第2章
培养对数据的敏锐观察力

人们所熟知的科学家都非常重视对观察力的培养和要求，如俄国化学家门捷列夫对观察有这样的理解：

> 科学的原理起源于实验的世界和观察的领域，观察是第一步，没有观察就没有接踵而来的前进。
>
> ——门捷列夫

英国生物学家达尔文把他的成就归功于他“在他人之上”的观察力，他这样描述自己的超人观察力：

> 既没有突出的理解力，也没有过人的机智，只在观察那些稍纵即逝的事物并对其进行精细观察的能力上，我可能在他人之上。
>
> ——达尔文

从心理学的角度，可以把观察定义为有目的、有计划、比较持久的知觉。观察是一个人积极的思维活动，是稳定有意识的注意过程，该过程中借助过去的经验来组织知觉，所以说观察是一个系统的、持久的知觉。观察力即为观察的能力，观察力的最可贵品质是从平常的现象中发现不平常的东西，从表面上貌似无关的东西发现相似点或因果关系，人们在观察能力水平上存在很大的个体差异，凡是在研究事业上成就较高的人，他们在观察能力水平上都比常人要高。

AI 算法工程师从事的是具有研究性质的工作，与科学研究存在很多相似性。学会观

察数据是 AI 算法工程师的基本素质，若要从纷繁复杂的数据中寻找算法设计的思路和灵感，则需要具备对数据的敏锐观察力。有些算法工程师偏好对算法理论的研究，而不重视对数据的观察和分析，导致在算法研发上难有创新，甚至为此在算法设计上走了很多弯路。

当前的 AI 算法技术水平还未发展到可以适应所有不同 AI 场景数据的程度，所有的 AI 产品设计都需要根据数据的特点选择合适的 AI 算法，并利用数据分析结果和数据训练的方法调整算法参数。虽然我们会把 AI 算法的泛化能力作为一个重要的设计目标，但这个泛化能力仅局限在一种 AI 场景方向上的不同数据。即使在同一个 AI 场景中，为了提升 AI 产品可用性，也会有很多的约束条件，如每天上下班时很多人需要面对的人脸考勤机，当你利用人脸进行上下班打卡时会有不少限制，如人脸需要正对着考勤机，人脸与考勤机之间需要保持一定的距离，头发不可盖住眉毛，不可戴墨镜，面部不带表情等。这些约束便是对所要处理的数据的特殊要求，若无法满足这些要求，你会发现 AI 机器变得无比“智障”。因此，脱离了数据分析，脱离了在数据分析过程中对数据的敏锐观察，难以想象可以把 AI 算法做好。

接下来，我们将从多方面探讨如何培养对数据的敏锐观察力。

2.1　心中有“数”

在 AI 算法研发过程中离不开数据的支撑，首先需要根据 AI 场景数据进行技术可行性分析，以及对数据进行更深入的分析，根据数据的特点和规律选择合适的算法；然后在算法设计过程中把数据分为测试集和训练集，利用测试集和训练集对算法进行调优；最后在评估或验收 AI 算法时，收集新的数据检验算法的泛化能力（又称推广能力）。AI 算法研发过程就是一个不断跟数据打交道的过程，也是一个对数据逐渐深入了解的过程。为此，我们必须做到心中有“数”。

如果你是从软件应用系统研发工作转到 AI 算法研发工作，则需要转变工作的思路和方法，需要分清楚应用系统研发和 AI 算法研发的特点，如表 2.1 所示。

从表 2.1 中可以看出，AI 算法研发相较应用系统研发显得更为单纯，它主要是一个算

法技术实现和改进的过程，而应用系统研发需要考虑的问题较多，在满足功能需求的前提下，还需要考虑系统的美观性、易用性、可维护性、稳定性等。但是单纯并不意味着简单，相反地，AI 算法研发是一个复杂的、较高难度的代码设计工作，并且在算法细化设计和优化过程中需要做很多的创新，特别是在对数据的观察和分析上，更能够考验一个人的创新能力水平。

表 2.1　两种软件系统研发的比较

应用系统研发	AI 算法研发
较多采用敏捷开发的模式，系统迭代频繁	较多采用瀑布开发模式或螺旋开发模式，采用较为稳重的推进模式
客户较容易说出他想要的功能，需求一般较为明确	客户对 AI 技术不太了解，大多情况下无法明确给出需求（算法性能目标）
较注重软件系统设计的易用性	较注重 AI 算法的可用性和可靠性
研发周期一般都较短，较少有一年以上的项目研发周期	研发周期一般都较长，两三年以上的研发时间非常常见
测试和验收较为简单，主要是功能性测试和稳定性测试	测试和验收较为复杂，需要有量化指标，而且还要面对评价数据不稳定性的难题
工程师的代码产出率较高，主要工作时间是在为功能的设计和调整编写代码	工程师的代码产出率较低，主要工作时间是在利用数据进行算法的优化
开发过程更侧重界面美观设计、流程和功能分析	开发过程更侧重数据分析和技术可行性验证
软件系统的维护工作主要是为了适应系统环境和客户需求的变更	AI 算法的维护工作主要是为了提升算法的可靠性和适应性

如果你没有 AI 算法研发经验，或者从未认真思考如何面对 AI 场景数据，那么以下这些问题将有助于你快速了解 AI 场景数据的相关内容。

1. 数据从何而来

例如对于图像数据，采集图像数据的方式非常多，有扫描仪扫描、手机拍摄、监控设备抓拍、高拍仪抓拍、显微电子设备图像生成等，这些不同的图像数据获取方式产生了不同的数据特点，导致对算法设计有不同的技术要求。表 2.2 描述的是两种不同方式获取的图像数据的特点。

表 2.2　两种图像获取方式的比较

图像采集方式	主要特点
扫描仪扫描	图像清晰，分辨率高，图像细节清晰可见，不受外界环境干扰
手机拍摄	受外界环境影响大，有反光、抖动、光照不均、背景复杂等现象，图像模糊，不同手机还存在拍照图像质量不一样的情况

在工业领域，数据的来源更是多样化，并且数据还不能同时得到，如环境温湿度数据可以实时获取，但是成分化验数据则需要通过采样化验后得到。对于这种无法同步得到的数据，如何去组织不同来源的数据是 AI 算法系统设计首先要解决的问题。

