目录

1. 引言

生产实习是自动化专业教学计划中重要的实践性教学环节，是对学生进行专业基本训练，培养实践动手能力和向实践学习，理论联系实际的重要环节。通过实习，能够培养学生学习及巩固所学专业知识，培养专业兴趣和专业技能，同时提高分析和解决实际问题的能力，深刻理解专业的知识体系结构，培养学生在现代化企业中思维和工作的习惯和方式，了解现代化企业中自动化人才应具备的素质，强化学生工程意识、工程素质，为工程实践能力和创新能力的培养,打下一定的基础。

此次生产实习我申请的是校内自主实习，跟自己的导师作项目。因为现在深度学习非常火热，自动化专业下也有模式识别与智能系统，考虑到之后研究生阶段需要做相关的方向的研究，所以通过自己考虑和老师建议下，在两周的时间内（7 月 15 日至 7 月 31 日），学习一门全球范围内的经典深度学习课程——CS231n（Convolutional Neural Networks for Visual Recognition），以此为以后的模式识别研究打下基础，并且为了保证学习的有效性，必须自己动手编程，完成 assignment1 和 assignment2 所有的作业（根据课程进度，大致一周能够完成一个 assignment），，因为最新年份作业与以往有改变，所以独立性强，全部运行成功才算完成这节的内容。

本次的实习单位就是李晶皎老师所在实验室，可以为我提供运行代码的机器及暑期实习阶段良好的学习氛围和各方面条件。下面是我的具体时间安排：

日期	时间	实施单位	实习内容
7 月 15 日 ~22 日	上午	信息学院	观看课程视频，详细阅读课程文档
	下午		编程实现课程对应的代码 assignment1
7 月 23 日 ~31 日	上午	信息学院	观看课程视频，详细阅读课程文档
	下午		编程实现课程对应的代码 assignment2

2. 正文

2.1 计算机视觉纵览

2.1.1 历史介绍

在我们现代生活中，存在着最多的信息载体就是图像和视频了，人们通过各大视频网站可以观看数以万记的视频，他们都来自我们的生活——自拍，监控，录像，行车记录仪，新闻记者镜头等等。所以我们的视觉是获得信息的主要感官，大脑皮层也有一半左右的神经元是用来处理与视觉相关的信号。所以现在人工智能发展的很大一块就来源与对视频/图像的处理的进步和发展。而计算机视觉的发展并不是单一学科的事情，其与多个学科交叉联合，例如物理学中的光学，生物学中的神经科学等等，下面这张图就展示计算机视觉与其他各学科的互相关联的关系：

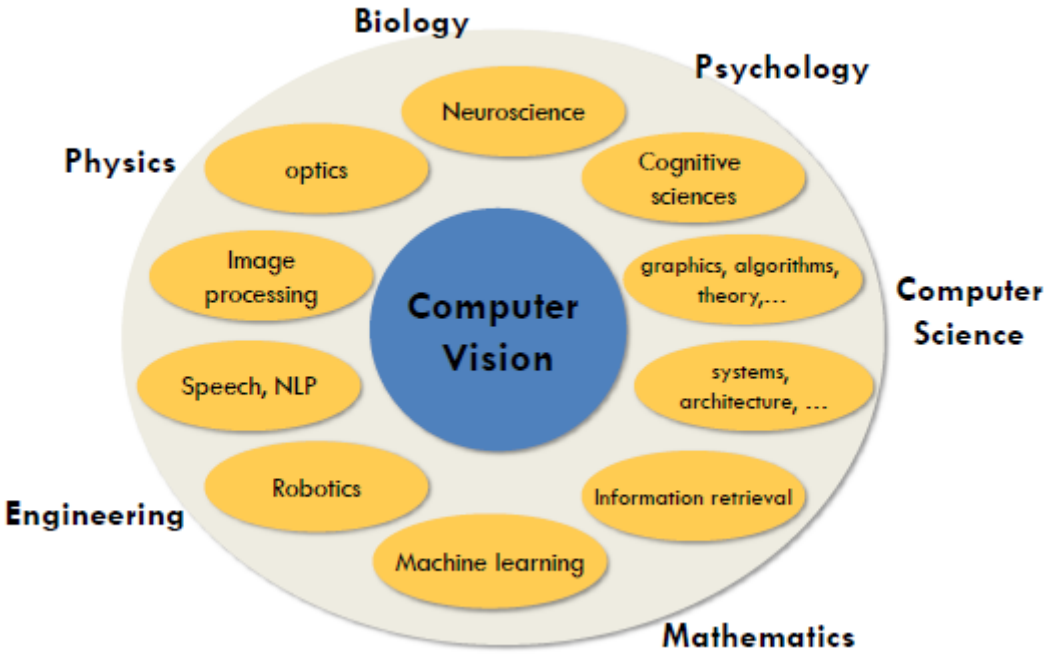

图 2-1 计算机视觉与其他学科的关联

在距今 5 亿 4 千 3 百万年前，生物学家通过考古发现，地球上的生物出现了一次物种大爆炸，物种的数量突然从少数的几种发展出几千万，这种现象至今仍无法很好地解释。一位澳大利亚的生物学家发现在那时的生物首次进化出了眼睛，也就是具备了视觉。视觉的出现，可能是导致物种大爆炸的原因之一，因为这促进了生物之间的捕食和竞争。可以看出，视觉对于生物来说，是高等智慧生

物生存所离不开的一种感觉。而人类通过制造机器获得机器视觉最开始也是模拟生物学开始的：通过小孔成像，光通过小孔，成像在暗箱底板上，从而获得了外界的图像。这也启发了早期照相机的形成。

2.1.2 早期相关算法的发展

早在 1959 年就有科学家 Huel 和 Wiesel 就非常想了解“哺乳动物的视觉处理机制是怎样的”。他们通过对猫的一系列研究发现，在猫的大脑视觉皮层，有一些细胞会对物体朝着某个特定方向运动，产生细胞神经的刺激反应，而一些更为复杂的细胞则会对边缘产生响应，但总的来说，视觉处理是始于视觉世界的简单结构。

而计算机视觉的历史也从 60 年代开始的，在 1963 博士生 Larry Roberts 提出了 Block world 的概念，这也被广泛地认为是世界上关于计算机视觉地第一篇博士论文，他将视觉世界简化为简单的几何形状（Block），目的是能够识别它们，重建这些形状是什么。

在 70 年代，数据少得可怜、计算机速度慢得可怜的情况下，计算机科学家就已经在思考，怎样能够越过用简单的几何形状来描述现实的视觉世界，如何更好地识别和表示对象。所以斯坦福大学的帕洛阿尔托以及斯里兰卡提出一个图形结构的广义圆柱体（Generalized Cylinder）的概念，基本思想是每个对象都是由简单的几何图单位组成，例如人可以通过广义的圆柱体形状拼接在一起或者由一些关键元素按照不同的间距组合在一起，都是通过将复杂的结构体简约成一个集合体。

在 80 年代，David Lowe 希望重建或者识别由简单的物体结构组成的视觉空间。他尝试识别剃须刀，主要是通过线和边缘进行构建，大部分还是直线与直线之间的组合。可以看出，这些早期关于计算机视觉的发展都停留在非常小的样本阶段，不能在现实生活世界中实际地应用。

在 90 年代末期，人们认为如果直接进行目标检测太困难的话，可以先对图像进行目标分割。就是需要将一张图片中的像素点归类到有意义的区域，也就是把在图像中可能属于同一类的事物从背景中扣出来。在 1997 年来自 Berkeley 的 Jitenra Malik 和他的学生 Jianbo Shi 所完成的，他们提出一种 Normalized Cut 图

算法来对图像就行分割，下面这张图中间就显示他们的研究成果。

再往前发展就在统计机器学习方面出现了很多的方法，推进了人脸检测，图像特征匹配相关工作的发展。而摄像机公司更是在相机中实现了能够在镜头中实现实时检测人脸，也逐渐将计算机视觉的工作应用到现实生活中。在 2005 年和 2009 年就有人研究在实际图片中比较合理地设计人体姿态和辨认人体姿态。这一方向后来被称作方向梯度直方图（Histogram of Gradients）和可变形部件模型（Deformable Part Models），如下图右下角所示的：

