ML–第一次实战总结

ML&DL成长之路

目录

数据预处理

版本号	作者	时间	改动内容
0.1	Lart	2018年10月17日	创建文档
0.2	Lart	2018年10月18日10:55:22	调整结构
0.3	Lart	2018年10月18日11:26:14	补充保存数据的内容

数据预处理

数据归一化

虽然这里有一系列可行的方法,但是这一步通常是根据数据的具体情况而明确选择的.这一方法对诸如自然图像这类数据是有效的,但对非平稳的数据则不然.特征归一化常用的方法包含如下几种:

- 简单缩放

  通过对数据的每一个维度的值进行重新调节(这些维度可能是相互独立的),使得最终的数据向量 **落在[0,1]或[-1,1] 的区间内**(根据数据情况而定).这对后续的处理十分重要,因为很多 *默认*参数(如 PCA-白化中的 epsilon)都假定数据已被缩放到合理区间.

  例子 在处理自然图像时,我们获得的像素值在 [0,255] 区间中,常用的处理是将这些像素值除以 255,使它们缩放到 [0,1] 中.

- 逐样本均值消减(也称为移除直流分量)

  前提: 数据要求是平稳的.
  对于每个图像样本,减去各自各个通道上的整幅图像的均值.
  因为具体来说,是因为不同色彩通道中的像素并不都存在平稳特性.

例子 在实验中,在ZCA whitening前进行数据预处理时,每列代表一个样本,但为什么是对patches每一维度0均值化?
(具体做法是：首先计算每一个维度上数据的均值,使用全体数据计算,之后再每一个维度上都减去该均值)
而以前的实验都是对每个样本整体进行0均值化(即：逐样本均值消减)?

i. 因为以前是灰度图,现在是RGB彩色图像,如果现在对每列平均就是对三个通道求平均,这肯定不行.因为不同色彩通道中的像素并不都存在平稳特性,而要进行逐样本均值消减(即：单独每个样本0均值化)有一个必须满足的前提：该数据是平稳的.
ii. 因为以前是自然图像,自然图像中像素之间的统计特性都一样,有一定的相关性,而现在是人工分割的图像块,没有这种特性.

- 特征标准化(使数据集中所有特征都具有 **零均值和单位方差**)

  独立地使得数据的每一个维度具有零均值和单位方差.
  这是归一化中最常见的方法并被广泛地使用.

  特征标准化的具体做法是：

  i. 首先**计算每一个维度上数据的均值**(使用全体数据计算)
  ii. 之后在**每一个维度上都减去该均值**
  iii. 下一步便是在数据的**每一维度上除以该维度上数据的标准差**.

  例子
  - 在使用支持向量机(SVM)时,特征标准化常被建议用作预处理的一部分.
  - 处理音频数据时,常用 Mel 倒频系数 MFCCs 来表征数据.然而MFCC特征的第一个分量(表示直流分量)数值太大,常常会掩盖其他分量.这种情况下,为了平衡各个分量的影响,通常对特征的每个分量独立地使用标准化处理.

白化(协方差矩阵为单位矩阵)

> https://blog.csdn.net/danieljianfeng/article/details/42147109

白化可以降低输入的冗余性,降低特征之间的相关性,使得所有的特征具有相同的方差.

在做完简单的归一化后,白化通常会被用来作为接下来的预处理步骤,它会使我们的算法工作得更好.实际上许多深度学习算法都依赖于白化来获得好的特征.

在进行白化时,首先使 **特征零均值化**是很有必要的.

白化处理分PCA白化和ZCA白化.

- PCA白化 先PCA进行**基转换**,降低数据的相关性,再对每个输入特征进行**缩放**(除以各自的特征值的开方),以获得单位方差.此时的协方差矩阵为单位矩阵I.
- ZCA白化 ZCA白化只是在**PCA白化**的基础上做了一个**逆映射操作**,使数据转换为原始基下,使得白化之后的数据更加的接近原始数据.

ZCA白化首先通过PCA去除了各个特征之间的相关性,然后使得输入特征具有相同方差,此时得到PCA白化后的处理结果.然后再把数据旋转回去,得到ZCA白化的处理结果.

二者的差异:

1. PCA白化保证数据各维度的方差为1,ZCA白化保证数据各维度的 **方差相同**
2. PCA白化可以用于降维也可以去相关性,而ZCA白化主要用于去相关性,且尽量使白化后的数据**接近原始输入数据**.

```
# 假设输入数据矩阵X的尺寸为[N x D]
X -= np.mean(X, axis = 0) # 对数据进行零中心化(重要)
cov = np.dot(X.T, X) / X.shape[0] # 得到数据的协方差矩阵
U,S,V = np.linalg.svd(cov)
Xrot = np.dot(X, U) # 对数据去相关性

# 对数据进行白化操作 除以特征值->PCA白化
Xwhite = Xrot / np.sqrt(S + 1e-5)

# 转换为原始基->ZCA白化
Xzcawhite = np.dot(Xwhite, U.T)
```

注意 对于大图像,采用基于 PCA/ZCA 的白化方法是不切实际的,因为协方差矩阵太大.在这些情况下我们退而使用 **1/f 白化方法**.

接下来在 PCA/ZCA 白化中我们需要选择合适的 `epsilon` (回忆一下,这是规则化项,*对数据有低通滤波作用*) . 选取合适的 `epsilon` 值对特征学习起着很大作用,这里可以看ML预处理.md文档 PCA/ZCA白化一节.

实际建议

> 在卷积神经网络中并不会采用这些PCA以及白化变换.然而对数据进行零中心化操作还是非常重要的,对每个像素进行归一化也很常见.

在这一部分中,我们将介绍几种在一些数据集上有良好表现的预处理标准流程.