2. 数据可靠吗

有些数据是人工录入的，并且没有经过严格的检查，数据难免存在人工录入的失误；有些数据是从数据库直接导出来的，可能存在导出的错误；有些数据存在中间过程的转换，可能存在转换过程中数据丢失或不完整的现象；有些数据可能因采集数据的设备出现了异常，导致数据无法真实反映情况。对于数据的可靠性是首先要确认和解决的问题，只有在确保数据可靠的前提下，才能开始后续的数据分析工作。

如表 2.3 所示的数据是在某个室内环境中从 4 个 CO_2 传感器获取的，通过数据对比分析发现第一个传感器得到的数据与其他 3 个传感器数据存在明显的差异。室内是连通的，4 个传感器相隔的距离都不远，可以确定是传感器出了问题，最后发现是因为第一个传感器没有调校而导致测量得到的数据偏差较大。

表 2.3　4 个 CO_2 传感器采集的数据

CO_2 测点 1（ppm）	CO_2 测点 2（ppm）	CO_2 测点 3（ppm）	CO_2 测点 4（ppm）
1569	803	803	786
1586	799	799	771
1619	792	792	779
1603	812	812	778
1616	810	810	784
1621	801	801	789

续表

CO_2 测点 1（ppm）	CO_2 测点 2（ppm）	CO_2 测点 3（ppm）	CO_2 测点 4（ppm）
1605	810	810	784
1591	792	792	786
1608	791	791	781
1604	809	809	787
1589	801	801	787
1585	802	802	785
1589	818	818	773
1603	790	790	777
1593	789	789	777
1618	783	783	774
1603	794	794	773

“尽信数，不如无数”，在算法设计之前观察和分析数据，尽可能地发现数据中的问题，可以大大降低后续的研发风险。

3. 数据量足够用于分析吗

数据分析主要是以概率统计为理论基础，如果是少量的数据则统计分析的数据与真实情况可能存在较大的偏差。例如可以从抛硬币的统计过程发现数据量大小的重要性，表 2.4 是试验得到的数据。

表 2.4　不同次数下的抛硬币概率统计数据

抛硬币/次	10	20	30	40	50	60	70	80
字面向上的概率/%	70	65	60	55	48	51	52	51

对于抛硬币过程中字面向上的概率，从理论上很容易理解它的概率应当是 50%，但从上面的试验可以发现，当抛硬币的次数较少时，统计得到的概率与 50%偏离很大，且次数越少偏离越大。

从这个试验可以发现，当我们在做数据统计分析时，如果数据量不够，则分析得到的数据准确性或可信度可能较低，那么多少数据量较合适呢？根据概率理论，数据量是越多越好，数据越多则越逼近真实情况，但是，现实中并不是我们想要多少数据就有多少数据，一个原因是现实条件的限制无法得到足够多的数据，另一个原因是数据量越大则需要越多的计算资源来支持，所以我们需要折中考虑。这里有一个较好的解决方案，首先分析 AI 场景可能存在的各种环境条件和状态，然后寻找各个条件和状态下的数据，以保证数据有较高的覆盖率（类似测试用例的设计）。至于每种条件和状态下多少数据量较合适，需要依据现实条件、所选择的 AI 算法和场景目标来确定。如用于测试的图像数量，如果识别准确率要求达到 1%，则需要至少 1000 张的图像来测试；如果是 0.1%精度，则需要至少 10 000 张的图像来测试。精度越高，需要测试的图像越多。对于训练图像的数量，不同的算法需要数据量有所不同，如对于 SVM 算法，较少量的图像（往往需要 500～1000 张的图像）就可以训练出较好的分类算法。但是对于深度学习，如果是预训练，则所需要训练图像数量是 SVM 算法的千倍以上，如果是微调（fine tuning），也需要数倍以上的图像数量。

4. 数据的格式是什么

数据的格式是指数据的组织方式，不同的数据有不同的组织方式。如汉字字符串，有不同的编码组织方式，如 GB2312、GBK、UCS2 等编码形式，甚至还会碰到三字节形式的、让人觉得很奇怪的 UTF8 编码。对于 UCS2 编码还有 big-endian 和 little-endian 两种字节顺序，当你处理中文字符出现乱码时，几乎都是因为处理数据时忽视了编码问题。特别是对于二进制方式存储的数据，更需要有一份数据格式的说明文档，否则可能需要大量的时间去研究和破解。如对于图像和视频数据，有各种各样的压缩格式，虽然有丰富的开源工具来直接读取，但在工业和医疗领域中可能存在私有格式的图像数据，这类数据可能无法用常用的工具读取图像数据而需要用专用的工具才能读取和分析其中的数据。

当经过算法或数据分析工具处理后的数据出现乱码，如中文字符变成了一堆让人看不懂的乱码、显示的图像出现错乱、输出的数据离正常值偏差太大，很有可能是数据格式问题导致的。

5. 读取数据的工具是什么

如何把数据导入数据分析工具？有些需要专用的数据解析工具，有些可能还需要编写特殊的程序进行读取。只有把数据读进来，才能做全面的数据分析。

对于列表类型的数据，如果你不想花心思通过编程来分析数据，可以用微软 Excel 文档工具来处理。导入数据的方法非常简单，把每列的数据用逗号分隔，然后把文件的扩展名更改为 csv，这时就可以用 Excel 打开该文件，在 Excel 窗口中可以发现每列数据被整齐分隔。现在的 Excel 功能非常强大，能够支持很多函数的使用，可以实现大部分的数据分析工作，并且都是可视化的操作，不需要编程，非常简单和高效。

6. 数据整理需要做哪些工作

数据整理的工作主要是为了得到可靠的数据，对于原始数据需要去掉其中没用的或异常的数据，有时为了算法系统中某个模块的设计需要，还需要对数据进行变换，如对于图像数据，可能需要对图像做旋转和切割的工作。

AI 算法设计中一般都会有一个数据预处理模块，该模块的主要功能就是对输入数据进行过滤、整理、变换，以降低后续模块数据处理的压力。在未进行 AI 算法设计前，通过数据整理的工作和分析可以预见将来在算法设计时所需要做的数据预处理功能。

7. 对每个维度的数据是否有充分的说明

对于数据的每个维度都需要去理解它，需要得到数据的充分说明，特别是在工业数据中，数据的每个维度代表了不同测点，具有不同的含义，为此，还需要了解相关的工艺知识，否则分析非常困难，甚至无从下手。

对数据进行归一化处理是解决数据不同维度差异的很好解决办法。如何进行归一化处理，需要详细了解每一维度的数据表示方法、数据分布特点和数据范围，这些信息都可以从数据说明信息或数据分析结果中获取到。

