计算机视觉总结
SUMMARY OF COMPUTER VISION

(第1版)

LVSHUAILIN

OPEN SOURCE
BEIJING

VERSION 1

一. 数据结构与算法-LeetCode Hot 100

二. PYTHON: 1) NUMPY; 2) PANDAS; 3) PYTHON多进程; 4) PYTHON分布式; 5) PYTHON界面;

三. 深度学习: TensorFlow 2.0; PYTORCH;

四. 图像配准

五. 强化学习

六. OTHERS: 1) Model INFERENCE by EXE; 2) GIT; 3) DOCKER

LVSHUAILIN

2020年2月

目　　录

第1章　LeetCode Hot 100

Goals to Achieve

1. unordered_map.

§ 1.1 两数之和

HOT100 1.1　问题描述

给定一个整数数组**nums** 和一个目标值**target**, 请你在该数组中找出和为目标值的那两个整数, 并返回他们的数组下标. 你可以假设每种输入只会对应一个答案. 但是, 你不能重复利用这个数组中同样的元素.

示例: 给定nums = [2, 7, 11, 15], target = 9; 因为nums[0] + nums[1] = 2 + 7 = 9;

所以返回[0, 1]

https://leetcode-cn.com/problems/two-sum

1.1.1 知识点(unordered_map)

unordered_map内部是一个关联容器, 采用hash 表结构, 有快速检索的功能.

哈希表是通过key关键字直接访问对应value值的数据结构. 特点是键和值一一对应, 查找时间复杂度**O(1)**.

Example_1: unordered_map插入, 迭代遍历.

unordered_map example_1 code

```
1 #include <iostream>
2 #include <unordered_map>
3 #include <string>
4 using namespace std;
5 int main()
6 {
```

```
    unordered_map<string, double> umap;
    umap["PI"] = 3.14;
    umap.insert(make_pair("a", 2.1));

    // find in umap
    string key = "PI";
    if (umap.find(key) == umap.end())
        cout << "cannot find PI" << endl;
    else
        cout << "find " << umap.find(key)->first << " = " << umap.find(key)->second << endl;

    // iterator of umap
    cout << "entire unorded_map is:"<<endl;
    unordered_map<string, double>::iterator itr;
    for (itr = umap.begin(); itr != umap.end(); ++itr)
        cout << "(" << itr->first << "," << itr->second << ")" << endl;
    system("pause");
    return 0;
}
```

output:
find PI = 3
all elements are:
(PI,3.14)
(a,2.1)

Example_2: 利用unordered_map输出一段文字中重复单词的个数

unordered_map example_2 code

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <sstream>

using namespace std;

void printWordFreq(const string& str)
{
    unordered_map<string, int> wordFreq;
    string word;
    stringstream ss(str);
```

```
    while (ss >> word)
        wordFreq[word]++;

    cout << "all elements are:" << endl;
    for (auto u : wordFreq)
        cout << "(" << u.first << "," << u.second << ")" << endl;
}

int main()
{
    string str = "studies very very hard";
    printWordFreq(str);
    return 0;
}
```

output:
all elements are:
(studies, 1)
(very, 2)
(hard, 1)

1.1.2 解题代码

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

vector<int> twoSum(vector<int>& nums, int target)
{
    unordered_map<int, int> map;
    vector<int> result={};
    int n = (int)nums.size();
    for(int i = 0; i < n; ++i) {
        auto p = map.find(target-nums[i]);
        if(p != map.end()) {
        result.push_back(p->second);
        result.push_back(i);
        }
```

```
        map[nums[i]] = i;
        }
        return result;
}

int main()
{
        vector< int > nums = {2,7,11,15};
        vector<int> result;
        result = twoSum(nums,9);
        cout<<"[ "<<result[0] << " , " <<result[1]<<" ] "<<endl;
        return 0;
}
```

§ 1.2 两数相加

HOT100 1.2 问题描述

给出两个非空的链表用来表示两个非负的整数. 其中, 它们各自的位数是按照逆序的方式存储的, 并且它们的每个节点只能存储一位数字. 如果, 我们将这两个数相加起来, 则会返回一个新的链表来表示它们的和.

您可以假设除了数字0之外, 这两个数都不会以0开头.

示例: 输入$(2->4->3) + (5->6->4)$, 输出: $7->0->8$, 原因: $342 + 465 = 807$

https://leetcode-cn.com/problems/add-two-numbers

1.2.1 知识点(linked list)

这里用c++ 链表来解决

Example_1: 创建链表并初始化

linked list example_1 code

```
#include <iostream>

using namespace std;

class Node{
public:
```

```
    int data;
    Node* next;
};

int main()
{
    Node* head = nullptr;
    Node* second = nullptr;
    Node* third = nullptr;

    head = new Node();
    head->data = 1;

    second = new Node();
    second->data = 2;

    third = new Node();
    third->data = 3;

    cout << head->data << "  " << second->data << "  " << third->data << endl;

    delete head;
    delete second;
    delete third;
    return 0;
}
```

output:
1 2 3

Example_2: 打印链表中的所有元素

linked list example_2 code

```
#include <iostream>

using namespace std;

class Node{
public:
    int data;
    Node* next;
```

```
};

void PrintLinkedList(Node* head)
{
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << "  ";
        temp = temp->next;
    }
    cout << endl;
}

int main()
{
    Node* head = nullptr;
    Node* second = nullptr;
    Node* third = nullptr;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = nullptr;


    PrintLinkedList(head);

    delete head;
    delete second;
    delete third;
    return 0;
}
```

output:
1 2 3

Example_3: 链表插入节点

linked list example_3 code

```
#include <iostream>

using namespace std;

class Node{
public:
    int data;
    Node* next;
};

// 在链表前面插入节点
void push(Node** head_ref, int newData)
{
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = (*head_ref);
    (*head_ref) = newNode;
}

// 在节点后面插入节点
void insertAfter(Node** prev_node, int newData)
{
    if ((*prev_node) == nullptr) {
        cout << "the previous node cannot be nullptr" << endl;
        return;
    }

    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = (*prev_node)->next;
    (*prev_node)->next = newNode;
}

// 在尾节点后插入节点
void append(Node** head_ref, int newData)
```

```
{
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = nullptr;
    if ((*head_ref) == nullptr) {
        (*head_ref) = newNode;
        return;
    }

    Node* move = (*head_ref);
    while (move->next != nullptr) {
        move = move->next;
    }
    move->next = newNode;
}

//打印链表
void PrintLinkedList(Node* head)
{
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << "  ";
        temp = temp->next;
    }
    cout << endl;
}

void destroyLinkedList(Node** head_ref) {
    Node* move = (*head_ref);
    Node* next = nullptr;
    while (move != nullptr) {
        next = move->next;
        delete move;
        move = next;
    }
    (*head_ref) = nullptr;
}

int main()
{
```

```
    Node* head = nullptr;

    append(&head, 6);

    push(&head, 7);

    push(&head, 1);

    append(&head, 4);

    insertAfter(&(head->next), 8);

    cout << "linked list is: ";
    PrintLinkedList(head);
    destroyLinkedList(&head);
    return 0;
}
```