- 自然灰度图像

均值消减->PCS/ZCA白化

灰度图像具有平稳特性,通常在第一步对每个数据样本分别做均值消减(即减去直流分量),然后采用 PCA/ZCA 白化处理.

其中的 `epsilon` 要**足够大以达到低通滤波**的效果.

- 彩色图像

分通道均值消减->PCA/ZCA白化

对于彩色图像,色彩通道间并不存在平稳特性.通常需要分通道均值消减,并且对数据进行特征缩放(使像素值位于 [0,1] 区间).

使用足够大的 `epsilon` 来做 PCA/ZCA.

- 音频 (MFCC/频谱图)

特征标准化->PCA/ZCA 白化

对于音频数据 (MFCC 和频谱图),每一维度的取值范围(方差)不同.

即使得数据的每一维度均值为0、方差为1,然后进行PCA/ZCA白化(使用合适的 `epsilon`).

- MNIST 手写数字

简单缩放/逐样本均值消减(->PCA/ZCA 白化)

MNIST 数据集的像素值在 [0,255] 区间中.我们首先将其缩放到 [0,1] 区间.

实际上,进行逐样本均值消去也有助于特征学习.

注：也可选择以对 MNIST 进行 PCA/ZCA 白化,但这在实践中不常用.

---

one-hot独热编码(离散数据)

独热编码即 One-Hot 编码,又称一位有效编码,其方法是**使用N位状态寄存器来对N个状态进行编码**,每个状态都由他独立的寄存器位,并且**在任意时候,其中只有一位有效**.

可以这样理解,对于每一个特征,如果它有m个可能值,那么经过独热编码后,**就变成了m个二元特征**.并且,这些特征互斥,每次只有一个激活.因此,数据会变成稀疏的.

> **稀疏数据相比密集数据而言,对于计算机加载处理也更为有优势.**

这样做的好处主要有:

1. 解决了分类器不好处理属性数据的问题
2. 在一定程度上也起到了扩充特征的作用

实际建议

对于标签数据可以转化为独热编码形式,这对于网络的分类输出而言,是更为实际的表达形式.

```
from keras.utils import np_utils
...
Y_train = np_utils.to_categorical(y_train, num_classes)
```

而对于真实的类别序号可以通过使用`NumPy argmax()` 函数来找到(这里的查找,也不一定对one-hot后的数据进行查找,对于得分数据亦可以,对于softmax计算后的也可以,因为是增函数).

数据增强(数据量有限)

> https://blog.csdn.net/mzpmzk/article/details/80039481
> https://blog.csdn.net/u010555688/article/details/60757932?utm_source=blogxgwz1

深层神经网络一般都需要大量的训练数据才能获得比较理想的结果。在**数据量有限的情况**下，可以通过数据增强（Data Augmentation）来增加训练样本的多样性， 提高模型鲁棒性，避免过拟合。

图片数据增强通常只是针对训练数据，对于测试数据则用得较少。测试数据常用的是：做 5 次随机剪裁，然后将 5 张图片的预测结果做均值。

常用手段:

- 翻转flip
- 旋转rotation
- 平移shift
- 缩放resize/尺度变换Random Scale
- 随机裁剪或补零Random Crop or Pad
- 色彩抖动Color jittering:HSV颜色空间随机改变图像原有的饱和度和明度（即，改变 S 和 V 通道的值）或对色调(Hue)进行小范围微调。
- 对比度变换contrast:在图像的HSV颜色空间，改变饱和度S和V亮度分量，保持色调H不变. 对每个像素的S和V分量进行指数运算(指数因子在0.25到4之间)，增加光照变化;
- PCA抖动PCA Jittering：首先按照RGB三个颜色通道计算均值和标准差，再在整个训练集上计算协方差矩阵，进行特征分解，得到特征向量和特征值，用来做PCA Jittering；
- 加噪声（Noise）
- 特殊的数据增强方法：
  - Fancy PCA（Alexnet）& 监督式数据扩充（海康）
  - 使用生成对抗网络（GAN） 生成模拟图像

保存数据

将训练集数据/标签,以及测试数据(可能有的标签),都事先处理后，并转化好 `.npy` 或者其他的整合的数据文件，便于后期的分布处理,而且也可以减少训练过程在读入数据上的处理时间.

> https://gist.github.com/lartpang/e2343fb06f25b36a8af03e68923dea06

补充

1. 处理离散型特征和连续型特征并存的情况时,必须进行特征的归一化,每个特征都单独进行归一化.对于连续型特征归一化的常用方法是放缩后标准化,对于离散的特征基本就是按照one-hot编码,该离散特征有多少取值,就用多少维来表示该特征.
2. 基于树的方法是不需要进行特征的归一化,例如随机森林, bagging 和 boosting等.基于参数的模型或基于距离的模型,都是要进行特征的归一化.
3. 进行预处理很重要的一点是：任何预处理策略（比如数据均值）都 **只能在训练集数据上进行计算**,算法训练完毕后再应用到验证集或者测试集上。

   例子 如果先计算整个数据集图像的平均值然后每张图片都减去平均值，最后将整个数据集分成训练/验证/测试集，那么这个做法是错误的。**应该怎么做呢？应该先分成训练/验证/测试集，只是从训练集中求图片平均值，然后各个集（训练/验证/测试集）中的图像再减去这个平均值。**

4. 对于神经网络模型而言,图片需要进行预处理的原因：
   i. 适应神经网络结构：网络结构可以接收的数据格式/类型是固定的，因此在训练过程之前，需要将训练样本预处理成为可以被神经网络读取的格式/类型；
   ii. 对训练样本进行 **提纯**：训练样本中可能存在“不好的数据”，通过预处理手段，可以将这部分数据剔除掉，或者消除掉它的影响,训练图片是越“纯”越好，也就是让网络只关注你想让它记住的；
   iii. 进行数据 **增强**：对训练样本进行预处理，可以增加数据的多样性。例如通过旋转、镜像、裁切等手段，将图片的空间多样性呈现出来，据此训练出来的模型也将具有更好的鲁棒性,可以提升泛化能力.