8. 数据与算法设计的目标是否相关

首先要确保获取的数据是实际应用场景中采集的数据，如果场景较复杂，可能获取的数据并不是我们想要的数据，特别是对于工业生产数据，由于工序较多，数据分析之前需要结合算法设计目标确定得到的数据是否相关，增加不相关的数据会加大算法设计的难度，而漏掉相关的数据则会降低算法系统的可靠性和性能稳定性。

9. 数据有什么特点和规律

数据分析的主要目的是寻找数据的特点和规律，AI 算法难以实现通用性的原因之一在于，在不同的 AI 场景中，数据的特点和规律难以相同，需要针对数据特殊性进行算法改进和算法

模型参数调优。如果对数据的特点和规律不了解或有误解，则会存在巨大的设计风险。

有经验的 AI 算法工程师通过观察可以快速发现数据的特点和规律，如果经验不够，则需要通过一些数据分析的工具或者编程来了解数据的特点和规律，在第 3 章我们会重点探讨寻找数据特点和规律的方法。

10. 新项目的数据与以往所经历的相似 AI 场景的数据有什么不同

对于有经验的算法研发人员，若忽略现有数据与所熟悉数据的差异，很容易犯经验主义的错误。通过发现数据的差异，便可以找出算法需要改进的地方，甚至可以预防掉入设计错误的陷阱。

数据在不同应用场景或实施项目中一定会有或大或小的差异性，为了解决这些差异性，通常需要做一些定制化的修改和新功能设计，这也是有些 AI 科技公司以项目形式为主要推广模式的原因。

11. 根据现有数据选择哪种算法实现较合理

针对不同的数据量和数据特点，可以选择不同的算法进行设计。在算法选择上很难选出效果最好的算法，需要因地制宜，有些算法需要数据量较少，有些算法需要数据量较多，有些算法实现起来较为简单，有些算法实现起来较为复杂且研发周期长。算法实现的难易程度还跟个人的自身能力和熟悉程度有关，原则上尽量选择自己熟悉的算法技术，避免跟风和求新，否则会增添很多设计上的风险。

其实，每一种 AI 算法都有其局限性，没有一种算法可以完全解决问题，当一种算法无法解决全部问题时，不妨考虑运用其他算法进行弥补。对于 AI 算法技术采用系统化的设计方法，运用工程的思想去解决问题，这样便可以广开思路，使得 AI 算法技术最终能够落地。

通过上面这些问题的分析将有助于我们快速了解数据，做到心中有“数”，为接下来的算法研发工作打下基础，甚至可以避免一些不必要的研发风险。

2.2　数据理解力

面对数据，不同的人会得到不同的理解，这与个人的认知水平和观察的角度有关。对

于欠缺算法研发经验的人，要快速理解 AI 数据是个较难的任务，这种情况下需要多些耐心，并持续观察和思考，在算法设计过程中逐渐对数据有较深的理解。数据是复杂和多变的，在不同的 AI 场景下，数据都会有各自的特点，提升数据理解力需要一定的经验积累，这是一个长期的锻炼过程，所以培养一名优秀的 AI 算法工程师并非易事，相对于应用系统研发工程师，AI 算法工程师的成长更难，需要更多的时间。

可以从如下几个方面考察 AI 算法工程师对数据的理解能力。

1. 对数据在高维空间中的分布是否具有想象力

人类习惯于三维空间的思维，当超过三维时，便让人难以想象和理解，如图 2.1 所示是四维空间中的正立方体。可是，AI 算法要处理的数据很少在三维以下，这需要对高维数据具备一定的思维能力，能够想象数据在高维空间中的分布特点。要做到这一点还需要借助一定的分析工具。

图 2.1　四维空间中的正立方体

人工智能算法经常提到“维数灾难”这个名词，维数灾难是指在高维空间中遇到的难以克服的难题。在高维空间中，数据的分布变得非常的稀疏，如果我们用 Euclidean 距离、Manhattan 距离等方法来描述不同数据之间的差异性，会发现存在直觉上无法理解的问题，如在图像识别测试过程中，会发现明明看起来很相似的两个图像计算得到的特征距离很大，而看起来差异很大的图像特征距离却很小。如果直接用 Euclidean 距离计算两个特征的相似度，在大部分情况下都难以准确反映数据之间的差异，导致机器学习效果不理想。

2. 能否从多角度寻找数据的分布规律

如图 2.2 所示的山峰，从不同角度观察可以得到不同的认知。对于数据的分布规律，如果从不同的角度去观察，也会得到不同的规律。对于一幅图像，如果从色调的角度去分析，可以得到色调的种类；如果从亮度的角度去分析，可以得到图像的明暗特点；如果从边缘梯度的角度去分析，可以得到图像的清晰度特点，等等，尝试从不同角度分析可以找到更多的规律，从而有更多解决问题的方法。

图 2.2　横看成岭侧成峰

从理论上来说，某 AI 场景中的数据若要具备可预测性，则数据必须具备一定的分布规律，否则无法被预测或分类。数据往往具有多面性，在某个方向找不出规律，但在另一个方向可能发现其规律，这需要耐心地观察和分析。

3. 能否找出不同状态下的数据变化规律

数据发生变化有其内在驱动因素，在不同的状态下，数据展现不同的变化规律。AI 算法系统中常设计一个有限状态机处理各种不同状态下的数据，并通过找出各种状态之间的转移规律控制 AI 算法系统中数据流的运转，运用有限状态机可以较好地完善 AI 系统要实现的功能。有限状态机图例如图 2.3 所示。