output:
linked list is: 1 7 8 6 4

1.2.2 解题代码

```
#include <iostream>
using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    int len1 = 1;//记录的长度l1
    int len2 = 1;//记录的长度l2
    ListNode* p = l1;
    ListNode* q = l2;
    while (p->next != NULL)//获取的长度l1
    {
```

```
        len1++;
        p = p->next;
    }
    while (q->next != NULL)//获取的长度l2
    {
        len2++;
        q = q->next;
    }
    if (len1 > len2)//较长，在末尾补零l1l2
    {
        for (int i = 1; i <= len1 - len2; i++)
        {
            q->next = new ListNode(0);
            q = q->next;
        }
    }
    else//较长，在末尾补零l2l1
    {
        for (int i = 1; i <= len2 - len1; i++)
        {
            p->next = new ListNode(0);
            p = p->next;
        }
    }
    p = l1;
    q = l2;
    bool count = false;//记录进位
    ListNode* l3 = new ListNode(-1);//存放结果的链表
    ListNode* w = l3;//的移动指针l3
    int i = 0;//记录相加结果
    while (p != NULL && q != NULL)
    {
        i = count + p->val + q->val;
        w->next = new ListNode(i % 10);
        count = i >= 10 ? true : false;
        w = w->next;
        p = p->next;
        q = q->next;
    }
    if (count)//若最后还有进位
```

```
57      {
58          w->next = new ListNode(1);
59          w = w->next;
60      }
61      return l3->next;
62  }
63
64  void printLinkedList(ListNode* head)
65  {
66      ListNode* move = head;
67      while (move != nullptr) {
68          cout << move->val << "  ";
69          move = move->next;
70      }
71  }
72
73  int main()
74  {
75  #if 1
76      ListNode* l1 = new ListNode(2);
77      ListNode* l1_1 = new ListNode(4);
78      ListNode* l1_2 = new ListNode(3);
79
80      l1->next = l1_1;
81      l1_1->next = l1_2;
82
83
84      ListNode* l2 = new ListNode(5);
85      ListNode* l2_1 = new ListNode(6);
86      ListNode* l2_2 = new ListNode(4);
87      l2->next = l2_1;
88      l2_1->next = l2_2;
89  #endif
90
91  #if 0
92      ListNode* l1 = new ListNode(5);
93      ListNode* l2 = new ListNode(5);
94  #endif
95
96      ListNode* result = addTwoNumbers(l1, l2);
```

```
97      printLinkedList(result);
98      return 0;
99  }
```

output:
7 0 8

§ 1.3 无重复字符的最长子串

HOT100 1.3 问题描述

给定一个字符串, 请你找出其中不含有重复字符的最长子串的长度.

示例1:

输入: “abcabcb”

输出: 3

解释: 因为无重复字符的最长子串是“abc”, 所以其长度为3.

示例2:

输入: “bbbbb”

输出: 1

解释: 因为无重复字符的最长子串是“b”, 所以其长度为1.

示例3:

输入: “pwwkew”

输出: 3

解释: 因为无重复字符的最长子串是“wke”, 所以其长度为3. 请注意, 你的答案必须是子串的长度, “pwke”是一个子序列, 不是子串.

https://leetcode-cn.com/problems/longest-substring-without-repeating-characters

1.3.1 知识点(double pointer algorithm和unordered_set)

c++提供两种关联型数据结构, 1) 树型结构, 如: map, set; 2) hash结构, 如: unordered_map, unordered_set. map和set是有序的, 其他两个是无序的.

1.3.2 解题思路

https://cloud.tencent.com/developer/article/1377650

这道题主要用到思路是: 滑动窗口

什么是滑动窗口?

其实就是一个队列, 比如例题中的abcabcbb, 进入这个队列(窗口)为abc满足题目要求, 当再进入a, 队列变成了abca, 这时候不满足要求. 所以, 我们要移动这个队列!

如何移动?

我们只要把队列的左边的元素移出就行了, 直到满足题目要求!

一直维持这样的队列, 找出队列出现最长的长度时候, 求出解!

时间复杂度: **O(n)**

1.3.3 解题代码

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <algorithm> // max, min

using namespace std;

int lengthOfLongestSubstring(string s) {
    if (s.size() == 0) return 0;
    unordered_set<char> lookup;
    int maxStr = 0;
    int left = 0;
    for (int i = 0; i < s.size(); i++) {
        while (lookup.find(s[i]) != lookup.end()) {
            lookup.erase(s[left]);
            left++;
        }
        maxStr = max(maxStr, i - left + 1);
        lookup.insert(s[i]);
    }
    return maxStr;
}

int main()
```

```
{
    string str = "abcabcb";
    cout << lengthOfLongestSubstring(str) << endl;
    return 0;
}
```

output:
3

§ 1.4 寻找两个有序数组的中位数

HOT100 1.4 问题描述

给定两个大小为m和n的有序数组nums1和nums2.

请你找出这两个有序数组的中位数, 并且要求算法的时间复杂度为**O(log(m + n))**.

你可以假设nums1和nums2不会同时为空.

示例1:

nums1 = [1, 3]

nums2 = [2]

则中位数是2.0

示例2:

nums1 = [1, 2]

nums2 = [3, 4]

则中位数是(2 + 3)/2 = 2.5

https://leetcode-cn.com/problems/median-of-two-sorted-arrays

1.4.1 知识点(二分查找算法)

用二分查找算法, 也叫做折半查找算法.

Example_1: 二分查找

二分查找算法*example_1 code*

```
// 二分查找– 折半查找
int search(int arr[], int key, int left, int right)
{
    while (left <= right)
```