   数据增强的目的是增加样本数量，这在数据很少的情况下是非常有效的，Caffe自己的数据输入层自带裁剪、镜像等功能，一般做法是把训练数据放大到输入尺寸的1.1倍左右，然后随机裁剪到输入尺寸，这个效果非常好。

   不好的数据增强容易带来模型过拟合，因为我们只是增加了数据并非泛化了数据分布；
   iv. 数据归一化：预处理可以将不同规格的数据转换成相同规格的训练数据，最典型的的例子就是图片的尺寸归一化。

   上面提到的减均值除以方差的方式，在Batch Normalization(BN)出现之前是很有必要的，因为这样拉到均值附近，学习的时候更加容易，毕竟激活函数是以均值为中心的，学习到这个位置才能将不同的类分开。

   但是**BN出现之后，这个操作就完全没必要**了，因为每次卷积后都有BN操作，BN就是把数据拉到0均值1方差的分布，而且这个均值和方差是动态统计的，不是只在原始输入上统计，因此更加准确。即采用BN后该预处理无必要。
   v. 压缩数据体积：预处理还可以减小训练数据的尺寸。例如原始数据是FHD尺寸的，若压缩至QQVGA则会大幅缩减数据体积;
   vi. 将数据处理为某种组织形式的存储格式，比如tfrecord，这是为了加速训练，这一步可不得小觑，大的模型训练是非常耗时的，提前做预处理可节省不少时间还可降低对设备内存要求等；
   vii. 如果训练数据存在极度不均衡情况，那么需要做图像预处理，欠采样或过采样或补充，目的是不让模型结果偏向于数据大的类别；

构建网络

版本号	作者	时间	改动内容
0.1	Lart	2018年10月17日	创建文档
0.2	Lart	2018年10月18日10:59:28	调整部分内容

前言

以经典的AlexNet网络为例,介绍构建网络的细节.

先给出AlexNet的一些参数:

- **卷积层：5层**
- **全连接层：3层**
- **深度：8层**
- **参数个数：60M**
- **神经元个数：650k**
- **分类数目：1000类**

卷积层

卷积层在计算的时候看上去像是在做卷积运算,实际上是一种等价的理解方式.

对于卷积层的理解有两种看待的角度:

1. 每一个卷积核(可移动的对整个图像进行滑动内积),都对应一个卷积层的深度切片,每个卷积层都有多个深度切片,也就是对应多个卷积核,多次不同的卷积计算.最终得到的特征图的输出的深度是和卷积层的切片数目(深度)一致.
2. 每一个卷积层切片对应的运算,不再看做一个单独的核在滑动卷积,而是看作具有多个神经元的一个网络层,每个神经元都有自己的局部感受野,只计算自己的那一部分.所有这一切片上的神经元的运算结果的结合就是得到的一层特征图.

由于卷积层的 **参数共享** 设定,导致两种理解是等价的. 而实际运算中,是以第一种理解更为直观,所以多用第一种理解方式来解释问题.

在计算卷积核的卷积时,每一次卷积运算都会有一个偏置项,这一点到时更贴合第2点的理解,每一个神经元的输入连接都有一个偏置项.

注意

- 关于感受野: 在卷积神经网络中，感受野的定义是 卷积神经网络每一层输出的特征图（feature map）上的像素点在原始图像上映射的区域大小。
- 关于参数共享: 如果一个特征在计算某个空间位置(x, y)的时候有用，那么它在计算另一个不同位置(x_2, y_2)的时候也有用。基于这个假设，可以显著地减少参数数量。
- 关于滤波器深度: 每个滤波器在空间上（宽度和高度）都比较小，但是**深度和输入数据一致**。

  > 有一种设定是不同深度,这会导致更为复杂一些的情况.
  > 沿着深度方向排列、感受野相同的神经元集合称为**深度列（depth column）**，也有人使用**纤维（fibre）**来称呼它们。
- 关于卷积核移动的步长: 一般是1,但是有些时候会有所变化.

  > 最近一个研究（Fisher Yu和Vladlen Koltun的论文）给卷积层引入了一个新的叫*扩张（dilation）*的超参数。到目前为止，我们只讨论了卷积层滤波器是连续的情况。但是，**让滤波器中元素之间有间隙也是可以的**，这就叫做扩张。换句话说，操作中存在1的间隙。在某些设置中，扩张卷积与正常卷积结合起来非常有用，因为在很少的层数内更快地汇集输入图片的大尺度特征。
- 关于卷积操作的反向传播: （同时对于数据和权重）还是一个卷积（但是是和空间上翻转的滤波器）。

每个卷积核对应的输出数据体在空间上的尺寸可以通过**输入数据体尺寸（W）**，卷积层中神经元的**感受野尺寸（F）**，**步长（S）**和**零填充的数量（P）**的函数来计算。（这里假设输入数组的空间形状是正方形，即高度和宽度相等）

输出数据体的空间尺寸为 $\frac{W-F+2P}{S}+1$

应该使用**小尺寸滤波器**（比如3x3或最多5x5），使用步长$S=1$。

还有一点非常重要，就是对输入数据进行零填充，这样卷积层就不会改变输入数据在空间维度上的尺寸。比如，当$F=3$，那就使用$P=1$来保持输入尺寸。当$F=5, P=2$，一般对于任意F，当$P=(F-1)/2$的时候能保持输入尺寸。如果必须使用更大的滤波器尺寸（比如7x7之类），通常只用在第一个面对原始图像的卷积层上。