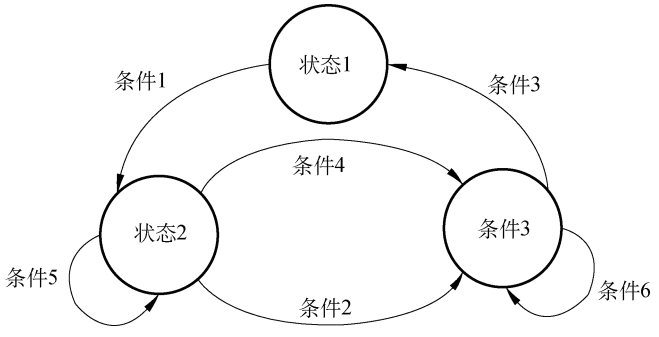


图 2.3　有限状态机图例

数据变化分稳态变化和非稳态变化，稳态变化的数据一般呈高斯分布，在第 3 章中我们将详细讨论高斯分布的特点和数学规律。非稳态变化有多种数据分布形式，最常见的是

如图 2.4 所示的数据变化规律——Sigmoid 函数。

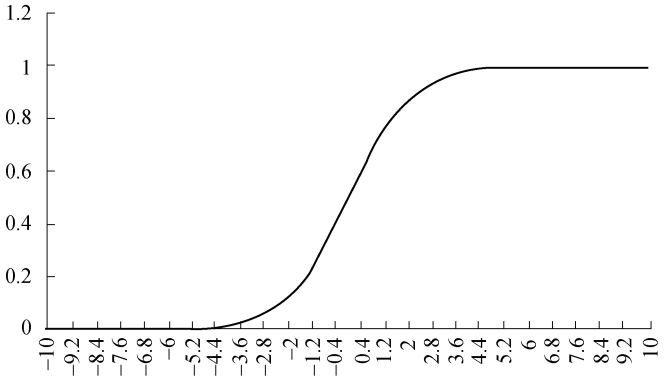


图 2.4　Sigmoid 函数图像

Sigmoid 函数又称 Logistic 函数，公式形式如下。

$$\text{Sigmoid}(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{2.1}$$

Sigmoid 函数的数据变化非常的自然和平滑，而且对 Sigmoid 函数进行求导非常简便，$S(x)$的导数可以表示为 $S(x)(1-S(x))$，　在神经网络类型的机器学习中 Sigmoid 函数还被应用于激活函数。

存在多种数据的变化类似 Sigmoid 函数所表现的数据变化特点，例如，人口增长，通常初始增长较慢，然后随着时间的推移逐渐加速，最终趋于饱和；产品市场饱和度，新产品在市场上的接受程度通常经历一个缓慢的增长阶段，然后随着市场渗透率的提高，增长速度逐渐加快，最终趋于饱和；等等。所以，利用 Sigmoid 函数，或对该函数稍加变形，即可模拟现实中的许多数据变化，建立各种用于测试的仿真环境。

4. 能否对数据进行转化以简化分析问题的复杂性

很多情况下我们需要对数据进行适当转化以简化问题的复杂性，如图 2.5 所示，可以通过图像增强得到更易于识别的图像。图像识别算法中常常对彩色图像进行灰度转化，然后进行二值转化，通过这样的方式可以降低图像数据的维度，从而降低算法设计的难度。当然，现在的卷积深度学习（CNN）技术已不需要这样做，但是在很多情况下，对数据进行转化后确实可以收获意想不到的效果。

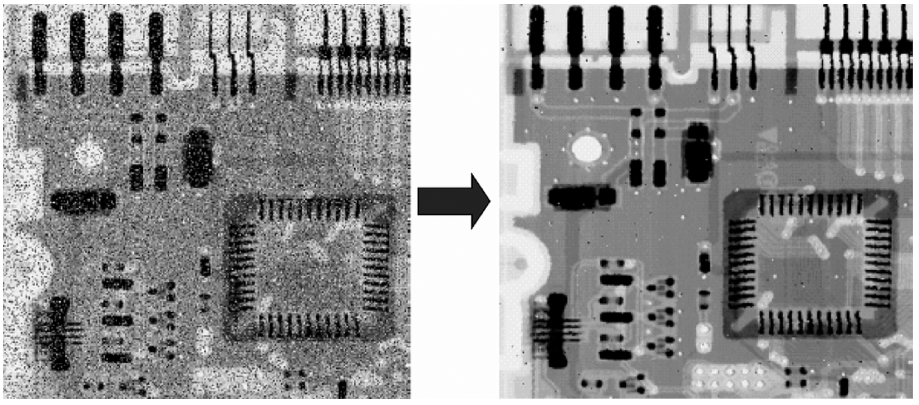


图 2.5　通过图像增强得到更易于识别的图像[①]

5. 现有数据对算法模型中的参数变化会有什么影响

现有数据如何影响算法模型中的参数，目前在深度学习技术中是个难题，因为它几乎是一个黑盒，而且其中的参数数量可能达到千万甚至上亿的级别，所以深度学习是一个难以分析内在原理的技术。解决问题的办法是多次训练和测试来寻找最佳参数组合。对于模型参数较少的机器学习算法可以较为直接地发现参数变化规律，从模型参数的变化找出训练数据的问题，通过解决训练数据的问题来提升训练效果，如纠正训练数据中标注有误的样本、剔除 AI 场景中不可能出现的训练样本、增加不同状态下的样本数据等。

6. 能否从测试结果评估数据中寻找算法的问题和训练数据的问题

算法测试是一个无法逃避的环节。为了从测试反馈的数据中发现问题并改进算法，首先需要非常熟悉数据，其次要能够判断哪些问题是因为代码设计错误，哪些问题是因为算法技术方案的错误。一般在算法程序设计过程中，初始阶段大部分工作是在解决代码设计错误，中期通过对数据的理解改进算法，后期需要增加算法系统的复杂度来提高算法可靠性，其规律如图 2.6 所示。

作为 AI 算法研发工程师，需要有意识地提升对数据的理解能力，并且认识到理解数据需要扎实的数据分析工作，在第 3 章我们将会深入讨论数据的分析方法。

① Akyol G. What is Image Enhancement? [EB/OL]. (2023-01-14) [2023-03-22], https://medium.com/@gokcenazakyol/what-is-image-enhancement-image-processing-3-32a813087e0a.

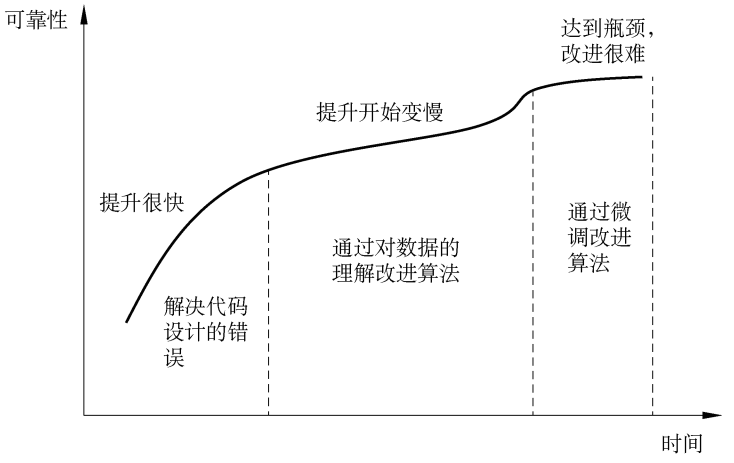


图 2.6　算法改进过程规律

2.3　实践经验积累

AI 算法研发不仅理论性很强，实践性也很强，需要具备一定的工程实践能力。没有经历过至少一两个 AI 算法项目的实践，就不会对算法理论有更深的认识，更无法体会数据的复杂性。如何在实践过程中快速提升对数据的理解力，可以从以下三个方面做起。