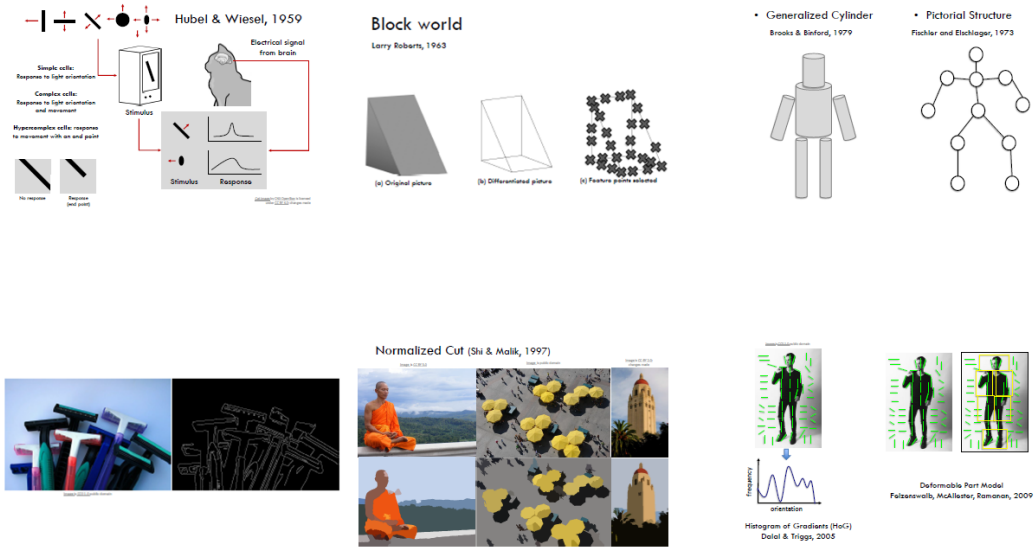

图 2-2 早期计算机视觉研究

最近几年的算法都是基于卷积神经网络构建的不断改进的模型，这在之后会具体提到，在此略过。

2.1.3 数据集介绍

该实验室的 ImageNet 是一个质量非常高的数据集，始于 2009 年李飞飞教授等在 CVPR2009 上发表了一篇名为《ImageNet: A Large-Scale Hierarchical Image Database》的论文[1]，是世界上图像识别最大的数据库，具有大约 1500 万的手动标注的图像，一共有 2 万 2 千多种类别。之后 ImageNet 每年都会举办一次大规模视觉识别挑战赛 Large Scale Visual Recognition Challenge(LSVRC)，直到 2017 年被 Kaggle 维护。就如深度学习的三要素：数据、算法、算力，ImageNet 这样一个高质量的数据库就大大促进了图像方面的研究，开启了深度学习的热潮。

此次课程使用的是一个小型的图像数据库，为了方便我们在笔记本电脑上就能运行出结果：CIFAR-10。CIFAR-10 数据集由 10 个类的 60000 个 32x32 彩色

图像组成，每个类有 6000 个图像。有 50000 个训练图像和 10000 个测试图像。数据集分为五个训练批次和一个测试批次，每个批次有 10000 个图像。测试批次包含来自每个类别的恰好 1000 个随机选择的图像。训练批次以随机顺序包含剩余图像，但一些训练批次可能包含来自一个类别的图像比另一个更多。总体来说，五个训练集之和包含来自每个类的正好 5000 张图像。以下是数据集中的类，以及来自每个类的 10 个随机图像：

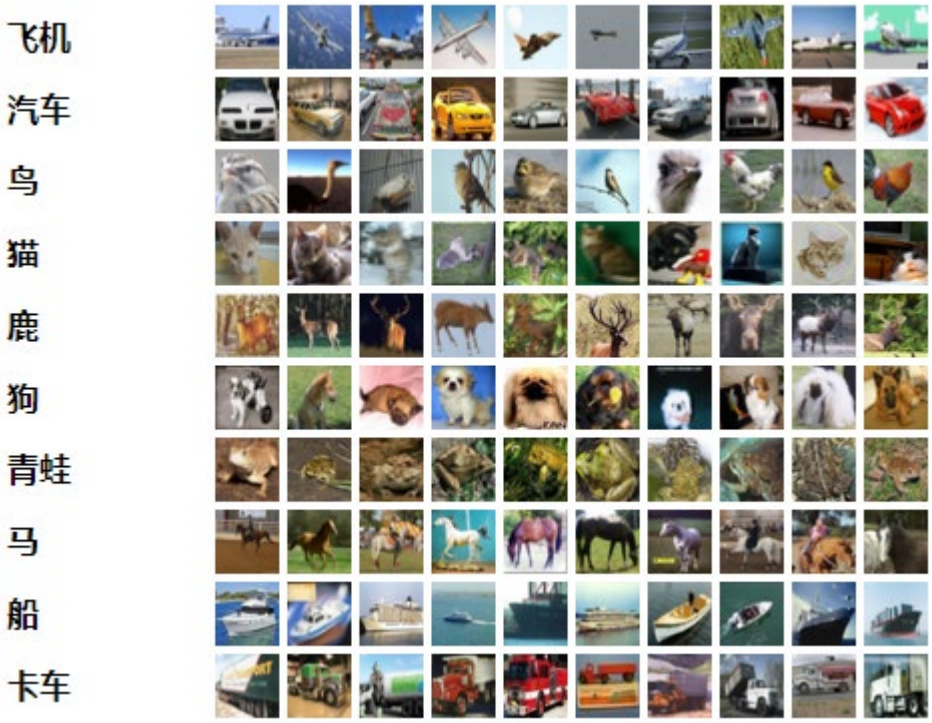

图 2-3 CIFAR-10 数据集类别及每类随机图像

2.2 图像分类—数据驱动方法

所谓图像分类问题，就是已有固定的分类标签集合，然后对于输入的图像，从分类标签集合中找出一个分类标签，最后把分类标签分配给该输入图像。虽然看起来挺简单的，但这可是计算机视觉领域的核心问题之一，并且有着各种各样的实际应用。在这个过程中，对于计算机想要识别成功一个物体还存在很多问题：如视角变化（Viewpoint variation）、大小变化（Scale variation）、形变（Deformation）、遮挡（Occlusion）、光照条件（Illumination conditions）、背景干扰（Background clutter）、类内差异（Intra-class variation）等，所以一般的方法不能很好地解决这些问题。

而所谓数据驱动方法和教小孩儿看图识物类似：给计算机很多数据，然后实现学习算法，让计算机学习到每个类的特征，从而输入一张新的之前从未见过的图片，他也能够给出正确的类别标签。接下来就是我在课程中学到的几种基于数

据驱动的分类方法，有 KNN 和线性分类器模型。

2.2.1 K-Nearest Neighbor

其实相对于 K-Nearest Neighbor，更为一般的就是最近邻（Nearest Neighbor）算法，而 K-Nearest Neighbor 是其扩展，多了一个超参数 K。最近邻算法是一个非常朴素非常自然的想法，简单地说，最近邻算法就是在训练阶段只是单纯记录训练数据和训练标签，并不做任何实质性的处理。在预测阶段，当输入一张新的图片，想要依靠训练过程的结果得出新图片的标签的时候，就是将该新图片与训练集中的所有图片一一做比较，通过某种图片相似性衡量标准，得到与新图片最相近的图片，因为他是属于训练集中的，我们可以得到它的标签，而我们也就将这个标签当作新图片的标签。K-Nearest Neighbor(KNN)在此基础上，将距离最近改为距离最近的 K 张图片，通过选取这最近的 K 个标签中的多数标签当作新图片的标签。这在直观上会改善预测的准确性，因为这避免了某个极端例子对于预测的干扰，使得算法更具广泛性。下图就显示了选取不同的超参数（超参数指的就是不是通过训练能够得到的参数，而是超越训练之上，人为指定的参数值，这些参数值在训练过程中并不会发生变化）K 值，呈现的分类结果，可以看到当 K=1 的时候，分类边界非常得曲折，也有噪点被错分类的孤点情况；而当 K 逐渐增大的时候，分类的边界显得更为平滑，证明此时算法的泛化性能更好！

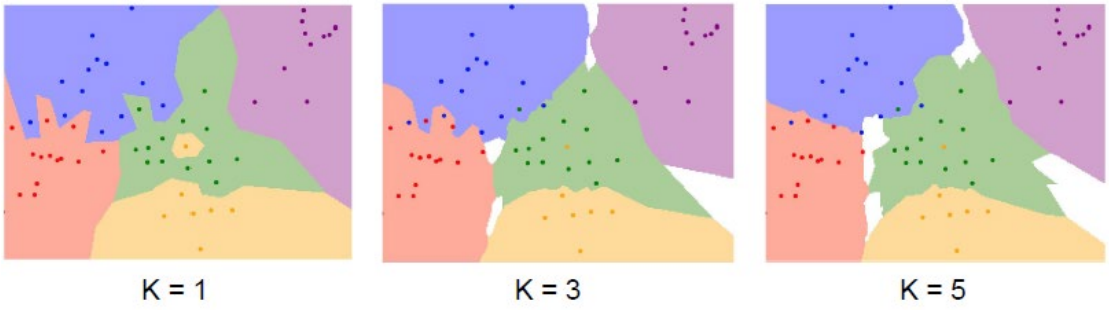

图 2-4 KNN 中选取不同 K 值分类边界的变化

除了有一个超参数 K 需要选择之外，其实在 KNN 算法中，我们还需要选择一个两张图片之间的相近的衡量指标函数，因为有时衡量指标选择的不一样，会造成结果的不一样。常见的有 L1 distance（也叫做曼哈顿距离）、L2 distance（也叫做欧式距离），其中 L1 distance 为：$d_1(I_1,I_2)=\sum_p |I_1^p - I_2^p|$；L2 distance 为：$d_1(I_1,I_2)=\sqrt{\sum_p (I_1^p - I_2^p)^2}$。这里隐含了一个条件就是：两个要比较的图片，大小必须是一致的。