直观说来，最好选择带有小滤波器的卷积层组合，而不是用一个带有大的滤波器的卷积层。
前者可以表达出输入数据中更多个强力特征，使用的参数也更少。
唯一的不足是，在进行**反向传播时，中间的卷积层可能会导致占用更多的内存。**

汇聚层

通常，在**连续的卷积层之间会周期性地插入**一个汇聚层。

它的作用是**逐渐降低数据体的空间尺寸，这样的话就能减少网络中参数的数量，使得计算资源耗费变少，也能有效控制过拟合。**

> 在大型网络中，一般担心的是过拟合的问题，而欠拟合关心的少些。

汇聚层使用`MAX操作`，**对输入数据体的每一个深度切片独立进行操作**，改变它的空间尺寸。

在实践中，最大汇聚层通常只有两种形式：一种是$F=3,S=2$也叫**重叠汇聚（overlapping pooling）**，另一个更常用的是$F=2,S=2$对更大感受野进行汇聚需要的汇聚尺寸也更大，而且往往对网络有破坏性。

除了最大汇聚，汇聚单元还可以使用其他的函数，比如*平均*汇聚*（average pooling）*或*L-2范式*汇聚*（L2-norm pooling）*。平均汇聚历史上比较常用，但是现在已经很少使用了。因为实践证明，**最大汇聚的效果比平均汇聚要好。**

负责对输入数据的空间维度进行降采样。**最常用的设置是用2x2感受野（即$F=2$）的最大值汇聚**，步长为2（$S=2$）。

注意

- 最大汇聚的反向传播就是对于`max`操作的一个求导问题,在向前传播经过汇聚层的时候，通常会把池中最大元素的索引记录下来（有时这个也叫作**道岔（switches）**），这样在反向传播的时候梯度的路由就很高效。
- 很多人不喜欢汇聚操作，认为可以不使用它。比如在Striving for Simplicity: The All Convolutional Net一文中，提出使用一种只有重复的卷积层组成的结构，抛弃汇聚层。通过在卷积层中使用更大的步长来降低数据体的尺寸。有发现认为，在训练一个良好的生成模型时，弃用汇聚层也是很重要的。比如变化自编码器（VAEs：variational autoencoders）和生成性对抗网络

（GANs：generative adversarial networks）。现在看起来，未来的卷积网络结构中，可能会很少使用甚至不使用汇聚层。

全连接层

在全连接层中，神经元对于前一层中的所有激活数据是全部连接的，这个常规神经网络中一样。它们的激活可以先用矩阵乘法，再加上偏差。

注意

- 对于任一个卷积层，都存在一个能实现和它一样的前向传播函数的全连接层。权重矩阵是一个巨大的矩阵，除了某些特定块（这是因为有局部连接），**其余部分都是零**。而在其中大部分块中，元素都是相等的（因为参数共享）。
- 任何全连接层都可以被转化为卷积层。比如，一个$K = 4096$的全连接层，输入数据体的尺寸是$7 \times 7 \times 512$，这个全连接层可以被等效地看做一个$F = 7, P = 0, S = 1, K = 4096$的卷积层。换句话说，就是将**滤波器的尺寸设置为和输入数据体的尺寸一致了**。因为只有一个单独的深度列覆盖并滑过输入数据体，所以输出将变成$1 \times 1 \times 4096$，这个结果就和使用初始的那个全连接层一样了。

在两种变换中，将全连接层转化为卷积层在实际运用中更加有用。

补充

卷积神经网络最常见的形式就是将一些卷积层和ReLU层放在一起，其后紧跟汇聚层，然后重复如此直到图像在空间上被缩小到一个足够小的尺寸，在某个地方过渡成全连接层也较为常见。最后的全连接层得到输出，比如分类评分等。换句话说，最常见的卷积神经网络结构如下：

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

其中*指的是重复次数，POOL?指的是一个可选的汇聚层。其中N >=0,通常N<=3,M>=0,K>=0,通常K<3。

例如，下面是一些常见的网络结构规律：

- INPUT -> FC，实现一个线性分类器，此处N = M = K = 0。
- INPUT -> CONV -> RELU -> FC

- **INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC**。此处在每个汇聚层之间有一个卷积层。
- *INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]2 -> FC*。此处每个汇聚层前有两个卷积层，这个思路适用于更大更深的网络，因为在执行*具有破坏性的汇聚操作*前，多重的卷积层可以从输入数据中学习到更多的复杂特征。

激活函数

> **激活函数的理论作用是使神经网络成为一个普适近似器。**

实际建议

在同一个网络中混合使用不同类型的神经元是非常少见的，虽然没有什么根本性问题来禁止这样做。

用ReLU非线性函数。注意设置好学习率，或许可以监控你的网络中死亡的神经元占的比例。
如果单元死亡问题困扰你，就试试Leaky ReLU或者Maxout，不要再用sigmoid了。
也可以试试tanh，但是其效果应该不如ReLU或者Maxout。

权重初始化

在训练完毕后，虽然不知道网络中每个权重的最终值应该是多少，但如果数据经过了恰当的归一化的话，就可以假设所有权重数值中大约一半为正数，一半为负数。
这样，一个听起来蛮合理的想法就是把这些权重的初始值都设为0吧，因为在期望上来说0是最合理的猜测。这个做法错误的！
因为如果网络中的每个神经元都计算出同样的输出，然后它们就会在反向传播中计算出同样的梯度，从而进行同样的参数更新。换句话说，如果权重被初始化为同样的值，神经元之间就失去了不对称性的源头。