1. 保持开放的心态

如果你是在一个团队中做研究，心态开放非常重要，要乐于与同事做技术交流，虚心向同事学习和请教，善于取长补短，这样不仅可以提升工作效率，更可获得解决问题的不同灵感。假如你不是在一个团队中工作（单干），可以通过各种机会与外界交流，如论坛、博客、讲座、研讨会等。我们身处在信息高度发达的时代，人与人之间的沟通和交流变得非常容易。人与人之间的关系是相互的，当你把你的想法和技巧告诉别人时，别人也会回馈给你他的想法和技巧。不要担心在交流过程中可能出现冲突，思想经过“碰撞”才能出现“火花”。当你秉持开放的心态做研究时，会发现自己的进步很快。

2. 多做反思和总结

学习、实践和总结是一个闭环，在实践中会碰到很多问题，甚至会走很多弯路，这些都需要我们去深入反思并总结解决问题的方法和规律。在 IT 行业中，大部分人的工作都比较紧张，很多人为了赶项目而“疲于奔命”。AI 算法研发是一个研究性很强的工作，应当留给自己更多的时间去思考，多反思和总结，这样才可以提升工作效率。工作效率提升了，就会有更多的时间去学习和思考，形成一个良性循环，否则，容易陷入工作难以收尾和不断处理各种问题的泥潭中。

完成算法设计文档是一个反思和总结研发工作的好方法，通过文档的编写，不仅可以加深对算法的认识，还可以通过文字的描述对算法设计进行抽象和总结，提升个人的技术沟通能力，增加技术交流的机会。算法设计文档模板如表 2.5 所示。

表 2.5　算法设计文档模板

××算法设计

一、算法功能目标概述

1. 应用场景

2. 性能要求

3. 算法输入数据的限制

4. 算法应用软硬件限制

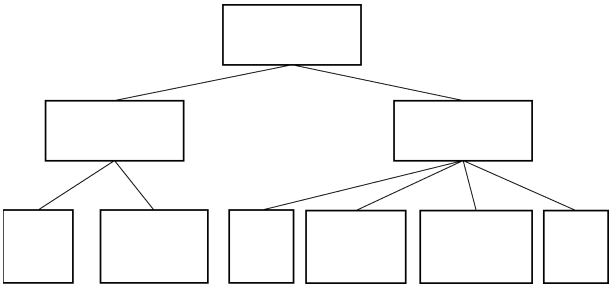
二、算法输入数据分析

……

三、算法实现原理和设计方法概述

……

四、算法模块结构图



续表

五、关键数据结构和数据文件描述

……

六、算法实现关键过程

1. 主体函数名称定义

2. 算法实现流程图

七、代码实现位置

1. 代码整体实现文件夹路径

2. 与该算法有关的代码文件

八、算法测试方法

……

九、算法应用需注意的问题

……

十、算法性能现状

……

十一、算法改进日志记录

……

3. 坚持学习和跟踪前沿技术

坚持学习是研发人员应当具备的优秀品质，理论和技术每天都在更新和变化，若没有及时跟进，则很容易成为“井底之蛙”。学习可以是多方面的，人工智能技术涉及的领域非常广泛，如计算机科学、数学理论、生命科学、认知心理学、人文社会科学、科学哲学等，这些都是需要去了解的信息。对于 AI 的前沿理论和技术更需要去了解，这样才能更好地确立和规划研发方向，才能保证研发的算法具有先进性和优越性。

2.4 数据的复杂性

设计算法时不要忽视数据的复杂性，只有想不到的情况，没有不可能存在的情况。如

图 2.7 所示的车辆检测算法训练图像，存在不同视角、不同车型、不同车身颜色等情况，针对这些情况，车辆检测算法需要的训练图像数量达到十万以上才能达到可用的效果。

图 2.7　车辆检测算法训练图像

OCR 是一种文字识别技术，即把图像中的文字转换成可以编辑处理的字符信息，20 世纪六七十年代开始有这方面的相关技术研究，主要是针对数字和字母的识别研究，对汉字的识别在 20 世纪 80 年代才有较好的突破。对于汉字识别，最开始的研究想从笔画上进行结构识别，通过笔画的种类和位置识别汉字，但在笔画提取时遇到困难（这里要区分联机识别和脱机识别，联机识别可以跟踪笔迹运动过程，所以很容易提取笔画，如图 2.8 所示的手写输入法便主要是通过笔画进行识别的，但它不属于 OCR 技术范围，如图 2.9 所示的过程为 OCR 技术），如断笔、粘糊、笔画方向判断问题等，如图 2.10 所示。对于人脑确实很容易识别笔画，但对计算机确实太难，人们低估了图像数据的复杂性。后来采用统计识别的方法绕过了笔画提取难题，终于在汉字识别上取得了较好的效果。20 世纪 90 年代又出现了卷积神经网络技术，该技术能够很好地适应图像旋转和变形，更好地解决了 OCR 技术问题，对于手写字符具有较好的适应性。回顾整个 OCR 技术发展历程，可以发现在 OCR 技术变化过程中为了解决图像数据的各种复杂性而创新了多种新技术。

图 2.8　联机手写输入（联机字符识别）

图 2.9　名片拍照识别（脱机字符识别，OCR）

断笔情况　　粘糊情况　　笔画类型和方向都一样

图 2.10　字符识别需要解决的难题

对于数据的复杂性有一个认识过程，刚开始特别是经验欠缺时，对数据的复杂性意识往往不够，随着研发的深入，数据的复杂性问题便会逐渐显现出来，AI 软件项目的研发无法按期交付往往受这个因素影响。

当前人工智能技术已渗透到各个行业，人工智能的项目机会非常多，但现实中能做成功的机会并不多，其中不仅是技术水平问题，更多是数据的问题。如有些企业连信息化的建设尚未完成就想一步跨入智慧化，这样的企业保存的数据不仅不足以支撑机器学习所要的数据量，而且在人工录入的数据中还存在大量的错误，最终导致项目无法按期完成甚至失败。