```
    {
        int mid = left + (right − left) / 2;
        if (key < arr[mid])
            right = mid − 1;
        else if (key > arr[mid])
            left = mid + 1;
        else
            return mid;
    }
    return −1;
}

int main()
{
    int arr[] = { 0,2 ,3,4};
    int value = 3;

    // left Index of the array
    int left = 0;

    // right Index of the array
    int right = sizeof(arr) / sizeof(arr[0]) − 1;

    cout << "left: " << left << ", right: " << right << endl;

    int ret = search(arr, value, left, right);
    if (ret == −1)
        printf("cannot find the value");
    else
        printf("found the value, the index is: %d\n", ret);
    system("pause");
    return 0;
}
```

output:
left: 0, right: 3
found the value, the index is: 2

1.4.2 解题思路

这道题让我们求两个有序数组的中位数，而且限制了时间复杂度为O(log (m+n))，看到这个时间复杂度，自然而然的想到了应该使用二分查找法来求解。那么回顾一下中位数的定义，如果某个有序数组长度是奇数，那么其中位数就是最中间那个，如果是偶数，那么就是最中间两个数字的平均值。这里对于两个有序数组也是一样的，假设两个有序数组的长度分别为m和n，由于两个数组长度之和m+n 的奇偶不确定，因此需要分情况来讨论，对于奇数的情况，直接找到最中间的数即可，偶数的话需要求最中间两个数的平均值。为了简化代码，不分情况讨论，我们使用一个小trick，我们分别找第(m+n+1) / 2 个，和(m+n+2) / 2 个，然后求其平均值即可，这对奇偶数均适用。假如m+n 为奇数的话，那么其实(m+n+1) / 2 和(m+n+2) / 2 的值相等，相当于两个相同的数字相加再除以2，还是其本身。

这里我们需要定义一个函数来在两个有序数组中找到第K个元素，下面重点来看如何实现找到第K个元素。首先，为了避免产生新的数组从而增加时间复杂度，我们使用两个变量i和j分别来标记数组nums1和nums2的起始位置。然后来处理一些边界问题，比如当某一个数组的起始位置大于等于其数组长度时，说明其所有数字均已经被淘汰了，相当于一个空数组了，那么实际上就变成了在另一个数组中找数字，直接就可以找出来了。还有就是如果K=1的话，那么我们只要比较nums1和nums2的起始位置i和j上的数字就可以了。难点就在于一般的情况怎么处理？因为我们需要在两个有序数组中找到第K个元素，为了加快搜索的速度，我们要使用二分法，对K二分，意思是我们需要分别在nums1和nums2中查找第K/2个元素，注意这里由于两个数组的长度不定，所以有可能某个数组没有第K/2个数字，所以我们需要先检查一下，数组中到底存不存在第K/2个数字，如果存在就取出来，否则就赋值上一个整型最大值。如果某个数组没有第K/2个数字，那么我们就淘汰另一个数字的前K/2个数字即可。有没有可能两个数组都不存在第K/2个数字呢，这道题里是不可能的，因为我们的K不是任意给的，而是给的m+n的中间值，所以必定至少会有一个数组是存在第K/2个数字的。最后就是二分法的核心啦，比较这两个数组的第K/2小的数字midVal1和midVal2的大小，如果第一个数组的第K/2个数字小的话，那么说明我们要找的数字肯定不在nums1中的前K/2个数字，所以我们可以将其淘汰，将nums1的起始位置向后移动K/2个，并且此时的K也自减去K/2，调用递归。反之，我们淘汰nums2中的前K/2个数字，并将nums2的起始位置向后移动K/2个，并且此时的K也自减去K/2，调用递归即可。

1.4.3 解题代码

```
/*分清 起始位置和第几个元素*/
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int findKthNumber(vector<int>& nums1, int i, vector<int>& nums2, int j, int k) {
    if (i >= nums1.size()) return nums2[j + k - 1];
    if (j >= nums2.size()) return nums1[i + k - 1];
    //if(k ==1) return (double(nums1[i] + nums2[j]));wrong
```

```
11      if (k == 1) return min(nums1[i], nums2[j]);
12      //查找有没有k个元素的位置/2 i + k/2 −1
13      int midVal1 = (i + k / 2 − 1 < nums1.size()) ? nums1[i + k / 2 − 1] : INT_MAX;
14      int midVal2 = (j + k / 2 − 1 < nums2.size()) ? nums2[j + k / 2 − 1] : INT_MAX;
15      if (midVal1 < midVal2)
16          return findKthNumber(nums1, i + k / 2, nums2, j, k − k / 2);
17      else
18          return findKthNumber(nums1, i, nums2, j + k / 2, k − k / 2);
19  }
20  class Solution {
21  public:
22      double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
23          int m = nums1.size(), n = nums2.size();
24          int left = (m + n + 1) / 2, right = (m + n + 2) / 2;
25          return (findKthNumber(nums1, 0, nums2, 0, left) + findKthNumber(nums1, 0, nums2, 0, right)) / 2.0;
26      }
27  };
```

output:
Null

第2章　Bayesian-Python

学习目标与要求

1. .
2. .
3. .
4. .

§ 2.1 Probalilities

A common and useful conceptualization in statistics is to think that data was generated from some probability distribution with unobserved parameters.

2.1.1 Gaussian Distribution

$$pdf(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

mu_params = [−1, 0, 1]
sd_params = [0.5, 1, 1.5]
x = np.linspace(−7, 7, 100)
f, ax = plt.subplots(len(mu_params), len(sd_params), sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
```

```
        mu = mu_params[i]
        sd = sd_params[j]
        y = stats.norm(mu, sd).pdf(x)
        ax[i, j].plot(x,y)
        ax[i, j].plot(0, 0, label="$\\mu$ = {:3.2f}\n$\\sigma$={:3.2f}".format (mu, sd), alpha=0)
        ax[i, j].legend(fontsize=12)
    ax[2,1].set_xlabel('$x$', fontsize=16)
    ax[1,0].set_ylabel('$pdf(x)$', fontsize=16)
    plt.tight_layout()
```

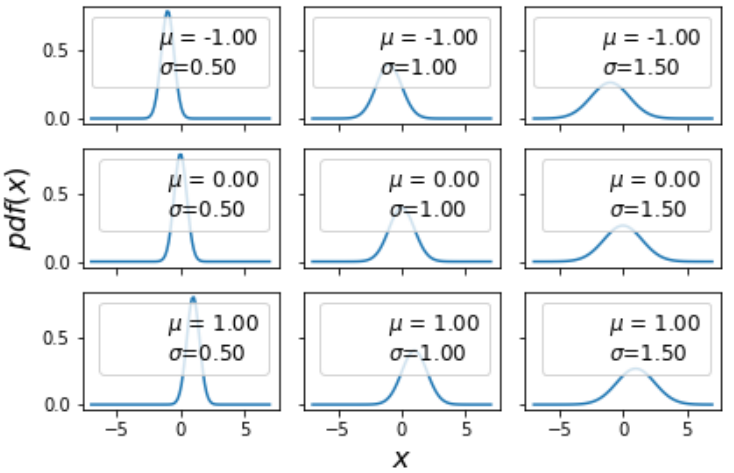

图 2.1 Gaussion Distribution

2.1.2 Coin-Flipping Problen

We will answer this question in a Bayesian setting. We will need data and a probabilistic model.

