数，包括两层全连接网络的权重和偏置项。

梯度更新使用非动量的算法，程序为：

train\_op\_dis = tf.train.RMSPropOptimizer(learning\_rate).minimize(dis\_loss, var\_list=dis\_vars)  
train\_op\_gen = tf.train.RMSPropOptimizer(learning\_rate).minimize(gen\_loss, var\_list=gen\_vars)

训练过程为：

sess.run([train\_op\_dis, clip\_dis], feed\_dict={x: input\_x, z: input\_z})  
sess.run([train\_op\_gen], feed\_dict={z: input\_z})

和前面的网络相比，判别器的训练多出了clip\_dis操作，即对参数范围的限定。

判别器和生成器的训练比例设置，经实验选择了3:1。



图4-8 损失函数变化趋势，对数纵坐标。



图4-9 WGAN生成示例

大约800次迭代之后，判别器的距离趋近Wasserstein距离，两个损失函数都开始稳步下降。6000次之后趋于平缓，意味着网络已经训练不动。经过8500次迭代，耗时171s，得到的图像如图4-9所示，人肉眼能识别少部分的图像，在使用简单的网络结构就可以达到这种效果，Wasserstein GAN确实很有优势。

对比实验，使用原始的GAN的损失函数以及相同的网络结构，无论怎样调参，生成的结果都是千篇一律的样式，并且很容易出现梯度消失训练不动的情况。生成的图像组合在一起仿佛是墙纸花纹，之前在某处看到的关于GAN功能的描述，在生成花纹和重复样式上效果很好。如图4-10：



图4-10 GAN生成示例

对比了全连接网络，下面使用Wasserstein GAN的方式对4.2中的DCGAN进行改进。

在网络结构不变的情况下，只修改了损失函数和梯度更新的方式，并增加了对判别器中参数的范围限制。判别器和生成器训练比例为1:5。称此网络为WDCGAN

在第340次迭代，耗时77秒时，生成图像如图4-11（左）所示，效果已和4.2中DCGAN在第800次迭代、耗时184秒时效果(图4-11右)接近。



图4-11 WDCGAN与DCGAN对比

在第460次迭代、104s时，效果已经很可观，如图4-11，此时不仅在分辨程度上相比DCGAN有了提升，对比度也增强了许多，用时上也更短，足以表现出Wasserstein GAN的优势。

使用Tensorflow自带的工具绘制损失函数变化如图4-12所示，从第500次迭代之后，判别器损失函数下降减缓，生成器的损失函数平稳，略有上升，此时应该增加生成器的训练次数，考虑到此时生成的效果，后续的调整不再做。



图4-11 WDCGAN生成效果示意



图4-12 WDCGAN损失函数

* 1. 对Wasserstein GAN的优化

考虑Wasserstein GAN中做的优化方案，对选择Wassertein距离作为损失函数深表叹服，同时在多次实验之后发现问题，第一，若是单纯的使用全连接网络，将会因全连接网络参数规模大而过于消耗内存，需要保存的权重和偏置项很多，并且生成的图像不够平滑，噪点很多。第二，使用现有的较好的模型，只修改判别函数，单次的训练速度会下降，不足以充分利用Wasserstein距离的优越性。第三，对于复杂的但是共性很多的图像，如celeba的人脸数据集，使用较为简单的网络，如全连接网络，仍会出现多样性不足的情况。第四，经过限制范围的判别器网络的参数，将会出现两极分化的情况，近似的变成了二值网络，没有充分利用网络的泛化能力。

故此，本人试图对Wasserstain GAN做一些改进。

对于全连接网络会出现噪声，一个原因是生成器中的某些参数可能过大或者过小，设想对判别器所做的限制也可以加到生成器里面去，以此来减少某些参数过大或者过小的情况，同时在最后一层增加一层卷积层来对图像起平滑的作用。

对WDCGAN做如下调整：

1. 对原生成器倒数第二层的反卷积操作，输出的从20\*20\*1的三维矩阵变成20\*20\*16的三维矩阵，最后一层的sigmoid激活改为tanh激活。增加一层卷积层，卷积核大小视图像而定，当前程序定为5\*5，卷积步长为1，边缘用0填充保证图像大小不变，卷积输出为20\*20\*1的三维矩阵。增加一层sigmoid激活层对上层结果激活。

gen\_h3 = tf.nn.conv2d\_transpose(gen\_h2, self.gen\_W4, [batch\_size, self.shape[0], self.shape[1], self.dim\_gen\_4], strides=[1, 2, 2, 1])  
gen\_h3 = tf.nn.tanh(gen\_h3)  
gen\_h4 = tf.nn.conv2d(gen\_h3, self.gen\_W5, [1, 1, 1, 1], **"SAME"**)  
gen\_h4 = tf.nn.sigmoid(gen\_h4)