在企业的商务推广中 AI 算法研发工程师的话语权一般较低，经常是项目可行性论证还不够就匆匆开始研发，还有就是有些算法研发工程师对技术过于乐观，凭借已有的经验和客户对数据的说明就确定了技术方案，结果当看到实际数据时往往就“傻眼”了，才发现原来的技术方案不可行，甚至根本做不了。

在没看到数据之前千万不要妄下定论，有如下四点原因。

（1）数据是复杂的，客户通常很难把数据描述清楚，除非客户自己能够做充分的数据分析。

（2）要应用的数据一般是多维的，哪些维度有用，哪些维度没有用，在行业专家分析后，还需要通过数据去验证。当维度多时，专家也会出现疏漏。

（3）不要低估数据的复杂性，特别是图像数据，只有你想不到的情况，没有不会出现的情况。

（4）根据数据的特点选择合适的产品实现策略非常重要，不要全依赖算法技术，如对于图像识别场景，可以通过改进硬件技术提升图像数据质量或者增加图像采集的约束条件，从而降低算法实现难度。

2.5　培养创新意识

当我们明白了 AI 算法所面对数据的复杂性和当前的 AI 算法技术水平，就更能体会创新是多么的重要，一位没有创新意识的 AI 算法研发工程师称不上研发工程师，甚至连应用开发工程师都可能无法胜任。创新并不是一定要想出一个新的算法思想，AI 算法实践中可以有很多微创新，有人把微创新称为工程技巧。不要小看这些微创新，它们积累到一定程度就是大创新。“实践出真知”，在实践过程中需要解决很多的问题，在解决问题的过程中便可以有很多的创新，你可能“灵光一现”，便发现了一个新的算法设计思路，甚至可能是

一个新的 AI 算法思想。

一个人的创新能力更能体现出他的研发能力，在研发工作过程中要有意识地开拓思路和提升创新能力水平，注意以下几点有助于你提升创新能力。

1. 要有独立思考的能力

现在是信息高度发达的社会，我们每天可以看到各种各样的信息，若没有独立思考的能力，便会被各种信息所迷惑，甚至会变得不知所措。作为 AI 算法领域的实践者，我们需要具备清醒的头脑，对当前的技术水平要有清醒的认识，不盲从，不迷信权威。在了解技术的过程中，若发现问题要敢于批判和反思。对于需要应用的技术要追根究底，摸清和彻底理解其技术原理。在寻找或搜索技术思路之前，多想想自己能够有什么思路，不要习惯性地使用开源的技术，开源技术虽然来得快和容易，但封闭了自己的创新能力。

2. 培养广泛的兴趣

人工智能技术是涉及面很广的科学，如图 2.11 所示，数学是其基础，但能够让你突发灵感的往往不是数学，我们可以从生命科学中寻找生物智能的原理，可以从科学哲学中寻找思维的方法，可以从人文科学中寻找人类解决问题的智慧，可以从心理学中寻找人的认知方法，甚至也可以在美学中寻找对美的感受和理解，等等。让自己的兴趣广泛些，可以让你对外界事物保持好奇心，开拓自己的视野，让你的思维充满活力，让你更具有创新能力。

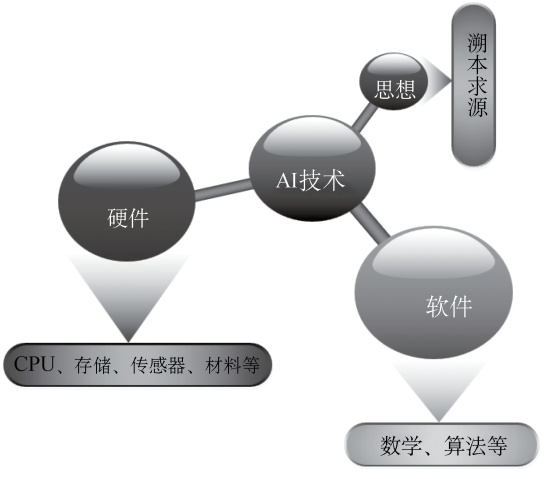


图 2.11　AI 技术所涉及的领域

3. 坚持学习

若抛弃了学习，仅一味地想创新，则只能让你的思路成为“无源之水”。创新是一个厚积薄发的过程，学习是创新的基础，并且“学无止境”，通过学习汲取各种知识的营养，才能让自己更有能力去创新。

4. 要有耐心和毅力

有些问题的解决需要较长的时间去思考，持续深入地思考才能找到解决问题的新办法。有问题并不可怕，可怕的是我们没有耐心和毅力去解决问题，如对于数据的观察，是否愿意耐着性子反复去琢磨？数据是抽象的，更是枯燥的，但当你每次去观察和分析数据时，总是会有新的发现，每次新的发现往往会引导你找到解决问题的新思路。AI 算法研发是一个长久的过程，少则半年，多则五六年，甚至更长的时间，在这漫长的过程中需要一个又一个的创新去突破，若没有耐心和毅力则很难坚持下来。

5. 劳逸结合

做研究常会有这样的感觉，无心插柳柳成荫，有心栽花花不开。能否创新，常常给人感觉是可遇而不可求。当你迷茫、困惑时，不妨放一放，劳逸结合，出去走走，或转移下注意力，灵感也许会在你不经意间闪现。紧张繁忙的工作一般很难有创新，因为在忙碌中很难有深入思考的时间。让自己从繁琐的事务中解脱出来，为自己创造一个易于创新的环境非常重要。很多高科技企业为研究人员安排一个安静、宽松的工作环境也正是基于这样的原因。

2.6　两种思维模式

美国著名的心理学家 Daniel Kahneman 编著的一本书名为《思考，快与慢》，该书从各个角度多方位地对人的思考模式做了深入探讨，他把人的思考模式分为两种，人脑中有两套系统分别执行两种模式的思考[①]。

① Kahneman D. 思考，快与慢[M]. 胡晓姣，李爱民，何梦莹，译. 北京：中信出版社，2012：5.