Data: we will assume that we have already tossed a coin a number of times and we have recorded the number of observed head, so the data part is done.

Model will be discussed soon.

2.1.2.1 The general model

The firs thing we will do is generalize the concept of bias. We will say that a coin with a bias of 1 will always land heads, one with a bias of 0 will always land tails, and one with a bias of 0.5 will land half of the time heads and half of the time tails. To represent the bias, we will use the parameter θ, and to represent the total number of heads for an N number of tosses, we will use the variable y. According to Bayes' theorem we have the following formula:

$$p(\theta|y) \propto p(y|\theta)p(\theta)$$

Notice that we need to specify which prior $p(\theta)$ and likelihood $p(y|\theta)$ we will use. Let's start with the likelihood.

2.1.2.2 Choosing the likelihood

Let's assume that a coin toss does not affect other tosses, that is, we are assuming coin tosses are independent of each other. Let's also assume that only two outcomes are possible, heads or tails. Given these assumptions, a good candidate for the likelihood is the binomial distribution (二项分布):

$$p(y|\theta) = \frac{N!}{N!(N-y)!}\theta^y(1-\theta)^{N-y}$$

This is a discrete distribution returning the probability of getting y heads (or in general, success) out of N coin tosses (or in general, trials or experiments) given a fixed value of θ. The following code generates 9 binomial distributions; each subplot has its own legend indicating the corresponding parameters:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

n_params = [1,2,4]
p_params = [0.25, 0.5, 0.75]
x = np.arange(0, max(n_params) + 1)
f, ax = plt.subplots(len(n_params), len(p_params), sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
        n = n_params[i]
        p = p_params[j]
        y = stats.binom(n=n, p=p).pmf(x)
        ax[i,j].vlines(x, 0, y, colors='b', lw=5)
        ax[i,j].set_ylim(0,1)
        ax[i,j].plot(0,0,label="n = {:3.2f}\np = {:3.2f}".format(n,p), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[2,1].set_xlabel('$\\theta$', fontsize=14)
ax[1,0].set_ylabel('$p(y|\\theta)$', fontsize=14)
ax[0,0].set_xticks(x)
plt.savefig('binomial_distribution.png')
```

The binomial distribution is also a reasonable choice for the likelihood. Intuitively, we can see that θ indicates how likely it is that we will obtain a head when tossing a coin, and we have observed that event y times. Following the same line of reasoning we get that $1-\theta$ is the chance of getting a tail, and that event has occurred $N-y$ times.

OK, so if we know θ, the binomial distribution will tell us the expected distribution of head. The only problem is that we do not kno θ!. But do not despair; in Bayesian statistics, every time we do not know the value of a parameter, we put a prior on it, so let's move on and choose a prior.

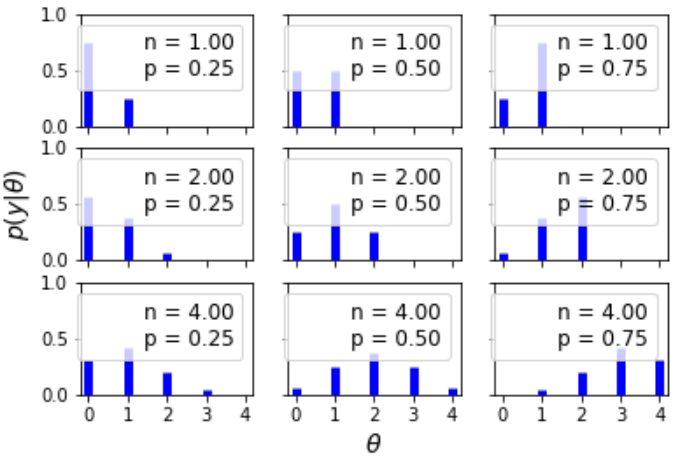

图 2.2 binomial_distribution

2.1.2.3 Choosing the prior

As a prior we will use a beta distribution (贝塔分布), which is a very common distribution in Bayesian statistics and looks like this:

$$p(\theta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

The first term is a normalization constant that ensures the distribution integrates to 1.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

params = [0.5, 1, 2, 3]
x = np.linspace(0, 1, 100)
f, ax = plt.subplots(len(params), len(params), sharex=True, sharey=True)
for i in range(4):
    for j in range(4):
        a = params[i]
        b = params[j]
        y = stats.beta(a, b).pdf(x)
        ax[i,j].plot(x, y)
        ax[i,j].plot(0,0,label="$\\alpha$ = {:3.2f}\n$\\beta$={:3.2f}".format(a,b), alpha=0)
        ax[i,j].legend(fontsize=12)
ax[3,0].set_xlabel('$\\theta$', fontsize=14)
ax[0,0].set_ylabel('$p(\\theta)$', fontsize=14)
```

```
19 plt.savefig('beta_distribution.png')
```