在实际操作的时候，我们会非常自然的问道：这些超参数，例如 KNN 中的 K 值还有距离度量函数，到底选哪一个比较好呢？这其实没有固定的标准，主要还是通过试凑法不断地设置一组超参数，然后用这一组超参数进行训练学习，最后得到在测试集上的准确率等指标，如果有进步就撇弃上一组超参数，然后继续尝试。所以选择超参数是一项繁重耗时的工作，因为没选择不同的一组超参数，就需要重新训练一遍。在此，类似 KNN 的想法，我们采用将数据分成训练集，验证集和测试集。然后采取交叉验证的方法：把训练集分成不同的几等分，依次将其中一份作为验证集，其余当作训练集，然后取其在验证集上的平均准确率，对每一组超参数都进行同样的交叉验证，就可以得到在哪一组超参数下算法表现得更好。下面这幅图就是我在 CIFAR-10 数据集上以 5000 张图片作训练集，500 张图片作为测试集得到不同超参数 K 值情况下交叉验证准确率的曲线图，可以看到在 K=10 左右的时候，平均交叉验证率最高，可以得出超参数选择 K=10 的时候效果最好。

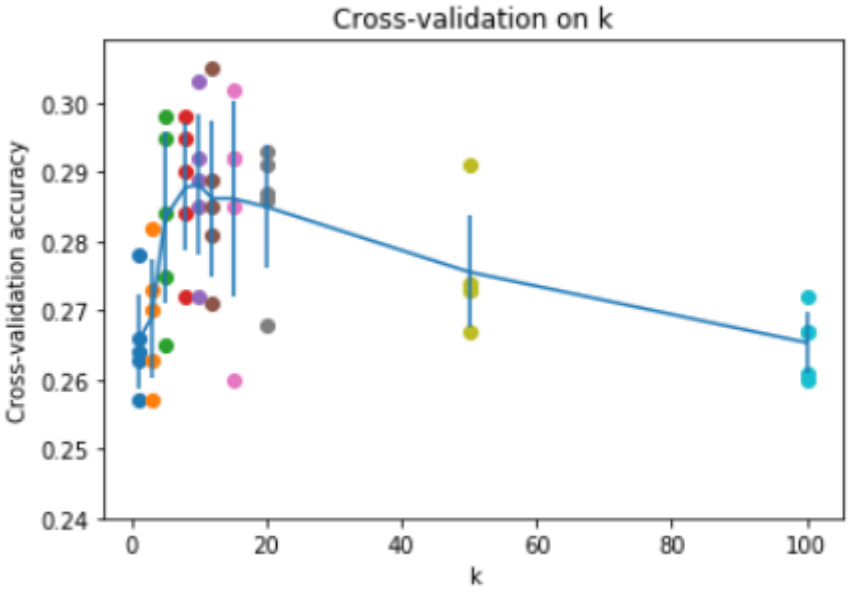

图 2-5 KNN 中不同 K 值的交叉验证曲线图

但是在深度学习中没有使用 K-Nearest Neighbor 算法的，原因有两点：

- KNN 训练复杂度为 O(1)，但是测试复杂度为 O(N)，即测试时太慢了
- 距离衡量指标让 L2 distance 并不能很好的提供图像之间抽象相似的信息

但是，KNN 确实给我们提供了一种直观的感觉，数据驱动式算法就是有训练环节和测试环节，通过交叉验证选择超参数，所以就产生了下面的线性分类器模型。

2.2.2 Multiclass SVM Classifier

我们要规避 KNN 的缺点，就必须以另外一种模式来训练和评价算法。这种方法主要有两部分组成：一个是评分函数（score function），它是原始图像数据到类别分值的映射。另一个是损失函数（loss function），它是用来量化预测分类标签的得分与真实标签之间一致性的。该方法可转化为一个最优化问题，在最优化过程中，将通过更新评分函数的参数来最小化损失函数值。而线性分类器主要体现在第一部分，将原始图像数据映射到类别分值上，这一步使用的是线性函数，即 $f:R^D \to R^K$ 中的评分函数为：

$$f(x_i,W,b)=Wx_i+b$$

其中 x_i 代表一个图像样本，其维度为 D（一个二位图像数据被拉长为一个长度为 D 的向量），通过矩阵乘法和加法就得到了对应图像 x_i 的 K 类类别分值。写成向量化的形式就是：

$$f(X,W,b)=WX+b$$

其中 X 为 N 个图像的集合，维数为[D x N]，大小为[K x D]的矩阵 W 和大小为[K x 1]列向量 b 为该函数的参数（parameters）。参数 W 被称为权重（weights）。b 被称为偏差向量（bias vector）。最后每一个图像都有对应每一类的分值。下面这张图很好地描绘出这种线性映射关系：

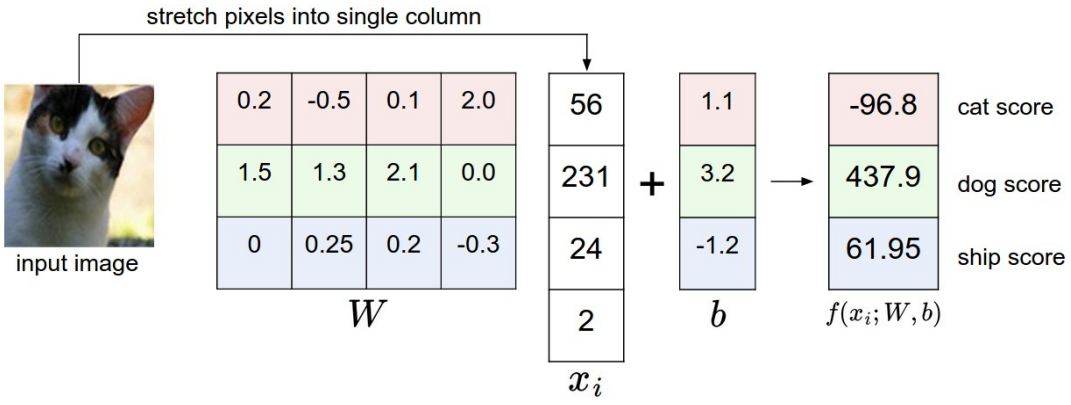

图 2-6 线性映射的评分函数

损失函数（loss function），它是用来衡量我们对预测结果的“不满意度”。当预测值与真实标签（Ground Truth）接近的时候，我们的损失值就小；当预测值与真实标签背离的时候，我们的损失值就增大，从而得到反馈，不断地更新参数，使得最终的损失值变小。

其中一种损失函数就是：多分类支持向量机损失 Multiclass Support Vector

Machine Loss，以这种为损失函数的分类器就叫做 SVM Classifier，其公式就是：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

其中，L_i 为第 i 个数据中的损失值。S 为不同分类类别的分值，如针对第 j 个类别的得分就是第 j 个元素 $s_j = f(x_i, W)_j$。从该损失函数的定义可以看出，只有在正确分类的类别分值 s_{y_i} 比其他类别分值高出一个阈值 Δ 的时候，损失值才为 0，否则损失值都会是一个正值，遍历整个数据集就是累加每个图像的损失值。如果和前面的线性矩阵运算结合起来，也可以写成这样：

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

下面这张图很好地解释了多分类支持向量机损失的含义：

图 2-7 多分类支持向量机损失函数直观示意图

2.2.3 Softmax Classifier

与 SVM Classifier 类似，Softmax Classifier 也是基于线性分类模型的，只是与 SVM Classifier 的损失函数不一样而已，也可以理解为逻辑回归分类器面对多个分类的一般化。它使用的是交叉熵损失（cross-entropy loss）：

$$L_i = -\log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$$

其中 $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ 被称作 softmax 函数，能将输入向量压缩到 0 和 1 之间，并且输出向量各元素之和为 1，这也是 Softmax Classifier 名称的由来。并且从信息熵的角度看交叉熵损失，可以理解为一种对估计分布与真实分布之间的一种度量，损失小的时候表明这两种分布之间比较接近。而这样的输出甚至也可以看成“概率”，因为这个交叉熵损失是负对数损失，所以当正确类别的概率越大时，损失是越低的，这也符合我们的预期。下面这幅图展示了两种分类器计算损失时的不同，两种计算方法经常是相似的效果，实际使用中人们更愿意使用解释性更强的 softmax loss。