（1）系统一的运行是无意识且快速的，不怎么费脑力，没有感觉，完全处于自主控制状态。

（2）系统二将注意力转移到需要费脑力的大脑活动上，例如复杂的运算。系统 2 的运行通常与行为、选择和专注等主观体验相关联。

如图 2.12 所示，系统一模式即为所谓的“快思考”，系统二为“慢思考”。快思考是我们最常用的思考方式，依赖情感、记忆和经验迅速做出判断，使我们能够迅速对眼前的情况做出反应，我们经常所说的直觉和无意识的思考就是快思考方式。快思考很容易使我们犯错误，它固守“眼见即为实”的原则，任由损失厌恶（一种避重就轻的心理）和乐观偏见之类的错觉引导我们做出错误的决策。有意识的慢思考则通过调动注意力来分析和解决问题，并相应做出决策，这个过程虽然比较慢，但不容易出错。

图 2.12　两种思维方式

我们的大脑其实很懒，很容易习惯性地做快思考，就如有些人做事情喜欢跟着感觉走，这种人一般属于乐观派，是最容易犯错误的人。做事严谨认真的人，往往会做更多的慢思考，把风险和问题考虑充分，喜欢“先谋而后动”，这种人往往给人较为稳重的感觉。在这里，我们并不认为仅凭直觉和经验做判断一定是错误的，相反，在大多情况下，它非常有用，特别是在非常紧急的情况下，直觉和经验显得非常重要，如在我们开车时，需要不断地做出快速的反应和判断，这时就需要我们做很多的快思考。

AI 算法研发工程师需要对数据具备敏锐的观察力，这里的“敏锐”并不是指我们一定要快速做出判断，而是指对数据要有深刻和全面的分析能力。在绝大多数情况下，我们会

有充分的、较长时间的分析和思考，不需要做那种“脑筋急转弯”的思考。我们在数据分析过程中会犯很多错误，如忽略数据的差异性、数据的空间分布特性、异常数据对机器学习训练模型的影响等，产生这些错误的原因往往是对数据分析缺乏耐心和细心，喜欢凭直觉和经验做判断，没有养成慢思考的思维习惯。

科学家发现，慢思考和快思考对大脑皮层有不同的影响，习惯于或经常做慢思考的人，他的大脑皮层与其他脑神经的联系更深，思维更有广度和深度。所以，对于 AI 算法研发工程师，需要习惯性地多做深度思考，放慢工作节奏，不要急于下结论和做出决策，对数据要多做分析，养成深度思考的习惯，这样可以提升我们的思考水平，更可以提升我们对数据的观察力。

2.7　观察数据实现算法的案例

这里我们通过一个案例展示观察数据而得到的算法，这是一个非常简单、高效的算法。

2.7.1　算法设计需求——检测电路板中的污渍

图 2.13 中右侧圈出来的部分是合格的电路，左侧圈出来的部分是受污染的电路（污渍），需要把受污染的电路检测出来。

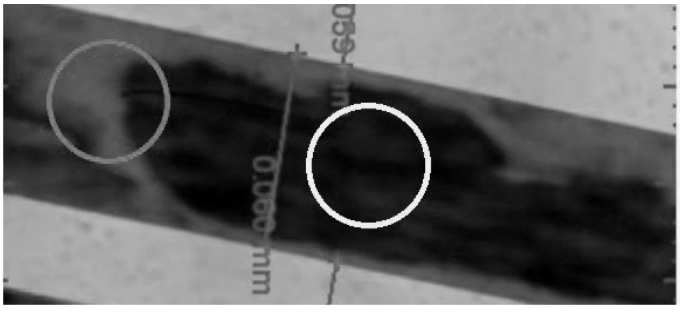


图 2.13　带瑕疵的电路板

2.7.2 观察数据

利用图像处理工具来观察数据的特点，这里我们用 CxImage 图像处理工具（见图 2.14）观察和处理图像数据，利用该工具在窗口状态栏中可以观察图像中每个点的位置和像素值。

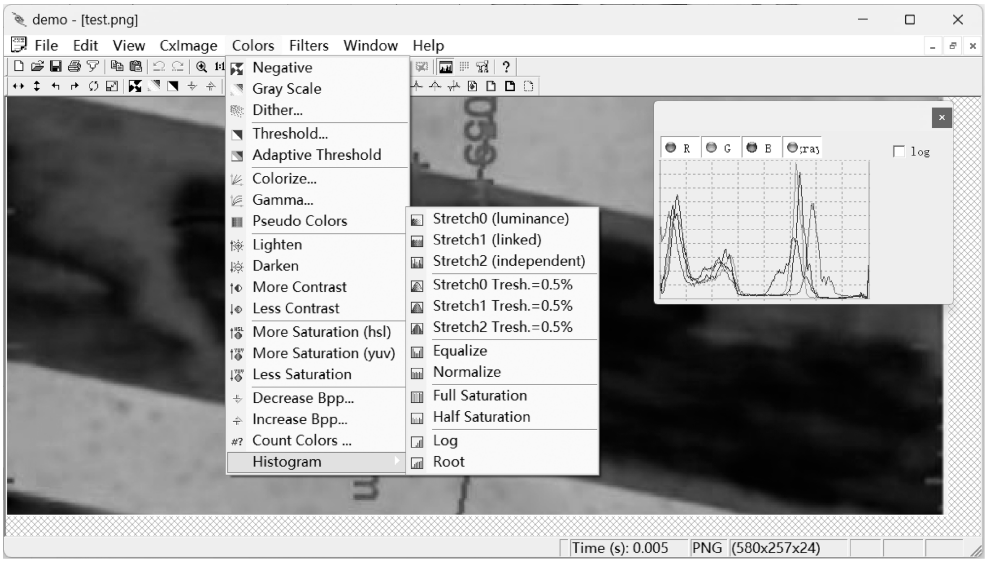


图 2.14　CxImage 图像处理工具界面

通过观察和分析可以发现带斑点的电路板图像数据有如下两个特点。

（1）黑色块图像的像素点红色分量一般都小于 60，而非黑色块的红色分量一般都大于 60。

（2）背景图像的像素点可以根据像素点红色分量是否大于 160 直接分离出来。

2.7.3 算法设计

该算法分为如下两步设计。

（1）利用彩色图像中的红色分量对图像进行灰度化处理。把红色分量大于 160 的像素点设置大小为 255 的灰度值（背景图像）。把红色分量小于 60 的像素点设置大小为 0 的灰