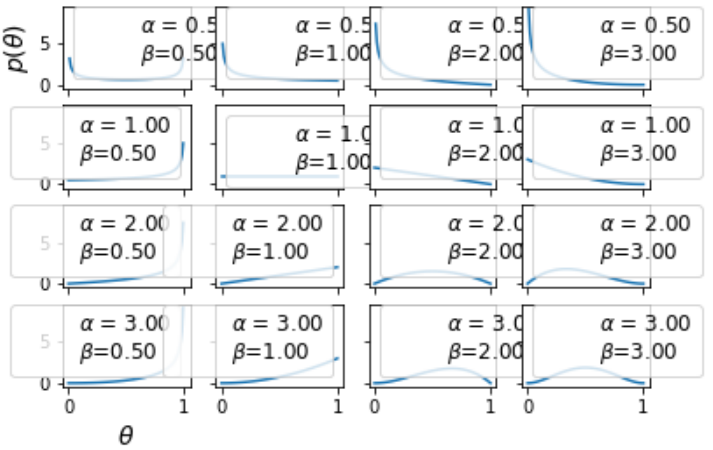

图 2.3 beta_distribution

Why are we using the beta distribution for our model? 1) One reason is that the beta distribution is restricted to be between 0 and 1, in the same way our parameter θ is. 2) Another reason is its versatility (通用性). As we can see in the preceding figure, the distribution adopts several shapes, including a uniform distribution, Gaussian-like distributions, U-like distributions, and so on. 3) A third reason is that the beta distribution is the conjugate prior (共轭先验) of the binomial distribution (which we are using as the likelihood). A conjugate prior of a likelihood is a prior that, when used in combination with the given likelihood, returns a posterior with the same functional form as the prior. There are other pairs of conjugate priors, for example, the Gaussion distribution is the conjugate prior of itself.

For many years, Bayesian analysis was restricted to the use of conjugate priors. Conjugacy ensures mathematical tractability of the posterior, which is important given that common problem in Bayesian statics is to end up with a posterior we cannot solve analytically. This was a deal beaker before the development of suitable computational methods to solve any possible posterior.

However, modern computational methods to solve Bayesian problems whether we choose conjugate priors or not.

2.1.2.4 Getting the posterior

The Bayes' theorem says that the posterior is proportional to the likelihood times the prior:

$$p(\theta|y) \propto p(y|\theta)p(\theta)$$

which turns out to be

$$p(\theta|y) \propto \frac{N!}{N!(N-y)!}\theta^y(1-\theta)^{N-y}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

To our practical concerns we can drop all the terms that do not depend on θ and our results will still be valid. So we can write the following:

$$p(\theta|y) \propto \theta^y(1-\theta)^{N-y}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

Reordering it, we have

$$p(\theta|y) \propto \theta^{\alpha-1+y}(1-\theta)^{\beta-1+N-y}$$

We will see that this expression has the same functional form of a beta distribution (except for the normalization) with $\alpha_{posterior} = \alpha_{perior} + y$ and $\beta_{posterior} = \beta_{prior} + N - y$, which means that the posterior for our problem is the beta distribution:

$$p(\theta|y) = Beta(\alpha_{prior} + y, \beta_{prior} + N - y)$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

theta_real = 0.35
trials = [0, 1, 2, 3, 4, 8, 16, 32, 50, 150]
data = [0, 1, 1, 1, 1, 4, 6, 9, 13, 48]

beta_params = [(1,1), (0.5,0.5), (20,20)]
dist = stats.beta
x = np.linspace(0, 1, 100)

for idx, N in enumerate(trials):
    if idx == 0:
        plt.subplot(4, 3, 2)
    else:
        plt.subplot(4, 3, idx+3)
    y = data[idx]
    for (a_prior, b_prior), c in zip(beta_params, ('b', 'r', 'g')):
        p_theta_given_y = dist.pdf(x, a_prior+y, b_prior+N-y)
        plt.fill_between(x, 0, p_theta_given_y, color=c, alpha=0.6)

    plt.axvline(theta_real, ymax=0.3, color='k')
    plt.plot(0,0,label="{:d} experiments\n{:d} heads".format(N,y),alpha=0)
    plt.xlim(0,1)
    plt.ylim(0,12)
    plt.xlabel(r'$\theta$')
    plt.legend()
    plt.gca().axes.get_yaxis().set_visible(False)
plt.tight_layout()
plt.savefig("posterior.png")
```

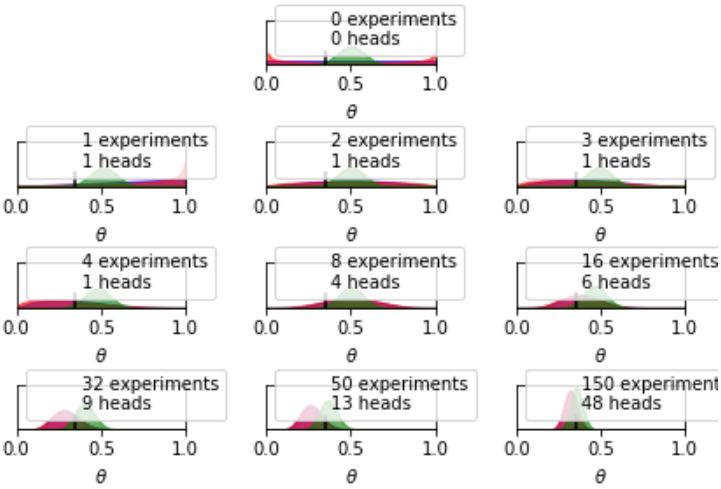

图 2.4 posterior

2.1.2.5 Model notation and visualization

A common notation to succinctly represent probabilistic is as follows:

$$\checkmark \theta \sim Beta(\alpha, \beta)$$

$$\checkmark y \sim Bin(n = 1, p = \theta)$$

2.1.3 Inference engines

There are several methods to compute the posterior even when it is not possible to solve it analytically. Some of the methods are:

1. Non-Markovian methods:
1.1 Grid computing
1.2 Quadratic approximation
1.3 Variation methods

2. Markovian methods:
2.1 Metropolis-Hastings
2.2 Hamiltonian Monte Carlo/No Y-Turn Sampler

NOwadays, Bayesian analysis is performed maninly by using Markov Chain Monte Carlo (MCMC) methods, with variational methods gaining momentum for bigger datasets. We do not need to really understand these methods to perform Bayesian analysis, that's the whole point of probabilistic programming languages, but konwing at least how they work at a conceptual level is often vary useful.

2.1.3.1 Non-Markovian methods

Let's start our discussion of inference engines with the non-Markovian methods. These methods are in general faster than Markovian ones.

Grid computing

Grid computing is a brute-force approach. Even if you are not able to compute the whole posterior, you may be able to compute the prior and the likelihood for a given number of points. Let's assume we want to compute the posterior for a single parameter model. The grid approximation is as follows:

1. Define a reasonable interval for the parameter (the prior should give you a hint).
2. Place a grid of points (generally equidistant) on that interval.
3. For each point in the grid we multiply the likelihood and the prior.

Optionally, we may normalize the computed values (divide the result at each point by the sum of all points).

It is easy to see that a larger number of points (or equivalently a reduced size of the grid) will result in a better approximation. In fact, if we take an infinite number of points we will get the exact posterior. The grid approach does not scale well for many parameters (also referred as dimensions); as you increase the number of parameters the volume of the posterior gets relatively smaller compared with the sampled volume. In other words, we will spend most of the time computing values with an almost null contribution to be posterior, making this approach unfeasible for many statistical and data problems.

The following code implements the grid approach to solve the coin-flipping problem.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

def posterior_grid_approx(grid_points=100, heads=6, tosses=9):
    """
    A grid implementation for the coin-flip problem
    """
    grid = np.linspace(0, 1, grid_points)
    prior = np.repeat(5, grid_points)
    likelihood = stats.binom.pmf(heads, tosses, grid)
    unstd_posterior = likelihood * prior
    posterior = unstd_posterior / unstd_posterior.sum()
    return grid, posterior

# Assuming we made 4 tosses and we observe only 1 head we have the following:

points = 15
h, n = 1, 4
```

```
grid, posterior = posterior_grid_approx(points, h, n)
plt.plot(grid, posterior, 'o-', label='heads = {}\ntosses = {}'.format(h, n))
plt.xlabel(r'$\theta$')
plt.legend(loc=0)
plt.savefig('grid_approach.png')
```