1. 修改参数范围限制的操作。对判别器只限制全连接层的权重，不限制偏置项和其他。对生成器的反卷积和全连接的权重加以限制，其他不做限制，相应的训练过程也做修改。

clip\_dis = [p.assign(tf.clip\_by\_value(p, -0.05, 0.05)) **for** p **in** dis\_vars **if "dis\_f\_W" in** p.name]  
clip\_gen = [p.assign(tf.clip\_by\_value(p, -0.05, 0.05)) **for** p **in** gen\_vars **if "gen\_f\_W" in** p.name **or "gen\_dc\_W" in** p.name]  
sess.run([train\_op\_dis, clip\_dis], feed\_dict={x: input\_x, z: input\_z})  
sess.run([train\_op\_gen, clip\_gen], feed\_dict={z: input\_z})

实验结果如下，对于这个已经效果很好的WDCGAN，优化的作用不是很明显，第300次迭代、耗时83s时效果（图4-13左）足以媲美WDCGAN中第460次迭代、104s时的效果（图4-13右）。



图4-13 优化后WDCGAN（左）和原WDCGAN（右）对比

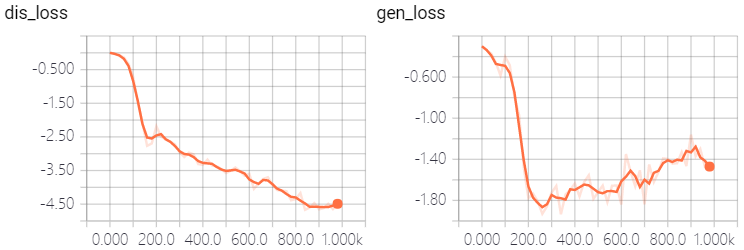


图4-14 优化后的WDCGAN的损失函数

从图4-14的损失函数变化可以看出，生成器的变化更稳定，但是由于学习率以及判别器和生成器的训练比例未做调整，生成器的损失函数后期有小幅上升，修改方案应当增加生成器的训练次数。

上述的实验均是不引入标签控制对抗生成网络生成方向的，引入标签之后效果效果更好。引入标签的方式是将标签进行独热码编码，再拼接到网络的神经元中。例如对全连接的输出，使用concat函数将标签拼接到输出上，作为下一层的输入。

dis\_h1 = tf.concat([dis\_h1, label], 1)

以下分别是带标签的DCGAN\_label、增加Wasserstein优化的WDCGAN\_label、增加本人优化方案的WDCGAN\_label\_update生成的图像。



图4-15 DCGAN\_label(左)、WDCGAN\_label(中)、WDCGAN\_label(右)效果示意

在达到相似效果时，迭代次数和用时分别为：780次187秒、580次137秒、280次78秒。从最后的结果来看，后者是优于前者的。

重申一遍优化的方法：在Wasserstein GAN的基础上，将生成器最后一层的sigmoid激活改为tanh激活，增加一层卷积层，最后仍用sigmoid激活作为输出。只对判别器的全连接的权重做clip，对生成器的全连接层和反卷积层（即能起到扩展数据的网络层）的权重做clip。前期判别器的训练次数较多，等判别器的损失函数开始稳步下降，意味着拟合结果逼近Wasserstein距离，此时增加生成器的训练次数，减少判别器的训练次数。

* 1. 对抗生成网络在人脸生成上的实验

对于验证码的数据集，每一类的字符有其自己的特征，所有的数据共同的特征并不明显，也就会出现4.3中两层全连接GAN中出现的结果，找出了所有数据的共性。

一组共性明显的数据集为人脸数据集，实验一共使用了两个数据集，UMass的LFW人脸数据集和CelebA数据集，前者数量较少并且人脸在图片中所占的比例较少，后者数据量更多并且共性更明显。下面将以CelebA数据集为例描述实验，图像转为128\*128的图像，并转为tensorflow专用数据格式，便于从磁盘中并发读取，不占用大量内存。