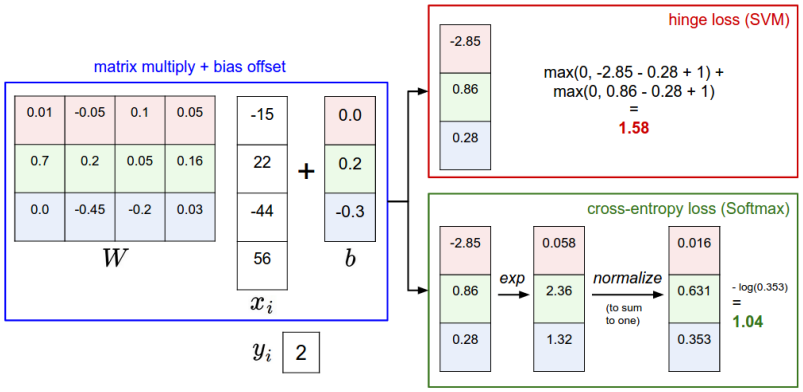

图 2-7 SVM 与 Softmax 的比较

2.2.4 正则化

在 SVM loss 中存在着参数矩阵 W 不唯一的现象，因为一旦得到使损失为 0 的权重 W，那么关于 W 的大于 1 的数乘都可使损失为 0。我们希望能向某些特定的权重 W 添加一些偏好，对其他权重则不添加，以此来消除模糊性。方法就是向损失函数里面添加一个正则化惩罚（regularization penalty）部分，常见的正则化项是 L2 范式：$R(W)=\sum_k\sum_l W_{k,l}^2$。这样数据损失就由两部分构成，一个是样例的平均损失 L_i，另一项就是正则化损失（regularization loss），则完整的 SVM Loss 公式可以写成：

$$L=\frac{1}{N}\sum_i\sum_{j\neq y_i}[\max(0,s_j-s_{y_i}+\Delta)]+\lambda\sum_k\sum_l W_{k,l}^2$$

完整的 Softmax Loss 公式可以写成：

$$L=-\frac{1}{N}\sum_i\log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})+\lambda\sum_k\sum_l W_{k,l}^2$$

其中 λ 是正则化强度，也是一个超参数，用来平衡两种损失。注意正则化项只针对参数权重 W，而不正则化偏差 b。L2 惩罚倾向于更小更分散的权重向量，这就会鼓励分类器最终将所有维度上的特征都用起来，而不是强烈依赖其中少数几个维度。正则化将会提升分类器的泛化能力，并避免过拟合。

2.2.5 模板匹配

其实还可以从模板匹配这种角度给我们一种对于线性分类器一种直观上的感觉。因为线性模型总要经过矩阵相乘的运算 $f(X,W,b)=WX+b$，得到的分值矩阵可以看成是矩阵 W 的每一行与列向量 X 求内积的结果，如果两个向量之间相似度越高，则正确的类别分值就会越高，从而使损失越低，这就造成了 W 矩

阵其实每一行向量如果按照 X 从二维向量变为一维向量的方式堆成二维向量，可以想象得到，如果对权重矩阵的每一个行向量进行可视化（图像有多少类，W 矩阵就有多少行），会发现与数据集的每一类数据是比较接近的，或者是他们的一个模板，下面就是我用两种不同的损失函数对权重矩阵 W 可视化的结果，发现是符合我们的预期的：

图 2-8 SVM 损失下权重矩阵可视化模板

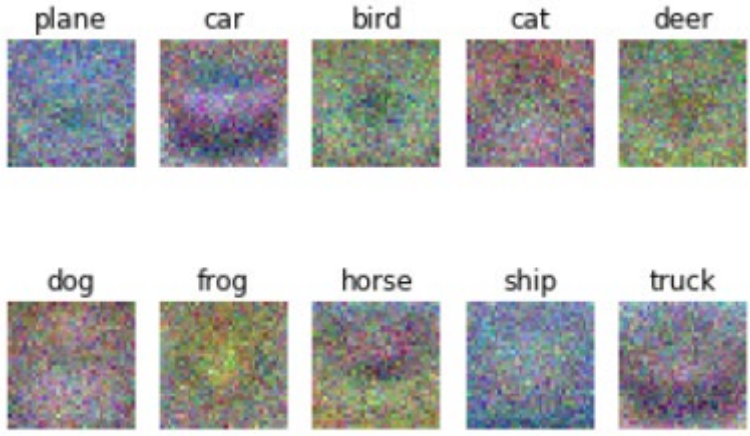

图 2-9 Softmax 损失下权重矩阵可视化模板

2.3 图像分类—神经网络

在以上，我们已经学会了更能高效分类的线性模型，其中就包括需要学习的权重矩阵 W 和偏置 b。我们之前就假定已经有一组参数，但是我们仍然会产生这样的问题？这样的参数是最好的了吗？如果不是，那又怎样来获得最好的参数呢？在这里，衡量参数的好坏是看损失值是否下降，选取一组最好的参数也就是找到一组参数使得损失值最小。

2.3.1 优化方法

一种自然朴素的想法就是随机搜索：随机选择不同的权重，然后选取其中最好的，用 python 写的伪代码是这样的：

```
bestloss = float("inf") # Python assigns the highest possible float value
for num in range(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print('in attempt %d the loss was %f, best %f' % (num, loss, bestloss))
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test exa
mples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
```

这样的寻找最优的权重策略在测试集上的准确率能够达到 15.5%，而针对 CIFAR 的 10 分类问题，完全随机猜测的正确率仅有 10%，虽然比完全随机猜测要好一点，但是这显然不能达到要求。

一种更为行之有效的方法就是梯度下降(Gradient Descent)。梯度下降是深度学习中最普遍的优化方法，很多其他有效的方法的基础就是梯度下降。具体来说，因为很多问题的损失函数是一个凸函数(convex function)，所以沿着梯度的负方向更新一步，损失值就减小一点，通过一次次的迭代，最终就能达到最优的状态。每次更新的步长就是学习率(learning rate)，也是一个超参数。下面这幅图显示了高维损失函数梯度下降示意图：

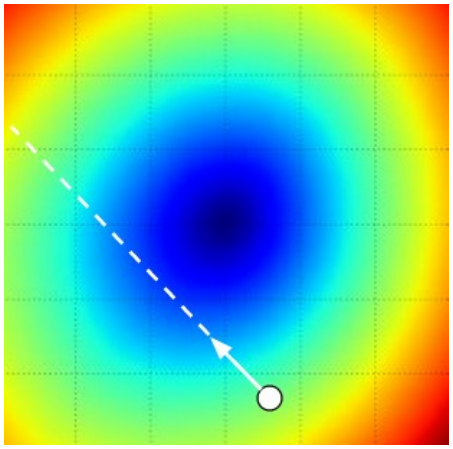

图 2-10 梯度下降可视化图

普通的梯度下降的 python 伪代码如下所示：

```
while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

对于之前的 SVM loss 损失函数而言， $L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$，我们可以求出损失函数关于权重矩阵 W 的解析梯度，其中 1(.)算子为逻辑算子，当括号中的表达式成立的时候，返回 1，否则返回 0：

$$\nabla_{w_{y_i}} L_i = -(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)) x_i$$

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

在实际使用中，要计算整个数据集上的梯度，这时的梯度下降叫做批梯度下降 BGD(Batch Gradient Descent)，数据集比较大的时候，这个计算代价是很大的。相比于梯度下降，会使用一小批(minibatch)的训练图像来计算当前的梯度，从而来更新参数。只要 batch_size 不要太小，他们的分布还是能在一定程度上代表整体数据的分布，。而当 batch_size 取为 1 的时候，这种特殊情况称作随机梯度下降 SGD(Stochastic Gradient Descent)。虽然 SGD 收敛得没有 BGD 平滑，但收敛速度总体还是比 BGD 快。python 的伪代码如下所示：

```
while True:
  data_batch = sample_training_data(data, batch_size) # sample examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

2.3.2 全连接网络

之前的模型都属于线性模型，对于线性可分的问题可以较好地处理，但是对于非线性问题就无能为力了，比如对于下面这样的二维问题，并不能通过一条直线将两类物体分开：

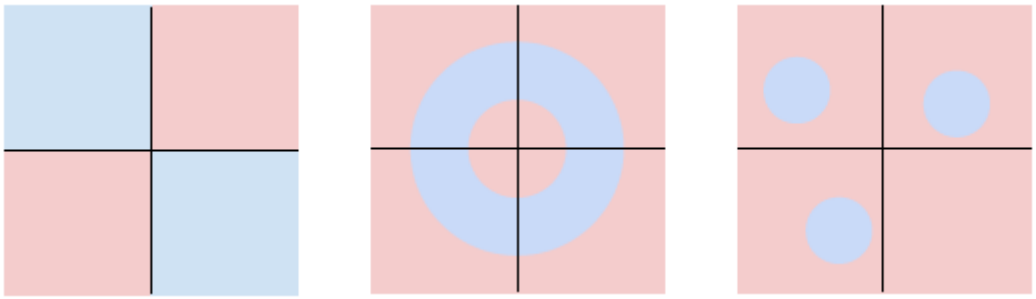

图 2-11 线性分类器难处理的情况

这时需要一个更高级的模型来帮助我们解决这样的问题。我们可以从人类大脑的神经细胞得到启发，人类的神经系统中大约有 860 亿个神经元，它们被大约 10^14-10^15 个突触（synapses）连接起来。下面图表的左边展示了一个生物学的神经元，右边展示了一个常用的数学模型。每个神经元都从它的树突获得输入信号，然后沿着它唯一的轴突（axon）产生输出信号。轴突在末端会逐渐分枝，通过突触和其他神经元的树突相连。在数学模型中，树突将信号传递到细胞体，信号在细胞体中相加。如果最终之和高于某个阈值，那么神经元将会激活，这个激活通过激活函数实现，表达了轴突上激活信号的频率。整个过程的示意图如下：