$$\checkmark \theta \sim Beta(\alpha, \beta)$$

$$\checkmark y \sim Bin(n = 1, p = \theta)$$

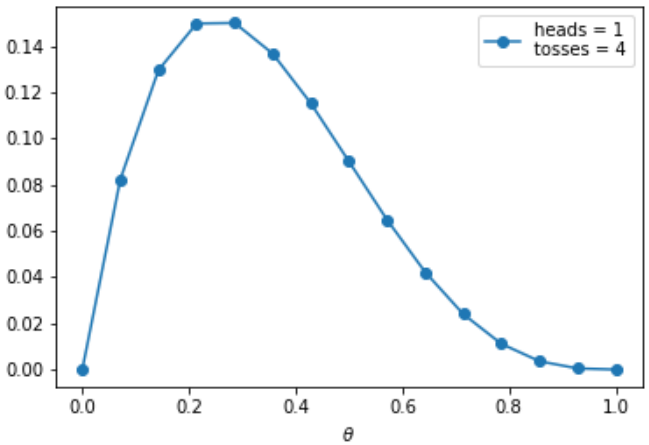

图 2.5 grid_approach to solve the coin flipping

Quadratic method The quadratic approximation, also known as the Laplace method or the normal approximation, consists of approximating the posterior with a Gaussion distribution. This method often works because in general the region close to the mode of the posterior distribution is more or less normal, and in fact in many cases in actually a Gaussian distribution. This method consists of two steps. First, find the mode of the posterior distribution. This method consists of two steps. Fist, find the mode of the posterior distribution. This can be done using optimization methods; that is, methods to find the maximum or minimum of a function, and there are many off-the-shelf methods for this purpose. This will be the mean of the approximating Gaussion. Then we can estimate the curvature of the function near the mode. Based on this curvature, the standard deviation of the approximating Gaussian can be computer. We are going to apply this method once we have introduced PyMC3.

Variational methods Most of modern Bayesian statics is done using Markovian methods (see the next section), but for some problems those methods can be too slow and they do not necessarily parallelize well. The naive approach is to simply run n chains in parallel and then combine the results, but for many problems this is not a really good solution. Finding effective ways of parallelizing them is an active research area.

Variational methods could be a better choice for large datasets (think big data) and/or for likelihoods that are too expensive to compute. In addition, these methods are useful for quick approximations to the posterior and as starting points for MCMC methods.

The general idea of variational methods is to approximate the posterior with a simpler distribution; this may sound similar to the Laplace approximation, but the similarities vanish when we check the details of the method. The main drawback of variational methods is that we must come up with a specific algorithm for each model, so it is not really a universal inference engine, but a model-specific one.

Of course, lots of people have tried to automatize variational methods. A recently proposed method is the automatic differentiation variational inference (ADVI) (see http://arxiv.org/abs/1603.00788). At the conceptual level, ADVI works in the following way:

1. Transform the parameters to make them live in the real line. For example, taking the logarithm of a parameter restricted to positive values we obtain an unbounded parameter on the interval $[-\infty, \infty]$.

2. Approximate the unbounded parameters with a Gaussian distribution. Notice that a Gaussian on the transformed parameter space is non-Gaussian on the original parameter space, hence this is not the same as the Laplace approximation.

3. Use an optimization method to make the Gaussian approximation as close as possible to the posterior. This is done by maximizing a quantity known as the Evidence LowerBound (ELBO). How we measure the similarity of two distributions and what ELBO is exactly, at this point, is a mathematical detail.

Markovian methods There is a family of related methods collectively know as MCMC methods. As with the grid computing approximation, we need to be able to compute the likelihood and prior for a given point and we want to approximate the whole posterior distribution. MCMC methods outperform the grid approximation because they are designed to spend more time in higher probability regions than in lower ones. In fact, a MCMC method will visit different regions of the parameter space in accordance with their relative probabilities. If region A is twice as likely as region B, then we are going to get twice the samples from A as from B. Hence, even if we are not capable of computing the whole posterior analytically, we could use MCMC methods to take samples from it, and the larger the sample size the better the results.

What is in a name? Well, sometimes not much, sometimes a lot. To understand what MCMC methods are we are going to split the method into the two MC parts, the Monte Carlo part and the Markov Chain part.

Monte Carlo The use of random numbers explains the Monte Carlo part of the name. Monte Carlo methods are a very broad family of algorithms that use random sampling to compute or simulate a given process. Monte Carlo is a very famous casino located in the Principality of Monaco. One of the developers of the Monte Carlo method, Stanislaw Ulam, had an uncle who used to gamble there. The key idea Stan had was that while many problems are difficult to solve or even formulate in an exact way, they can be effectively studied by taking samples from them, or by simulating them. In fact, as the story goes, the motivation was to answer questions about the probability of getting a particular hand in a solitary game. One way to solve this problem was to follow the analytical combinatorial problem. Another way, Stan argued, was to play several games of solitaire and just count how many of the hands we play match the particular hand we are interested in! Maybe this sounds obvious, or at least pretty reasonable; for example, you may have used re-sampling methods to solve your statistical problems. But remember this mental experiment was

performed about 70 years ago, a time when the first practical computers began to be developed. The first application of the method was to solve a problem of nuclear physics, a problem really hard to tackle using the conventional tools at that time. Nowadays, even personal computers are powerful enough to solve many interesting problems using the Monte Carlo approach and hence these methods are applied to a wide variety of problems in science, engineering, industry, arts, and so on.