1. 小随机数初始化: 权重初始值要非常接近0又不能等于0。解决方法就是将权重初始化为很小的数值，以此来*打破对称性*。其思路是：如果神经元刚开始的时候是随机且不相等的，那么它们将计算出不同的更新，并将自身变成整个网络的不同部分。高斯分布/均匀分布一般都可以,差别不大.
   注意 并不是小数值一定会得到好的结果。例如，一个神经网络的层中的权重值很小，那么在反向传播的时候就会计算出非常小的梯度（因为梯度与权重值是成比例的）。这就会很大程度上减小反向传播中的“梯度信号”，在深度网络中，就会出现问题。
2. 偏置参数初始化: 通常还是使用0来初始化偏置参数。
3. 批量归一化: 在神经网络中使用批量归一化已经变得非常常见。在实践中，使用了批量归一化的网络对于不好的初始值有更强的鲁棒性。批量归一化可以理解为在网络的每一层之前都做预处理，只是这种操作以另一种方式与网络集成在了一起.

实际建议 当前的推荐是使用ReLU激活函数，并且使用`w = np.random.randn(n) * sqrt(2.0/n)`来进行权重初始化.网络中可以使用批量归一化来动态处理权重.

正则化

主要目的是为了防止网络过拟合.

主要分为两类.

一类是作为损失函数的正则化损失部分，常见的是L范数.

- L2正则化: **L2正则化可以直观理解为它对于大数值的权重向量进行严厉惩罚，倾向于更加分散的权重向量。**可以通过惩罚目标函数中所有参数的平方将其实现。即对于网络中的每个权重w，向目标函数中增加一个$\frac{1}{2}\lambda w^2$，其中λ是正则化强度。前面这个$\frac{1}{2}$很常见，是因为加上$\frac{1}{2}$后，该式子关于w梯度就是λw而不是$2\lambda w$了。
- L1正则化: 是另一个相对常用的正则化方法。对于每个w我们都向目标函数增加一个$\lambda|w|$。L1正则化有一个有趣的性质，它会让权重向量在最优化的过程中变得稀疏（即非常接近0）。

L1正则化的神经元最后使用的是它们最重要的输入数据的稀疏子集，同时对于噪音输入则几乎是不变的了。
L2正则化中的权重向量大多是分散的小数字。

实际建议 **如果不是特别关注某些明确的特征选择，一般说来L2正则化都会比L1正则化效果好。**

L1和L2正则化也可以进行组合：$\lambda_1|w| + \lambda_2 w^2$，这也被称作Elastic net regularizaton。

另外一类是网络传递中的操作，如随机失活.

- 使用dropout可以使得部分节点失活，可以起到简化神经网络结构的作用，从而起到正则化的作用。
- 因为dropout是使得神经网络的节点随机失活，这样会让神经网络在训练的时候不会使得某一个节点权重过大。因为该节点输入的特征可能会被清除，所以神经网络的节点不能依赖任何输入的特征。
- dropout最终会产生收缩权重的平方范数的效果，来压缩权重，达到类似于L2正则化的效果。

1. 普通随机失活

```
""" 普通版随机失活: 不推荐实现 """
p = 0.5 # 激活神经元的概率. p值更高 = 随机失活更弱，越容易保留神经元

def train_step(X):
```

```
    """ X中是输入数据 """

    # 3层neural network的前向传播
    H1 = np.maximum(0, np.dot(W1, X) + b1) # 第一层输出
    U1 = np.random.rand(*H1.shape) < p # 第一个随机失活遮罩，要失活的位置就是0，也就是值大于p的要失活
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2) # 第二层输出
    U2 = np.random.rand(*H2.shape) < p # 第二个随机失活遮罩
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3 # 第三层输出 = 第三层输入
    # 反向传播:计算梯度... (略)
    # 进行参数更新... (略)

def predict(X):
    # 前向传播时模型集成
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # 注意：激活数据要乘以p
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # 注意：激活数据要乘以p
    out = np.dot(W3, H2) + b3
```

注意 **在训练的时候,要随机失活,但是在预测的时候,**不进行随机失活,但是对于两个隐层的输出都要乘以p,调整其数值范围.