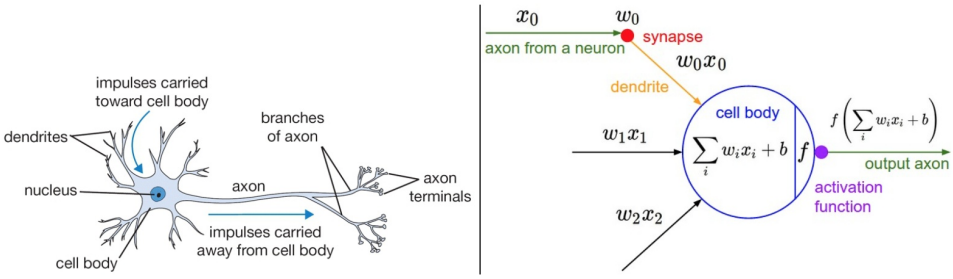

图 2-12 单个神经元模型

可以看到，相比于普通的线性模型，神经元模型只是在原来的基础上，加了一个非线性环节——激活函数。常见的激活函数有 sigmoid、tanh、ReLU 函数，他们的数学表达式和图像分别列在下面：

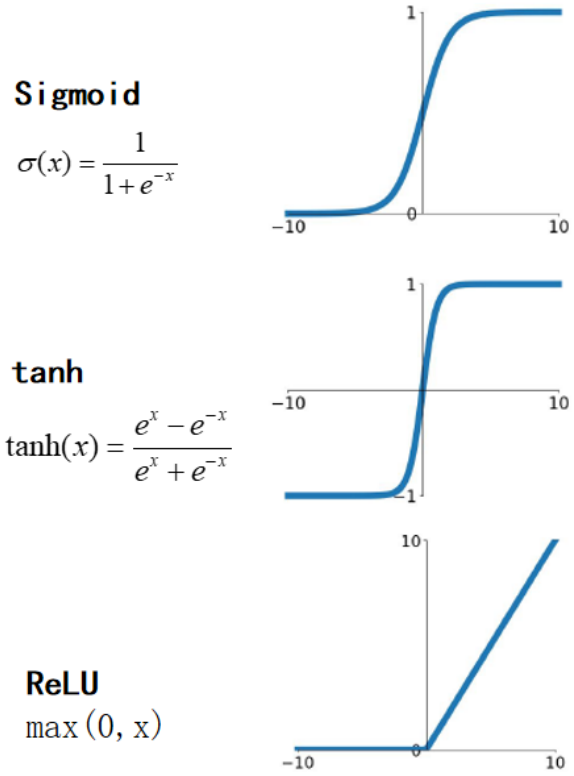

图 2-13 常见激活函数及图像

将神经网络模型（这里列举 3 层神经网络为例）用数学表达式表达出来就是：

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

其中 $x \in R^D, W_1 \in R^{H_1 \times D}, W_2 \in R^{H_2 \times H_1}, W_3 \in R^{C \times H_2}$，max(0,_)算子将数与 0 比较，返回两者之间的大值，也就是激活函数 ReLU。通常神经网络模型中神经元是分层的，而不是像生物神经元一样聚合成大小不一的团状。对于普通神经网络，最普通的层的类型是全连接层（fully-connected layer）。全连接层中的神经元与其前后两层的神经元是完全成对连接的，但是在同一个全连接层内的神经元之间没有连接。下面这个就是 3 层的全连接神经网络的示意图（层数计算中不包括输入层，这里的输出层并不一定都是一个输出）：