A classic pedagogical example of using a Monte Carlo method to compute a quantity of interest is the numerical estimation of . In practice there are better methods for this particular computation, but its pedagocial value still remains. We can estimate the value of π with the following procedure:

1. Throw N points at random into a square of side $2R$.
2. Draw a circle of radius R inscribed in the square and count the number of points that are inside that circle.
3. Estimate $\hat{\pi}$ as the ration $\frac{4 \times inside}{N}$.

A couple of notes: We know a point is inside a circle if the following relation is true: $\sqrt{(x^2 + y^2)} \leq R$

The area of the square is $(2R)^2$ and the area of the circle is πR^2. Thus we know that the ratio of the area of the square to be the area of the circle is $\frac{4}{\pi}$, and the area of the circle and square are proportional to the number of points inside the circle and the total N points, respectively.

Using a few line of Python we can run this simple Monte Carlo simulation and compute π and also the relative error of our estimate compared to the rule value of π:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

N = 10000
x,y = np.random.uniform(−1, 1, size=(2, N))
inside = (x**2 + y**2) <= 1
pi = inside.sum() * 4 / N
error = abs((pi − np.pi) / pi ) * 100

outside = np.invert(inside)

plt.plot(x[inside], y[inside], 'b.')
plt.plot(x[outside], y[outside], 'r.')
plt.plot(0, 0, label='$\hat \pi$ = {:4.3f}\nerror = {:4.3f}'.format(pi, error), alpha=0)
plt.axis('square')
plt.legend(frameon=True, framealpha=0.9, fontsize=16)

plt.savefig('monte_carlo.png')
```

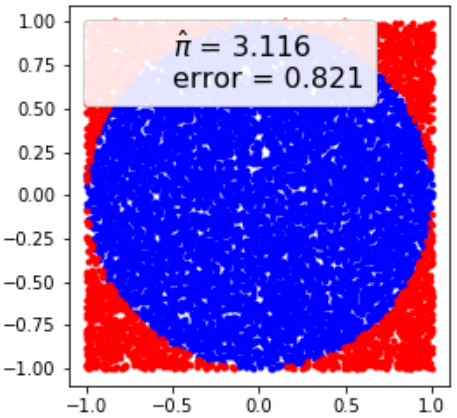

图 2.6 monte_carlo

In the preceding code we can see that the outside variable is only used to get the plot; we do not need it for computing $\frac{4 \times inside}{N}$. Another clarification: because our computation is restricted to the unit circle we can omit computing the square root from the computation of the inside variable.

Markov chain A Markov chain is a mathematical object consisting of a sequence of states and a set of probabilities describing the transitions among those states. A chain is Markovian if the probability of moving to other states depends only on the current state. Given such a chain, we can perform a random walk by choosing a starting point and moving to other states following the transition probabilities. If we somehow find a Markov chain with transitions proportional to the distribution we want to sample from (the posterior distribution in Bayesian analysis), sampling becomes just a matter of moving between states in this chain. So, how do we find this chain if we do not know the posterior in the first place? Well, there is something known as **detailed balance condition**. Intuitively, this condition says that we should move in a reversible way (a reversible process is a common approximation in physics). That is, the probability of being in state i and moving to state j should be the same as the probability of being in state j and moving towards state i.

In summary, all this means that if we manage to create a Markov Chain satisfying detailed balance we can sample from that chain with the guarantee that we will get samples from the correct distribution. This is a truly remarkable result! The most popular method that guarantees detailed balance is the **Metropolis-Hasting algorithm**.

Metropolis-Hastings To conceptually understand this method, we are going to use the following analogy. Suppose we are interested in finding the volume of water a lake contains and which part of the lake has the deepest point. The water is really muddy so we can't estimate the depth just by looking to the bottom, and the lake is really big, so a grid approximation does not seem like a very good idea. In order to develop a sampling strategy, we seek help from two of our best friends, Markovia and Monty. After some discussion they come up with the following algorithm that requires a boat; nothing fancy, we can even use a wooden raft, and a very long stick. This is cheaper than a sonar and we have already spent all our money on the boat, anyway!

1. Initialize the measuring by choosing a random place in the lake and move the boat there.

2. Use the stick to measure the depth of the lake.
3. Move the boat to some other point and take a new measurement.
4. Compare the two measures in the following way:
4.1. If the new spot is deeper than the old one, write down in your notebook the depth of the new spot and repeat from 2.
4.2 If the spot is shallower than the old one, we have two options: to accept or reject. Accepting means to write down the depth of the new spot and repeat from 2. Rejecting means to go back to the old spot and write down (again) the value for the depth of the old spot.

How do we decide to accept or reject a new spot? Well, the trick is to apply the Metropolis-Hastings criteria. This means to accept the new spot with a probability that is proportional to the ratio of the depth of the new and old spots.

If we follow this iterative procedure, we will get not only the total volume of the lake and the deepest point, but we will also get an approximation of the entire curvature of the bottom of the lake. As you may have already guessed, in this analogy the curvature of the bottom of the lake is the posterior distribution and the deepest point is the mode. According to our friend Markovia, the larger the number of iterations the better the approximation.

Indeed, theory guarantees that under certain general circumstances, we are going to get the exact answer if we get an infinite number of samples. Luckily for us, in practice and for many, many problems, we can get a very accurate approximation using a relatively small number of samples.

Let's look at the method now in a little bit more formal way. For some distributions, like the Gaussian, we have very efficient algorithms to get samples from, but for some other distributions such as many of the posterior distributions, we are going to find this is not the case. Metropolis-Hastings enables us to obtain samples from any distribution with probability $p(x)$ given that we can compute at least a value proportional to it. This is very useful since in a lot of problems like Bayesian statistics the hard part is to compute the normalization factor, the denominator of the Bayes' theorem. The Metropolis-Hastings algorithm has the following steps:

1. Choose an initial value for our parameter x_i. This can be done randomly or by using some educated guess.

2. We choose a new parameter value x_{i+1}, sampling from an easy-to-sample distribution such as a Gaussian or uniform distribution $Q(x_{i+1}|x_i)$. We can think of this step as perturbing the state x_i somehow.

3. We compute the probability of accepting a new parameter value by using the Metropolis-Hastings criteria $p_\alpha(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})q(x_i|x_{i+1})}{p(x_i)q(x_{i+1}|x_i)}\right)$.

4. If the probability computed on 3 is larger than the value taken from a uniform distribution on the interval $[0, 1]$ we accept the new state, otherwise we stay in the old state.

5. We iterate from 2 until we have enough samples. Later we will see what enough means.

A couple of things to take into account:

1. If the proposal distribution $Q(x_{i+1}|x_i)$ is symmetric we get $p_\alpha(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})}{p(x_i)}\right)$, often referred to as **Metropolis criteria** (we drop the **Hastings** apart).

2. Steps 3 and 4 imply that we will always accept or move to a most probable state, to a most probable parameter value. Less probable parameter values are accepted probabilistically given the ratio between the probability of the

new parameter value x_{i+1} and the old parameter value x_i. This criteria for accepting steps gives us a more efficient sampling approach compared to the grid approximation, while ensuring a correct sampling.

3. The target distribution (the posterior distribution in Bayesian statistics) is approximated by saving the sampled (or visited) parameter values. We save a sampled value x_{i+1} if we accept moving to a new state x_{i+1}. If we reject moving to x_{i+1}, we save the value of x_i.

At the end of the process we will have a list of values sometimes refereed to as a **sample chain or trace**. If everything was done the right way these samples will be an approximation of the posterior. The most frequent values in our trace will be the most probable values according to the posterior. An advantage of this procedure is that analyzing the posterior is simple. We have effectively transformed integrals (of the posterior) into just summing values in our vector of sampled values. The following code illustrates a very basic implementation of the Metropolis algorithm. Is not meant to solve any real problem, only to show it is possible to sample from a function if we know how to compute its value at a given point. Notice also that the following implementation has nothing Bayesian in it; there is no prior and we do not even have data! Remember that the MCMC methods are very general algorithms that can be applied to a broad array of problems. For example, in a (non-Bayesian) molecular model, instead of func.pdf(x) we would have a function computing the energy of the system for the state x.

The first argument of the metropolis function is a SciPy distribution; we are assuming we do not know how to directly get samples from this distribution.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

def metropolis(func, steps=10000):
    """
    A very simple Metropolis implementation.
    """
    samples = np.zeros(steps)
    old_x = func.mean()
    old_prob = func.pdf(old_x)

    for i in range(steps):
        new_x = old_x + np.random.normal(0, 0.5)
        new_prob = func.pdf(new_x)
        acceptance = new_prob / old_prob
```

```
        if acceptance >= np.random.random():
            samples[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            samples[i] = old_x
    return samples

"""
In the next example we have defined func as a beta function,
simply because is easy to change their parameters and get
different shapes. We are plotting the samples obtained by
metropolisas a histogram and also the True distribution as
a red line:
"""

func = stats.beta(0.4, 2)
samples = metropolis(func=func)
x = np.linspace(0.01, .99, 100)
y = func.pdf(x)
plt.xlim(0, 1)
plt.plot(x, y, 'r-', lw=3, label='True distribution')
plt.hist(samples, bins=30, normed=True, label='Estimated distribution')
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$pdf(x)$', fontsize=14)
plt.legend(fontsize=14)
plt.savefig('metropolis_algorithm.png')
```