2. 反向随机失活

```
"""
反向随机失活： 推荐实现方式.
在训练的时候drop和调整数值范围，测试时不做任何事.
"""
p = 0.5 # 激活神经元的概率. p值更高 = 随机失活更弱

def train_step(X):
    # 3层neural network的前向传播
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # 第一个随机失活遮罩. 注意/p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # 第二个随机失活遮罩. 注意/p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

  # 反向传播:计算梯度... (略)
```

```
    # 进行参数更新... （略）

def predict(X):
    # 前向传播时模型集成
    H1 = np.maximum(0, np.dot(W1, X) + b1) # 不用数值范围调整了
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

注意 **在训练的时候,要随机失活,并除以对应的概率,但是在预测的时候,**不进行随机失活,对于两个隐层的输出不再需要乘以p.

实际建议

- 通过交叉验证获得一个全局使用的L2正则化强度是比较常见的。
- 在使用L2正则化的同时在所有层后面使用随机失活也很常见。

p值一般默认设为0.5，也可能在验证集上调参。

损失函数

我们已经讨论过损失函数的正则化损失部分，它可以看做是对模型复杂程度的某种惩罚。

损失函数的第二个部分是*数据损失*，它是一个有监督学习问题，用于衡量分类算法的预测结果（即分类评分）和真实标签结果之间的一致性。数据损失是对所有样本的数据损失求平均。

一般的分类问题

在该问题中，假设有一个装满样本的数据集，每个样本都有一个唯一的正确标签（是固定分类标签之一）。

- 一个最常见的损失函数就是SVM（是Weston Watkins 公式）:$L_i = \sum_{j \neq y_i} max(0, f_j - f_{y_i} + 1)$.
  有些学者的论文中指出平方折叶损失（即使用$max(0, f_j - f_{y_i} + 1)^2$）算法的结果会更好.
- 第二个常用的损失函数是Softmax分类器，它使用交叉熵损失: $L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$

类别数目巨大

当标签集非常庞大（例如字典中的所有英语单词，或者ImageNet中的22000种分类），就需要使用 *分层Softmax（**Hierarchical Softmax**）* 了（参考文献）。

分层softmax将标签分解成一个树。每个标签都表示成这个树上的一个路径，这个树的每个节点处都训练一个Softmax分类器来在左和右分枝之间做决策。树的结构对于算法的最终结果影响很大，而且一般需要具体问题具体分析。

属性（Attribute）分类

上面两个损失公式的前提，都是假设每个样本只有一个正确的标签y_i。但是如果y_i是一个二值向量，每个样本可能有，也可能没有某个属性，而且属性之间**并不相互排斥**呢?

比如在Instagram上的图片，就可以看成是被一个巨大的标签集合中的某个子集打上标签，一张图片上可能有多个标签。

在这种情况下，一个明智的方法是为每个属性创建一个独立的二分类的分类器。
一个方法: 针对每个分类的二分类器会采用下面的公式：$L_i = \sum_j max(0, 1 - y_{ij} f_j)$.
上式中，求和是对所有分类j，y_{ij}的值为1或者-1，具体根据**第i个样本是否被第j个属性打标签**而定，当该类别**被正确预测并展示**的时候，分值向量f_j为正，其余情况为负。
也就是说,第i个样本是(1)否(-1)被第j个属性打标签时,又是(+)否(-)被预测为该类别,结果为负时,就计算损失.即被打了标签,却没有预测为该类,没打标签却被预测为该类时就计算损失.

另一种方法: 对每种属性训练一个独立的逻辑回归分类器。
然后损失函数最大化这个对数似然函数，问题可以简化为：

$$L_i = \sum_j y_{ij} log(\sigma(f_j)) + (1 - y_{ij}) log(1 - \sigma(f_j))$$

回归问题

是预测实数的值的问题，比如预测房价，预测图片中某个东西的长度等。对于这种问题，通常是计算预测值和真实值之间的损失。然后用L2平方范式或L1范式度量差异。

L2范式：$L_i = ||f - y_i||_2^2$;
L1范式则是要将每个维度上的绝对值加起来：$L_i = ||f - y_i||_1 = \sum_j |f_j - (y_i)_j|$

注意 L2损失比起较为稳定的Softmax损失来，其**最优化过程要困难很多**。

- 直观而言，它需要网络具备一个特别的性质，即对于每个输入（和增量）都要输出一个确切的正确值。而在Softmax中就不是这样，**每个评分的准确值并不是那么重要：只有当它们量级适当的时候，才有意义**。
- L2损失鲁棒性不好，因为异常值可以导致很大的梯度。

所以在面对一个回归问题时，**先考虑将输出变成二值化是否真的不够用**。例如，如果对一个产品的星级进行预测，使用5个独立的分类器来对1-5星进行打分的效果一般比使用一个回归损失要好很多。分类还有一个额外优点，就是能给出关于回归的输出的分布，而不是一个简单的毫无把握的输出值。

实际建议 如果确信分类不适用，那么使用L2损失吧，但是一定要谨慎：L2非常脆弱，**在网络中使用随机失活（尤其是在L2损失层的上一层）不是好主意**。

> 当面对一个回归任务，首先考虑是不是必须这样。一般而言，尽量把你的输出变成二分类，然后对它们进行分类，从而变成一个分类问题。

结构化预测（structured prediction）

结构化损失是指**标签可以是任意的结构**，例如图表、树或者其他复杂物体的情况。通常这种情况还会**假设结构空间非常巨大，不容易进行遍历**。结构化SVM背后的基本思想就是**在正确的结构y_i和得分最高的非正确结构之间画出一个边界**。

解决这类问题，并不是像解决一个简单无限制的最优化问题那样使用梯度下降就可以了，而是需要设计一些特殊的解决方案，这样可以有效利用对于结构空间的特殊简化假设。

实际建议

1. 需要记住的是：**不应该因为害怕出现过拟合而使用小网络。相反，应该进尽可能使用大网络，然后使用正则化技巧来控制过拟合。**
2. 在构建卷积神经网络结构时，最大的瓶颈是内存瓶颈。要注意三种内存占用来源：
    i. 来自中间数据体尺寸：卷积神经网络中的每一层中都有激活数据体的原始数值，以及损失函数对它们的梯度（和激活数据体尺寸一致）。通常，大部分激活数据都是在网络中靠前的层中（比如第一个卷积层）。在训练时，这些数据需要放在内存中，因为**反向传播**的时候还会用到。但是在**测试时可以聪明点**：让网络在**测试运行时候每层都只存储当前的激活数据，然后丢弃前面层的激活数据，这样就能减少巨大的激活数据量**。
    ii. 来自参数尺寸：即整个网络的参数的数量，在反向传播时它们的梯度值，以及使用momentum、Adagrad或RMSProp等方法进行最优化时的每一步计算缓存。因此，存储参数向量的内存通常需要在参数向量的容量基础上乘以3或者更多。

iii. 卷积神经网络实现还有各种零散的内存占用，比如成批的训练数据，扩充的数据等等。

如果你的网络工作得不好，一个常用的方法是降低批尺寸（batch size），因为**绝大多数的内存都是被激活数据消耗掉**了。

训练数据

版本号	作者	时间	改动内容
0.1	Lart	2018年10月17日	创建文档
0.2	Lart	2018年10月18日10:41:53	补充关于模型保存的问题/调整部分结构

在最初始的学习训练中,多使用预训练好的模型.在其基础上进行微调(FineTune).

优化器选择

> https://keras.io/zh/optimizers/

SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

随机梯度下降优化器

包含扩展功能的支持:

- 动量（momentum）优化,
- 学习率衰减（每次参数更新后）
- Nestrov动量(NAG)优化

RMSprop

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp优化器.

建议使用优化器的默认参数 （除了学习率lr，它可以被自由调节）

这个优化器通常是训练循环神经网络RNN的不错选择。

Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam优化器.

实际建议 在实际操作中，我们**推荐Adam作为默认的算法，一般而言跑起来比RMSProp要好一点。但是也可以试试SGD+Nesterov动量**。

完整的Adam更新算法也包含了一个偏置*（bias）矫正*机制，因为**m,v**两个矩阵初始为0，在没有完全热身之前存在偏差，需要采取一些补偿措施。

补充

训练一个神经网络需要：

- 利用小批量数据对实现进行**梯度检查**，还要注意各种错误。
- 进行**合理性检查**，确认初始损失值是合理的，在小数据集上能得到100%的准确率。
- 在训练时，**跟踪损失函数值，训练集和验证集准确率**，如果愿意，还可以**跟踪更新的参数量相对于总参数量的比例**（一般在1e-3左右），然后如果是对于*卷积神经网络*，可以将第一层的**权重可视化**。
- 推荐的两个**更新方法**是SGD+Nesterov动量方法，或者Adam方法。
- 随着训练进行**学习率衰减**。比如，在固定多少个周期后让学习率减半，或者当验证集准确率下降的时候。
- 使用**随机搜索**（不要用网格搜索）来搜索最优的超参数。分阶段从粗（比较宽的超参数范围训练1-5个周期）到细（窄范围训练很多个周期）地来搜索。
- 进行**模型集成**来获得额外的性能提高。

回调函数选择

> https://keras.io/zh/callbacks/

回调函数是一个函数的合集，会在训练的阶段中所使用。你可以使用回调函数来查看训练模型的内在状态和统计。你可以传递一个列表的回调函数（作为 `callbacks` 关键字参数）到 `Sequential` 或 `Model` 类型的 `.fit()` 方法。在训练时，相应的回调函数的方法就会被在各自的阶段被调用。

模型保存–ModelCheckpoint

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False, mode='auto', period=1)
```