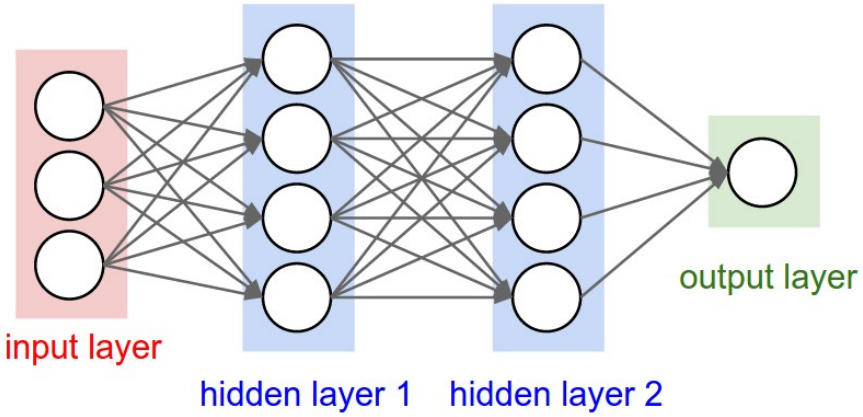

图 2-14 3 层全连接神经网络示意图

这里从输入层到输出层的一个信号流动，就完成了神经网络的前向传播(forward pass)，也就是不断重复的矩阵乘法与激活函数交织。这是神经网络组织成层状的一个主要原因，就是这个结构让神经网络算法使用矩阵向量操作变得简单和高效。下面就是用 python 写的一个 3 层神经网络的前向传播：

```
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)# calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

2.3.3 全连接网络的反向传播

之前的优化方法中就提到了通过梯度下降法来寻找最优的参数，使得损失函数值达到最小。全连接神经网络的反向传播(backward pass)与此类似，当前向传播中，我们计算出了各神经元的激活值，还有最终的损失值，现在就需要从总的损失值计算关于各层之间的权重矩阵 W 和偏置 b 的梯度，然后根据梯度来更新参数。这其中运用一种叫做计算图(computational graph)的概念，可以把复杂

的复合运算，拆成一个个简单的基本的四则运算，也就是一个个门(gate)，然后通过链式法则(chain rule)，把上游梯度(upstream gradient)乘以本地梯度(local gradient)就得到了当前的梯度，然后重复这个步骤，从当前往下游(downstream)传播，直至输入层。以 $f(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$ 为例，这是关于 w,x 的函数，想要得到 w,x 的梯度，整个式子虽然比较复杂，但拆分后看到也只是包含了四则运算和指数运算，下面这张图很好地展示了计算图对于计算梯度的便捷性与清晰性：

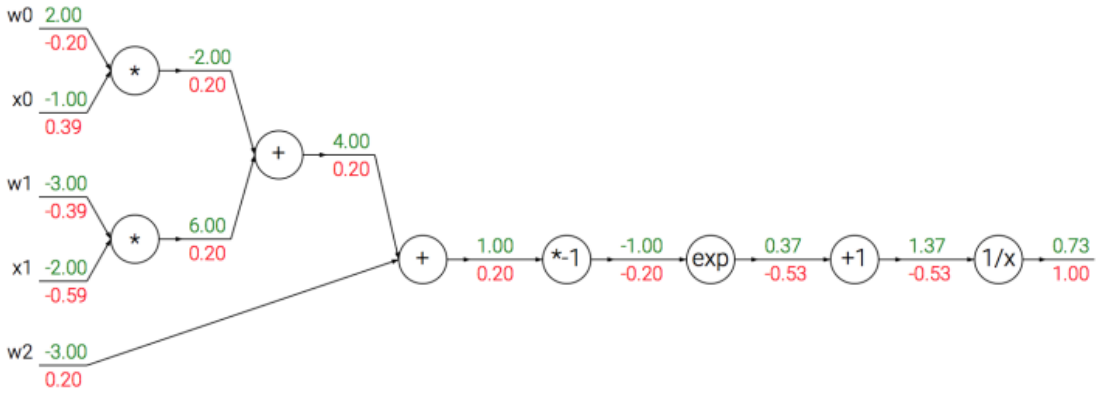

图 2-15 计算图反向传播例子

图 2-15 计算图计算梯度，绿色数值代表各门正向传播时的值，红色数值代表各门的反向传播的梯度值

从中我们可以总结出对于一些基本的门，例如加、减（可以看成特殊的加法）、乘、除（可以看成特殊的乘法），取最大值等的梯度传播的法则：

- **加法门单元**：把输出的梯度相等地分发给它所有的输入，这一行为与输入值在前向传播时的值无关。这是因为加法操作的局部梯度都是简单的+1，所以所有输入的梯度实际上就等于输出的梯度，因为乘以 1.0 保持不变。
- **取最大值门单元**：对梯度做路由。和加法门不同，取最大值门将梯度转给其中一个输入，这个输入是在前向传播中值最大的那个输入。这是因为在取最大值门中，最高值的局部梯度是 1.0，其余的是 0。
- **乘法门单元**：它的局部梯度就是输入值，但是是相互交换之后的，然后根据链式法则乘以输出值的梯度。相当于上游梯度经过乘法门时会互相交换。

对于全连接神经网络，前向传播中涉及到的运算就只是矩阵乘法和激活函数，对于 sigmoid 函数也就是取最大值门。对于向量 $x\in R^N, y\in R^M$，根据多

元函数的梯度，可以得到向量 y 对于向量 x 的梯度是一个雅可比矩阵（Jocobian）：$\frac{\partial y}{\partial x} \in R^{N\times M}, (\frac{\partial y}{\partial x})_{n,m} = \frac{\partial y_m}{\partial x_n}$。对于矩阵乘法 $f(X,W,b) = WX + b$，[2]中给出了矩阵之间的梯度推导，一般情况下，矩阵 f 对于 W 和 X 的梯度是一个高维的雅可比矩阵，而且是一个稀疏矩阵，直接存储会占用大量的内存，但其实推导出的结果却非常简洁高效：$\frac{\partial f}{\partial W} = \frac{\partial L}{\partial f} X^T, \frac{\partial f}{\partial X} = W^T \frac{\partial L}{\partial f}$，其中 $\frac{\partial L}{\partial f}$ 表示上游梯度。而激活函数也可以直接利用前向传播中计算出来的值计算当前梯度，所以这样损失对于各参数的梯度就都可以计算出来了。

2.4 卷积神经网络

卷积神经网络(Convolutional Neural Network)和之前的神经网络非常相似：它们都是由神经元组成，神经元中有具有学习能力的权重和偏差。每个神经元都得到一些输入数据，进行内积运算后再进行激活函数运算。整个网络依旧是一个可导的评分函数：该函数的输入是原始的图像像素，输出是不同类别的评分。在最后一层（往往是全连接层），网络依旧有一个损失函数（比如 SVM 或 Softmax）。

卷积神经网络针对输入全部是图像的情况，将结构调整得更加合理，变为了三维排列(3D volumes of neurons)。与常规神经网络不同，卷积神经网络的各层中的神经元是 3 维排列的：宽度、高度和深度（这里的深度指的是激活数据体的第三个维度，而不是整个网络的深度，整个网络的深度指的是网络的层数）。举个例子，CIFAR-10 中的图像是作为卷积神经网络的输入，该数据体的维度是 32x32x3（宽度，高度和深度）。我们将看到，层中的神经元将只与前一层中的一小块区域连接，而不是采取全连接方式，并且还有权值共享的特点。下面这张图描绘了卷积神经网络的整体结构特点：

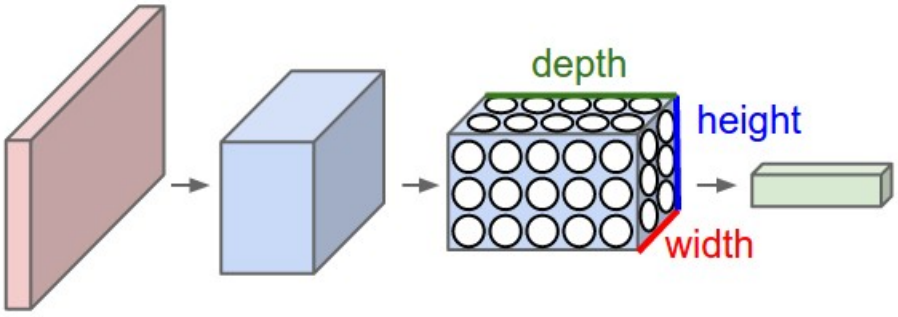

图 2-16 卷积神经网络架构简述

卷积神经网络主要由三种类型的层构成：卷积层，池化（Pooling）层和全连接层（全连接层和之前神经网络中的一样）。通过将这些层叠加起来，就可以构

建一个完整的卷积神经网络。卷积神经网络(CNN)自从在 2012 年的 LSVRC 取得突破性的成绩之后，就引起了一大波研究热潮。下面介绍卷积神经网络中的两种基本的层结构：卷积和池化。

2.4.1 卷积和池化

卷积操作基本在各种卷积神经网络中都存在的操作，通过一层一层地卷积，就对输入的图像提取更高级抽象的特征。二维卷积在数学上的定义为：$y(p,q)=\sum_{i=0}^{M}\sum_{j=0}^{N}h(p-i,q-j)u(i,j)$。卷积层的参数是有一些可学习的卷积核（也叫滤波器）集合构成的。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。举例来说，卷积神经网络第一层的一个典型的滤波器的尺寸可以是 5x5x3（宽高都是 5 像素，深度是 3 是因为图像应为颜色通道，所以有 3 的深度）。在前向传播的时候，让每个滤波器都在输入数据的宽度和高度上（也就是在 spatial 维度上）滑动（更精确地说是卷积），然后计算整个滤波器和输入数据任一处的内积。当滤波器沿着输入数据的宽度和高度滑过后，会生成一个 2 维的激活图（activation map），激活图给出了在每个空间位置处滤波器的反应。在处理图像这样的高维度输入时，让每个神经元都与前一层中的所有神经元进行全连接是不现实的。相反，我们让每个神经元只与输入数据的一个局部区域连接。该连接的空间大小叫做神经元的感受野（receptive field），它的尺寸是一个超参数（其实就是滤波器的空间尺寸）。在深度方向上，这个连接的大小总是和输入量的深度相等。