为了简化网络的构建，本人对tensorflow中的部分函数做了二次封装，现做如下说明：

**def** fully\_connected(value, output\_shape, name=**"fully\_connected"**)

全连接层，输入分别为：输入数据、输出的一维向量长度、网络层名称，返回值为全连接之后的输出数据

**def** lrelu(x, leak=0.2, name=**"lrelu"**)

leaky relu激活层，x为输入数据，leak为负数放缩比例，name为网络层名称，返回值为激活后数据

**def** relu(value, name=**'relu'**)

relu激活层，value为输入数据，name为网络层名称，返回值为激活后数据

**def** deconv2d(value, output\_shape, k\_h=5, k\_w=5, strides=(1, 2, 2, 1), name=**'deconv2d'**)

二维反卷积层，输入为：输入数据、输出数据规模、卷积核垂直大小、卷积核水平大小、卷积步长、网络层名称，返回值为反卷积后数据

**def** conv2d(value, output\_dim, k\_h=5, k\_w=5, strides=(1, 2, 2, 1), name=**'conv2d'**)

二维卷积层，输入为：输入数据、卷积核个数（输出数据最后一维大小）、卷积核垂直大小、卷积核水平大小、卷积步长、网络层名称，返回值为卷积后的数据

**def** batch\_norm(value, is\_train=**True**, name=**'batch\_norm'**)

批归一化层，输入为：输入数据、是否是训练过程、网络层名称，输出为归一化后结果

1. DCGAN

网络结构：

判别器：

1. 对输入数据进行卷积核大小为5、输出为32、步长为2的卷积、relu激活
2. 对上一层数据进行卷积核大小为5、输出为64、步长为2的卷积、批归一化、relu激活，输出64\*64\*64的数据
3. 对上一层数据进行卷积核大小为5、输出为128、步长为2的卷积、批归一化、relu激活，输出32\*32\*128的数据
4. 对上一层数据进行卷积核大小为5、输出为256、步长为2的卷积、批归一化、relu激活，输出16\*16\*256的数据
5. 对上一层数据进行重排列，每一张图像对应的数据转为长65536的一维向量，全连接到1个神经元，最后对输出做sigmoid激活

生成器：

1. 将噪声向量全连接为长16384的一维向量，做relu激活，重排列为8\*8\*256的三维向量
2. 反卷积成16\*16\*128的三维向量，做批归一化、relu激活
3. 反卷积成32\*32\*64的三维向量，做批归一化、relu激活
4. 反卷积成64\*64\*32的三维向量，做批归一化、relu激活
5. 反卷积成128\*128\*3的三维向量，sigmoid激活，作为最后的输出

损失函数与4.2中损失函数计算方式相同，不做赘述。判别器和生成器的训练比例1:2。

1. WDCGAN

在(a)中修改了损失函数和梯度更新方法，增加了对判别器的clip，在25次迭代以内，判别器与生成器的训练比例为25:2，之后比例为5:2，前期增加判别器训练次数是为了使判别器的损失函数大致满足Wasserstein距离。

1. WGAN

(a)、(b)的网络结构都较为复杂，效果虽好但是速度慢、并且图像扭曲，由于Wasserstein GAN使用简单的全连接网络也可以训练，本次实验则验证了这一说法。

判别器：图像重排列成一维向量，全连接到长1024的一维向量，做lrelu激活，再全连接到长为1的向量做输出。

生成器：噪声全连接到长1024的一维向量，做relu激活，再全连接到图像尺寸，做sigmoid激活，重排列成和图像数据一样的尺寸128\*128\*3。

1. WGAN的优化

(c)中的网络虽然不复杂了，但是有一个很严重的问题，简单的网络泛化能力不足，并且，图像噪点严重，另外一个问题就是全连接网络开销巨大，对于最后的输出是一个128\*128\*3的图像来说，若是第一层全连接输出的是长M的一维向量，那么第二层全连接层的权重将有49152M个，这种空间的开销是巨大的。反卷积和全连接一样，都具有扩充数据的功能，尽管反卷积可能带来数据的重复。基于对空间开销的考虑，网络结构又回到了(b)中。

对WDCGAN和WGAN的优化包括判别器和生成器的clip方式改变，增加一层卷积层。

首先是WDCGAN的实验结果：

然后是WGAN增加优化后的结果：

1. Improved Training of Wasserstein GANs 所做改进

在五月初的时候，浏览到了Wasserstein GAN的作者Martin Arjovsky对Wasserstein GAN的改进。作者意识到了Wasserstein GAN存在训练困难、收敛速度慢等问题，认为关键在于原设计中Lipschitz限制的施加方式不对，并在新论文中提出了解决方案。

在对Lipschitz施加限制时，原设计使用的是对判别器的网络参数进行clip，但是这会导致权重的分布集中在最大限制和最小限制两个极端上，判别器趋紧一个简单的映射，类似于二值网络，没有充分利用网络的泛化能力。同时，不同的网络clip的值也不同，找到这个合适的值是一个很耗时间的事情。Clip的值如果限制的稍小，每经过一层网络，梯度就会变小一点，多层之后就会指数衰减，反之梯度会指数爆炸。这也就是之前对已有的较好的网络施加Wasserstein GAN的技术效果并不明显的原因，网络层次深，clip值不好确定。

作者在新论文中提出gradient penalty方法，相应的网络模型简称为WGAN-GP。由于时间原因，本人在WGAN-GP模型上的实验和学习并不充分，只是单纯的去做了一些实验。

1. Wasserstein GAN的关键技术
2. Wasserstein GAN的关键技术