在每个训练期之后保存模型。

`filepath` 可以包括命名格式选项，可以由 `epoch` 的值和 `logs` 的键（由 `on_epoch_end` 参数传递）来填充。

例如：如果 `filepath` 是 `weights.{epoch:02d}-{val_loss:.2f}.hdf5`， 那么模型被保存的的文件名就会有训练轮数和验证损失。

LearningRateScheduler

```
keras.callbacks.LearningRateScheduler(schedule, verbose=0)
```

学习速率定时器。

评估标准选择

对于任何分类问题，你都希望将其设置为 `metrics = ['accuracy']`。评估标准可以是现有的标准的字符串标识符，也可以是自定义的评估标准函数。

注意 回归问题可以使用mae评估

```
# 多分类问题
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 二分类问题
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 均方误差回归问题
# 评估标可以选择 mae
model.compile(optimizer='rmsprop',
              loss='mse')

# 自定义评估标准函数
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

数据集的划分

训练集（Training Set）：帮助我们训练模型，简单的说就是通过训练集的数据让我们确定拟合曲线的参数。

验证集（Validation Set）：用来做模型选择（model selection），即做模型的最终优化及确定的，用来辅助我们的模型的构建，可选；

测试集（Test Set）： 为了测试已经训练好的模型的精确度。当然，test set这并不能保证模型的正确性，他只是说相似的数据用此模型会得出相似的结果。因为我们在训练模型的时候，参数全是根据现有训练集里的数据进行修正、拟合，有可能会出现过拟合的情况，即这个参数仅对训练集里的数据拟合比较准确，这个时候再有一个数据需要利用模型预测结果，准确率可能就会很差。

显然越高次数的多项式模型越能够适应我们的训练数据集，但是适应训练数据集并不代表着能推广至一般情况，我们应该选择一个更能适应一般情况的模型。我们需要使用交叉验证集来帮助选择模型。

模型选择的方法为：

1. 使用训练集训练出多个模型
2. 用多个模型分别对交叉验证集计算得出交叉验证误差（代价函数的值）
3. 选取代价函数值最小的模型
4. 用步骤3中选出的模型对测试集计算得出推广误差（代价函数的值）

注意 一般而言,只有两个数据集,一个是训练集,一个是测试集,验证集可以从训练集上划分出来一部分. 但实际应用中，一般只将数据集分成两类，即训练集Training set 和测试集Test set，大多数文章并不涉及验证集Validation set。

划分比例

传统的机器学习领域中，由于收集到的数据量往往不多，比较小，所以需要将收集到的数据分为三类：训练集、验证集、测试集。若有验证集，则划为6:2:2. 这样划分确实很科学，当数据量不大的时候（万级别及以下）。
也有人分为两类，就是不需要测试集。一般将训练集和测试集划为7：3.

在大数据时代的机器学习或者深度学习领域中，如果还是按照传统的数据划分方式不是十分合理.
因为测试集和验证集用于评估模型和选择模型，所需要的数据量和传统的数据量差不多，但是由于收集到的数据远远大于传统机器学习时代的数据量，所以占的比例也就要缩小。
比如我们拥有1000000，这么多的数据，训练集：验证集：测试集=98:1:1。如果是两类，也就是相同的道理。

fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None)
```