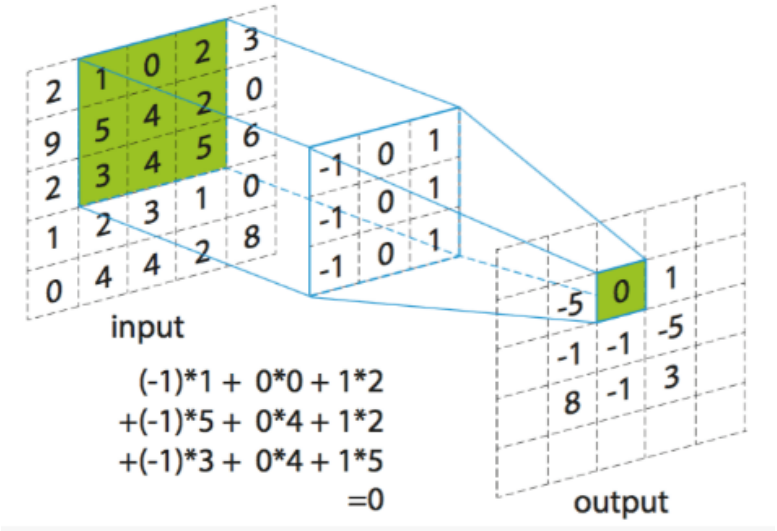

图 2-17 卷积神经网络卷积操作

池化也叫做降采样（downsampling）。在连续的卷积层之间会周期性地插入一个池化层。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。池化层使用 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。最常见的形式是池化层使用尺寸 2x2 的滤波器，以步长为 2 来对每个深度切片进行降采样，将其中 75%的激活信息都丢掉。每个 MAX 操作是从 4 个数字中取最大值（也就是在深度切片中某个 2x2 的区域）。深度保持不变。最大池化反向传播时得先记录前向传播时最大值索引，然后将上游梯度传给最大值所在位置。常见的池化有均值池化和最大池化。下面这幅图演示的就是最大池化：

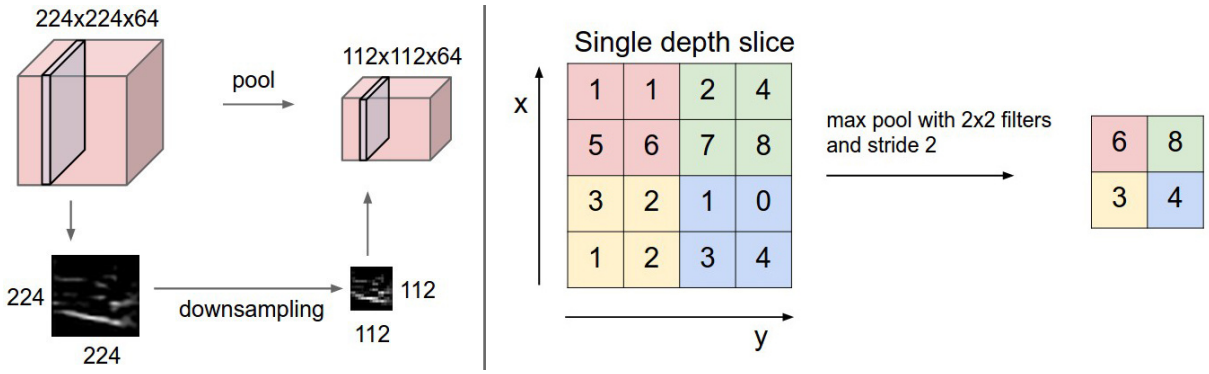

图 2-18 卷积神经网络最大池化操作

在设计卷积神经网络架构时，通常要考虑空间维数的匹配，下面的表 1 就总结了关于卷积和池化操作，从输入到输出的激活值的维数之间的关系：

表 1 卷积池化操作相关计算公式

操作	卷积	池化
输入数据体大小	$W_1 \times H_1 \times D_1$	$W_1 \times H_1 \times D_1$
需要指定的超参数	卷积核数： K 卷积核大小： F 步长： S 零填充数量： P	卷积核大小： F 步长： S
输出数据体大小	$W_2 \times H_2 \times D_2$	$W_2 \times H_2 \times D_2$
计算公式	$W_2 = (W_1 - F + 2P)/S + 1$ $H_2 = (H_1 - F + 2P)/S + 1$ $D_2 = K$	$W_2 = (W_1 - F)/S + 1$ $H_2 = (H_1 - F)/S + 1$ $D_2 = D_1$

2.4.2 Dropout

在之前也介绍过，在损失函数中加上一项惩罚项，来进行正则化，使得不要

过拟合。在神经网络的训练中除了上面这种方法，还有很多正则化的方法：Dropout[3]、Batch Normalization[4]、Data Augmentation、DropConnect[5]、Fractional Max Pooling[6]、Cutout[7]、Mixup[8]等等。在这里介绍两种常用并且比较成熟，在各大框架中都已经有集成的正则化方法：Dropout 和 Batch Normalization。

Dropout 是一个简单又极其有效的正则化方法，与 L1 正则化，L2 正则化和最大范式约束等方法互为补充。在训练的时候，随机失活的实现方法是让神经元以超参数 P 的概率被激活或者被设置为 0。对于 Dropout 有几种解释：（1）网络有可能提取了冗余的特征，Dropout 能防止特征的互适；（2）Dropout 可以被认为是对完整的神经网络抽样出一些子集，在测试过程中不使用随机失活，可以理解为是对数量巨大的子网络做了模型集成（model ensemble）。下面这张图显示了应用 Dropout 前后的全连接神经网络的模型的示意图：

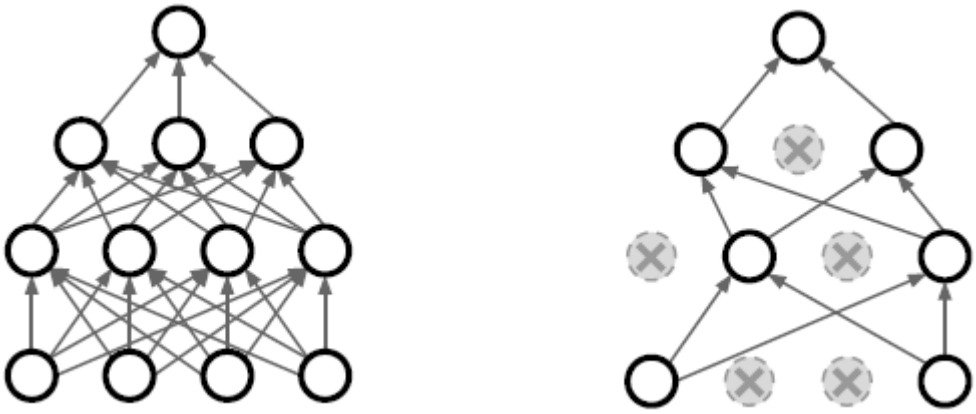

图 2-19 Dropout 应用前后对比图

因为在训练过程中应用 Dropout，这引入了一种随机过程。在测试过程中为了消除这种随机性，需要进行一个缩放，伪代码如下：

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
```

```
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

2.4.3 Batch Normalization

Batch Normalization 是[4]在 2015 年提出来的。自从提出后因为特别好的效果，收到了广泛的关注。IID 独立同分布假设，就是假设训练数据和测试数据是满足相同分布的，这是通过训练数据获得的模型能够在测试集获得好的效果的一个基本保障，这是机器学习中的一个很重要的假设。论文中引入了“Internal Covariate Shift”的概念，它是作者指出深度神经网络收敛慢，难训练的原因，意思就是深层神经网络在做非线性变换前的激活输入值随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动。一般是整体分布逐渐往非线性函数的取值区间的上下限两端靠近（对于 Sigmoid 函数来说，意味着激活输入值是大的负值或正值），所以这导致反向传播时低层神经网络的梯度消失，这是训练深层神经网络收敛越来越慢的本质原因，而 Batch Normalization 就是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值为 0 方差为 1 的标准正态分布，其实就是把越来越偏的分布强制拉回比较标准的分布，这样使得激活输入值落在非线性函数对输入比较敏感的区域，这样输入的小变化就会导致损失函数较大的变化，避免梯度消失问题产生，而且梯度变大意味着学习收敛速度快，能大大加快训练速度。其算法如下：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

图 2-20 Batch Normalization 前向传播算法论文截图

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

图 2-20 Batch Normalization 反向传播算法论文截图

2.4.4 卷积神经网络的反向传播

这是卷积过程的详细图：

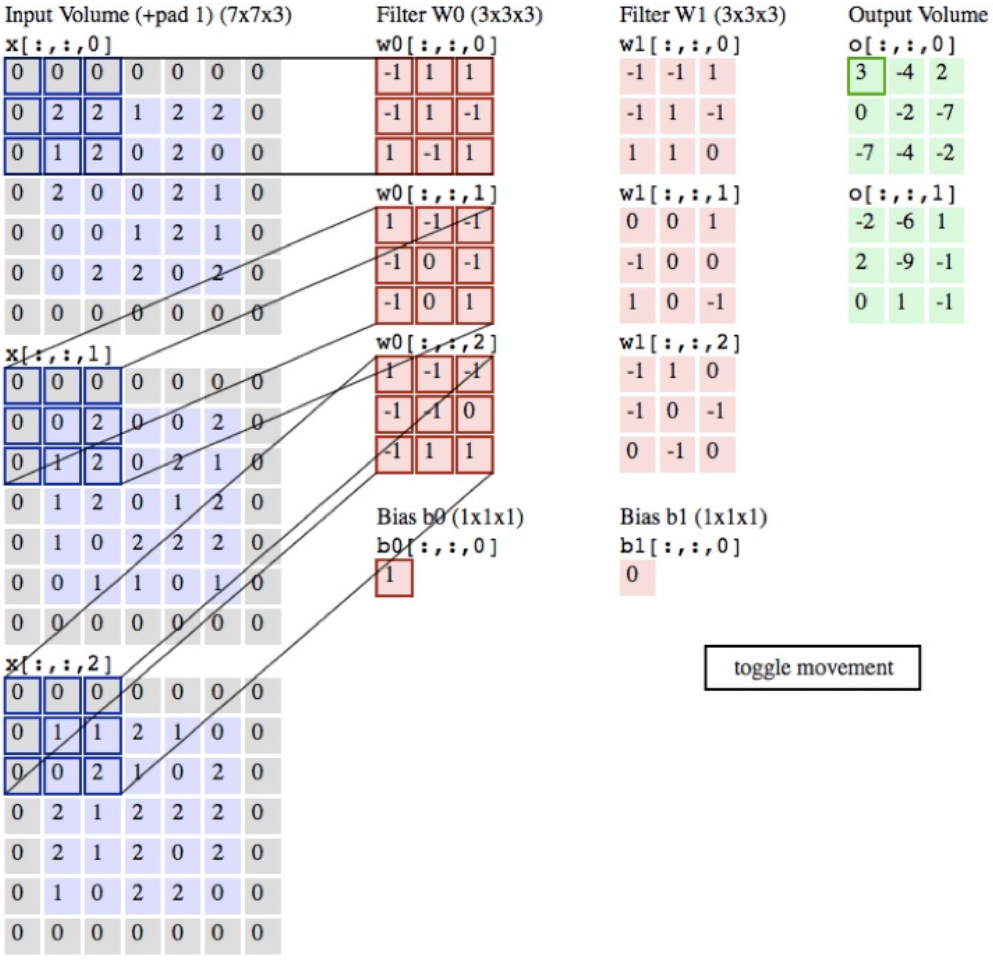

图 2-21 卷积详细图

在以上的过程中，我们对 Dropout、Batch Normalization、池化都有说明反向传播时怎么处理，针对卷积过程，我们可以看到是卷积核与局部感受野作内积产生出 feature map 中的一块响应值。而内积就是加和和乘积运算，可以根据之前的加和门与乘积门，找到卷积核滑动的规律，就可以由上游梯度计算出对于图像像素输入和卷积核的梯度，下面是我用 python 写的非常自然朴素的卷积反向传播代码：