度值（非污渍图像）。把红色分量介于 60～160 的像素点设置大小为 150 的灰度值（污渍图像）。在 CxImage 中用 C++语言实现上面的过程，代码如下。

```
int height = image->GetHeight();
int width =  image->GetWidth();

//过程1：利用颜色值把背景灰度设置为255，把黑块的灰度值设置为0，把斑点的灰度值设置为150
for(y = 0; y < height; y++)
    for(x = 0; x < width; x++) {
        RGBQUAD color = image->GetPixelColor(x,y);

        //根据红色信息进行判断
        if(color.rgbRed < 60)
           image->SetPixelColor(x,y,RGB(0,0,0));
        else if(color.rgbRed > 160)
          image->SetPixelColor(x,y,RGB(255,255,255));
        else
            image->SetPixelColor(x,y,RGB(150,150,150));
    }
```

通过上面的方法可以把如图 2.15 所示的原始图像转换成如图 2.16 所示的灰度图。

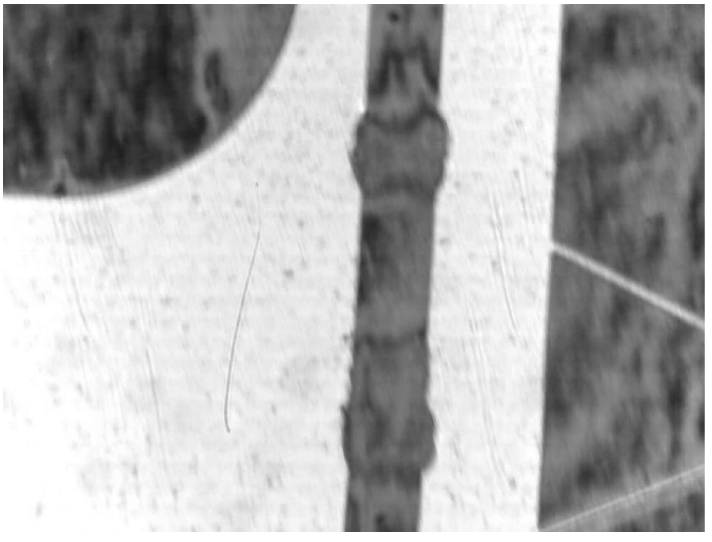


图 2.15　电路板原始图像

图 2.16　灰度转换后的图像

（2）利用连通域分析去除灰度图中的小块白点。通过队列数据结构可以快速实现像素点的连通域分析，在这里要做的是白色块（即灰度值为 255）的连通域分析，连通域分析（图像切割和去噪常用的方法）后即可得到所有互相分离的白色块，最后把较小的白色块去掉（把灰度值转换为 155）。代码如下。

```
//过程2：利用连通域分析，检测小白块（由于阈值设置的原因可能会出现小白块），把小白块设置为斑点

// 定义连通域结构体
typedef struct {
    int x;
    int y;
} Point;

#define CHECK_PIXEL(X,Y) {RGBQUAD PixelColor2 = image->GetPixelColor(X,Y);\
    if(PixelColor2.rgbRed == 255){ \
    image->SetPixelColor(X,Y,RGB(200,200,200));\
    queue[rear].x = X;queue[rear].y = Y;rear++;\
    blockPixels[PixelsCount].x = X;blockPixels[PixelsCount].y = Y;
PixelsCount++;\
```

```
    if(minX > X)minX = X;\
    if(maxX < X)maxX = X;\
    if(minY > Y)minY = Y;\
    if(maxY < Y)    maxY = Y;}}

// 定义连通域队列
Point* queue = NULL;
Point* blockPixels = NULL;
queue = (Point*)malloc(sizeof(Point)*1000*1000);
blockPixels = (Point*)malloc(sizeof(Point)*1000*1000);
for(y = 0; y < height; y++)
for(x = 0; x < width; x++){
   RGBQUAD PixelColor = image->GetPixelColor(x,y);
    if (PixelColor.rgbRed == 255 ) { // 如果是背景像素且未访问过

    int blockWidth,blockHeight;
        int front = 0;
        int rear = 0;
        int minX = width;
        int maxX = 0;
        int minY = height;
        int maxY = 0;
        int PixelsCount = 0;

       image->SetPixelColor(x,y,RGB(200,200,200)); // 标记为已访问

       queue[rear].x = x;
       queue[rear].y = y;
       rear++;
        blockPixels[0].x = x;
        blockPixels[0].y = y;
        PixelsCount++;


        // 使用广度优先搜索遍历连通域
        while (front < rear) {
           int x1,y1;
```

```
    Point current = queue[front];
    front++;

    // 检查当前像素的上下左右八个邻域像素
x1 = current.x - 1;
     y1 = current.y;
     if (x1 >= 0 ) {
           CHECK_PIXEL(x1,y1);
      }

     x1 = current.x -1;
     y1 = current.y -1;
     if (x1  >= 0 && y1 >= 0 ) {
       CHECK_PIXEL(x1,y1);
     }

     x1 = current.x -1;
     y1 = current.y +1;
     if (x1  >= 0 && y1 < height ) {
       CHECK_PIXEL(x1,y1);
     }

     x1 = current.x + 1;
     y1 = current.y;
     if (x1 < width)  {
       CHECK_PIXEL(x1,y1);
     }

     x1 = current.x + 1;
     y1 = current.y - 1;
     if (x1  < width && y1 >= 0 ) {
       CHECK_PIXEL(x1,y1);
      }

     x1 = current.x + 1;
     y1 = current.y +1;
     if (x1  < width && y1 < height ) {
```

```
            CHECK_PIXEL(x1,y1);
          }

          x1 = current.x;
          y1 = current.y -1;
          if (y1 >= 0 ) {
             CHECK_PIXEL(x1,y1);
          }

          x1 = current.x;
          y1 = current.y +1;
          if (y1 < height ) {
            CHECK_PIXEL(x1,y1);
          }
        }//end while

        //如果小白块较小，则设置为斑点信息
        blockWidth = maxX - minX + 1;
        blockHeight = maxY - minY + 1;
        if(blockWidth > 0 && blockHeight > 0 && blockWidth < width / 10
&&
blockHeight < height / 10)
        {
            for(int k = 0; k < PixelsCount; k++){

image->SetPixelColor(blockPixels[k].x,blockPixels[k].y,RGB(150,150,150));
            }
        }//end if
 }//end for

free(queue);
free(blockPixels);

//过程3：把背景图像的灰度还原为255。
for(y = 0; y < height; y++)
for(x = 0; x < width; x++){
    RGBQUAD color = image->GetPixelColor(x,y);
```

```
    if(color.rgbRed == 200)
   image->SetPixelColor(x,y,RGB(255,255,255));
}
```

最后输出的图像如图 2.17 所示。

图 2.17　连通域分析后得到的图像