以固定数量的轮次（数据集上的迭代）训练模型。

其中的参数 **validation_split** 在 0 和 1 之间浮动。用作验证集的训练数据的比例。模型将分出一部分不会被训练的验证数据，并将在每一轮结束时评估这些验证数据的误差和任何其他模型指标。验证数据是混洗之前 `x` 和`y` 数据的最后一部分样本。

而参数 **validation_data 是一个**元组 `(x_val，y_val)` 或元组 `(x_val，y_val，val_sample_weights)`，用来评估损失，以及在每轮结束时的任何模型度量指标。模型将不会在这个数据上进行训练。这个参数会覆盖 `validation_split`。

实际建议

1. 关于训练时的代码调试,可以先不要指定太大的Epoch,这样可以快速的完成一次完整的训练,使得尽快的查看一次完整的过程的结果.观察训练流程之外的代码设计是否有误.
2. 对于使用的数据,最好在预处理后就将其保存成一个文件,比如利用 `np.save()&np.load()` 来使用 `.npy` 文件,使用numpy数组.分成不同的 `.py` 书写,可以减少每次调试具体内容的时间.
   若是 `jupyter notebook` ,则建议分开cell实现.(在实现小规模运算的时候,自身电脑可以使用它,但是稍大一点的,还是不要使用它了,容易卡死)
3. 关于模型的训练/预测阶段可以分离,尽可能的分离功能单元.
   训练阶段保存模型,可以利用上面的 `ModelCheckpoint` 来实现;
   测试阶段使用模型,搭建相同的网络,使用 `load_weights` 即可.
4. 在多GPU机器上跑代码测试的时候,建议指定GPU运行.Keras 默认是占满所有显存，这时候其他的进程就无法使用 GPU 资源了(即使keras 实际上用不完).

   ```
   CUDA_VISIBLE_DEVICES=0 python test.py
   ```

   关于多GPU并行处理,还需研究.一般可以只是用单卡.若是提示显存不足,可以适当降低批次.

预测/导出/显示

项目名称：第一次机器学习实战总结

版本号	作者	时间	改动内容
0.1	Lart	2018年10月18日10:16:29	创建文档

预测

核心代码

predict

```
predict(self, x, batch_size=None, verbose=0, steps=None)
```

为输入样本生成输出预测。

计算逐批次进行。

对于已经训练好的网络而言,获得其已经保存好的最好权重,在相同的架构上载入模型即可使用 `predict` 来进行预测数据了.要注意路径

```
# 构建不带分类器的预训练模型
inputshape = (224, 224, 3)
base_model = VGG16(input_shape=inputshape,
                           weights='imagenet',
                           include_top=True)
old_layer = base_model.get_layer('fc2').output
predictions = Dense(4, activation='linear')(old_layer)

# 构建我们需要训练的完整模型
model = Model(inputs=base_model.input, outputs=predictions)
model.load_weights('./model_data/weights_best_vgg16.hdf5', by_name=True)
```

```
# 导入数据进行测试
box_path = './test/box/'
img_path = './test/img/'

img_names = os.listdir(img_path)

images = []
inputs = []
for img_name in img_names:
    img = image.load_img(img_path + img_name, target_size=(224, 224))
    img = image.img_to_array(img)
    images.append(imread(img_path + img_name))
    inputs.append(img.copy())

inputs = preprocess_input(np.array(inputs))

# 预测
print('总的输入形状:', inputs.shape)
results = model.predict(inputs, batch_size=1, verbose=1)
print('预测结果形状:', results.shape)
```

导出

关于导出数据,主要内容是预测的测试集的结果的导出.根据任务不同,可能会有不同的处理手段,这里以类似预目标检测任务的一个输出的导出为例.

注意 这里的导出的格式很关键.因为有些任务对你的运行结果的检测,可能是针对固定的格式而言的.应该仔细查看任务要求.甚至可能会给你的 `example` 示例文件.这里的核心关键是 看清要求!

例子

下面代码实现了导出到xml文件的一个流程.

```
from scipy.misc import imread
from xml.etree import ElementTree as ET

results = model.predict(inputs, batch_size=1, verbose=1)
print('预测结果形状:', results.shape)
```

```
# 保存结果到xml文件
for i, img_name in enumerate(img_names):
    img_read = imread(img_path + img_name)
    # Parse the outputs.
    det_xmin = results[i][0]
    det_ymin = results[i][1]
    det_xmax = results[i][2]
    det_ymax = results[i][3]

    # 指定根节点/子节点
    prediction = ET.Element("prediction")
    bndbox = ET.SubElement(prediction, "bndbox")
    xmin = ET.SubElement(bndbox, "xmin")
    ymin = ET.SubElement(bndbox, "ymin")
    xmax = ET.SubElement(bndbox, "xmax")
    ymax = ET.SubElement(bndbox, "ymax")
    conf = ET.SubElement(bndbox, "conf")

    if det_xmax > img_read.shape[1]:
        det_xmax = img_read.shape[1] - 1
    if det_ymax > img_read.shape[0]:
        det_ymax = img_read.shape[0] - 1

    xmin.text = str(int(round(det_xmin)))
    ymin.text = str(int(round(det_ymin)))
    xmax.text = str(int(round(det_xmax)))
    ymax.text = str(int(round(det_ymax)))
    conf.text = '1.0000'

    # 保存结构树
    tree = ET.ElementTree(prediction)
    tree.write(box_path + img_name[:-3] + 'xml')
```

显示

这个涉及到了对于预测结果的一个可视化的问题.具体的代码不是很重要,关键是要明确如何更为直观清楚的表达出你的预测的准确与否.

有时,对于无法明确查看的测试集(没有没有测试标签),直接将预测结果绘制出来,可能会更容易理解和判别.

例子 对于目标检测问题,图像上的框远比终端里的文字清楚的多.