```
def conv_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a convolutional layer.

    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None

    # TODO: Implement the convolutional backward pass.
    #

    x, w, b, conv_param = cache
    stride = conv_param['stride']
    pad = conv_param['pad']
    N, F, H_out, W_out = dout.shape
    F, WC, HH, WW = w.shape

    dw = np.zeros(w.shape)
    dx = np.zeros(x.shape)
    db = np.zeros(b.shape)

    stride = conv_param['stride']
    pad = conv_param['pad']
    for data_point in range(N):
        xx = np.pad(x[data_point,:,:,:], pad_width=((0,0),(pad,pad),(pad,pad)), mode='constant')
        dx_x = np.zeros(xx.shape)
        for filt in range(F):
```

```
            for hh in range(H_out):
                for ww in range(W_out):
                    dw[filt,:,:,:] += dout[data_point,filt,hh,ww] * xx[:,stride*hh:stride*hh+HH,stride*ww:stride*ww+WW]
                    db[filt] += dout[data_point,filt,hh,ww]
                    dx_x[:,stride*hh:stride*hh+HH,stride*ww:stride*ww+WW] += dout[data_point,filt,hh,ww] * w[filt,:,:,:]
        dx[data_point,:,:,:] = dx_x[:,pad:-pad,pad:-pad]

    return dx, dw, db
```

3. 结论

3.1 实习内容与平时所学的关联

虽然计算机视觉听上去更像是计算机专业的东西，但我觉得其实里面还是和自动化里面有很多东西是非常相近的，里面的算法和思想与平常所学的还是有关联的。比如其实对于一个分类问题（对于其他方向的问题也是这样），我们除了需要构建一个合适的模型之外，找到一个能够正确表示预测值和真实值之间的损失函数也是非常重要的，因为一个不正确的损失函数不能按照我们的要求的效果进行学习，而减小损失本质就是一个优化问题，在专业学习的过程中我们其实学了很多的优化方法，梯度下降只是其中一个，还有很多其他算法，包括一些智能启发式算法，只是两者针对的问题不一样，我觉得之后有可能可以在优化算法方面结合专业知识，在这方面提出一些创新性算法。

还让我感觉与所学的自动化控制中比较接近的思想就是闭环的思想。我们在自动控制原理等系列课程就一直被灌输这样的概念：闭环比开环更稳定，而这主要就是有了反馈。在卷积神经网络模型里面，不仅有正向传播，也有反向传播，这其实就是充当了“反馈”的角色，只有通过反向传播，才能够让参数得到更新，最好达到“稳定”。我觉得以后在这方面也可以将这两方面结合起来，甚至能将 PID 的思想融入其中，可能会取得不一样的令人惊讶的效果。

3.2 实习感悟

这接近半个月的实习让我对计算机视觉和图像识别有了一定的认识，也对我们自动化专业的所学有了更深的认识，通过这样的学习，我自己也萌发了几点想法，在这里总结一下。

认真学好基础理论知识

根基不牢，地动山摇。这充分体现了基础知识的重要性。在学习的过程中，

我之前认为学习一定要学好，但只是单纯地因为从小的教育，或者说学习好能找到一个好工作，但是没有想过基础知识是很重要的，历史上的很多国家或者企业能够屹立不倒，很多就是重视基础教育。在学习 CS231n 这门课的时候，就感觉到之前的高等数学，线性代数，概率论与数理统计，包括英语在内都会大有用场。这些都是大一上的基础课程，但是这些却是之后深入研究所不可缺少的，一些理论知识方面的推导理解不到位，是不可能通过代码复现出来的。

也许我们现在感觉自己学的东西都没用，但是在以后的研究中，我们会发现，我们不定什么时候就会用到所学的知识。我们所认真学习的每一项都将会为我们以后的成功打好基础。大三的我们或许失去了大一大二刚来的时候的满腔热血，但经过这次实习，我们应该清楚地认识到，无论何时，都应该踏实学习，都应该饱含激情，你的每一份努力都会得到回报。

脚踏实地实践研究

我在学习这门课程的时候，不免有时候会在理论上遇到很多困难，并且自己的编程基础不是特别扎实。有时在理论部分还没真正搞懂的时候就想着去编代码，完成作业，去赶进度。这样实际的效果其实是比花时间把理论知识搞懂以后再去编写代码还要慢，所以我们在学的过程中，一定要踏实，你骗得了别人，却不能骗自己，更骗不了以后工作上的种种问题。不懂的话就是不懂，这时候需要去寻找各种关于这方面的知识点，通过各种手段去让自己理解这块知识。

在大学两三年的学习生活中，理论学习占了我们大多数时间，而深入实践的动手操作机会不多，我们很容易就能发现理论学习和动手实践决不能一概而论。实践是检验真理的唯一标准，因此，我们在平时做实验时，不要只想快点做完实验，达到实验效果就可以了，我们应该更多的去关注实验过程中遇到的问题，一步步地把实验过程中遇到的问题弄明白、解决掉，这样才能达到做实验的目的。我们才能在实践的过程中有所收获。“纸上得来终觉浅，绝知此事要躬行”，我们要积极主动地寻找实践机会，将理论知识合理运用到实际生活中。

独立思考和争取创新

我在这次学习的过程中，体会到创新的作用，也体会到创新并不是一件难事。因为创新，所以人们能够推动这一方向的飞速发展，就拿 LSVRC 比赛来说，因为每年都有新的方法模型，或者在参数上有了更好的调整，使得每年比赛的错误率都在大幅下降，从 2012 年开始使用卷积神经网络模型开始，到 2015 年，仅用了三年的时间，就让模型的准确率超越了人类，这一切的背后正是比赛推动了参赛选手的创新，鼓励全球范围内的研究人员参与进来。但是就如 mini batch 的使用，其实原来整个数据集都投入训练计算量很大，应该可以很自然想到只用以小批量的数据来训练，这并不是非常高深的想法；对于 GoogleNete、ResNet 等经典神经网络模型，其中的 inception 和 residual block 其实都是遇到问题之后产生

的非常自然的想法，但却取得了不错的效果，发表在顶会上。所以之后我也要在遇到问题时看看自然的想法能否成功。

我们在参加数模竞赛和电子设计大赛的时候也能够发现，只有创新、另辟蹊径才能取得更好的成绩，而创新精神不是说有就有的，是需要长时间培养的。因此，在日常的学习中，我们要注意培养自己的独立思考能力和创新精神，遇到问题的时候，一定要想想，除了这个，有没有更好的解决方法，培养自己的创新精神。

耐得了寂寞

在暑期这半个月的实习中，自己在学校独自一个人学习，不像其他同学都是有伴成对出去的。这时候自律是非常重要的，每天规律地生活，按时看视频课程，看课程详细笔记，最后通过编程练习来检验自己的学习情况，这需要耐得住寂寞。在以后的研究生阶段的学习过程中，只有越来越深入某一领域，而一起学习的人只会越来越少。心理需要更强大，忍受学习时的孤独，也享受真正做出属于自己的东西时候的自豪，这才是一个正循环，才能促进自己对社会国家作出贡献。

正视自己，做好人生规划

本次生产实习让我对自己有了一个更加清醒的认识，我也在思考我是不是一名合格的郎世俊实验班学生。我的自身还存在很多不足，我还有很多东西需要学习，我要给自己制定一个明确的目标，不然的话我靠什么实现自己人生的价值。有一个明确的目标，清楚自己想要什么，有一个明确的人生规划，这才是一名合格的大学生应有的精神风貌。

因此，我决定在以后的学习生活中，要正视自己的能力，并且要克骄克躁，注重学习的过程，而不要只注重结果，任何事情都没有一蹴而就的，唯有自始至终初心不变的坚持才能成就自己。另外，给自己一个小规划，我是就业，还是读研，读研的话我读哪个方面的研究生呢？经常思考一下自己有没有更好的选择。一生只有一次，不可随遇而安，必须要努力争取，给自己定下小目标，通过不断实现小目标为实现大目标做好准备。有规划、有目标、再加上自己脚踏实地的努力，我相信自己定会取得让自己满意的结果。

3.3 致谢

真诚感谢学校给我们这次自主实习的机会，让我们有机会学习感兴趣的领域，让我们在学好知识的同时增长见识。这次实习对我们以后的学习和生活真的意义重大。感谢老师们为这次实习安排做的不懈努力，感谢李晶皎老师战略性的指导和建议，这将是我日后最美好的一段回忆！

4. 参考文献

[1] Deng J , Dong W , Socher R , et al. ImageNet: a Large-Scale Hierarchical Image Database[C]// 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA. IEEE, 2009.

[2] Justin Johnson. Backpropagation for a Linear Layer[N]. 19 April 2017.

[3] Srivastava N , Hinton G , Krizhevsky A , et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting[J]. Journal of Machine Learning Research, 2014, 15(1):1929-1958.

[4] Ioffe S , Szegedy C . Batch normalization: accelerating deep network training by reducing internal covariate shift[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2015.

[5] Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., Fergus, R.: Regularization of neural networks using dropconnect. In: International conference on machine learning. pp.1058{1066 (2013)

[6] Graham B . Fractional Max-Pooling[J]. Eprint Arxiv, 2014.

[7] Devries T , Taylor G W . Improved Regularization of Convolutional Neural Networks with Cutout[J]. 2017.

[8] Zhang H , Cisse M , Dauphin Y N , et al. mixup: Beyond Empirical Risk Minimization[J]. 2017.