Hamiltonian Monte Carlo/NUTS MCMC methods, including Metropolis-Hastings, come with the theoretical guarantee that if we take enough samples we will get an accurate approximation of the correct distribution. However, in practice it could take more time than we have to get enough samples. For that reason, alternatives to the general Metropolis-Hastings algorithm have been proposed. Many of those alternative methods such as the Metropolis-Hastings algorithm itself, were developed originally to solve problems in statistical mechanics, a branch of physics that studies properties of atomic and molecular systems. One such modification is known as **Hamiltonian Monte Carlo or Hybrid Monte Carlo (HMC)**. In simple terms a Hamiltonian is a description of the total energy of a physical system. The name Hybrid is also used because is was originally conceived as a hybridization of molecular mechanics, a widely used simulation technique for molecular systems, and Metropolis-Hastings. The HMC method is essentially the same as Metropolis-Hastings except that instead of proposing random displacements of our boat we do something more clever instead; we move the boat following the curvature of the lake's bottom. Why is this clever? Because in doing so we try to avoid one of the main problems of Metropolis-Hastings: the exploration is slow and samples tend

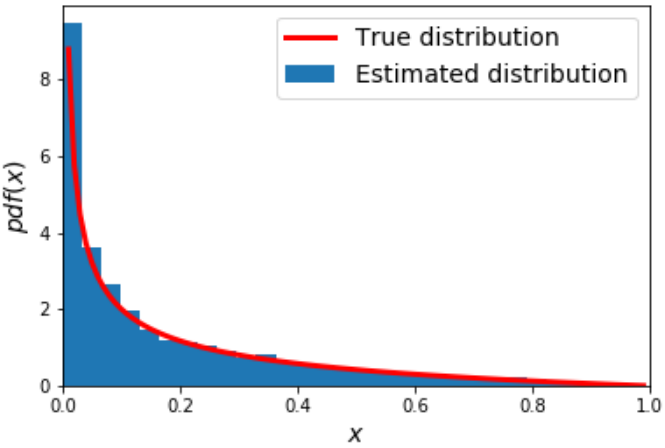

图 2.7 metropolis_algorithm

to be autocorrelated, since most of the proposed moves are rejected.

So, how can we try to understand this method without going into mathematical details? Imagine we are in the lake with our boat. In order to decide where to move next we let a ball roll at the bottom of the lake, starting from our current position. Remember that this method was brought to us by the same people that treat horses as spheres, so our our ball is not only perfectly spherical, it also has no friction and thus is not slowed down by the water or mud. Well, we throw a ball and we let it roll for a short moment, and then we move the boat to where the ball is. Now we accept or reject this step using the Metropolis criteria just as we saw with the Metropolis-Hastings method. The whole procedure is repeated a number of times. This modified procedure has a higher chance of accepting new positions, even if they are far away relative to the previous position.

Out of our Gedanken experiment and back to the real world, the price we pay for this much cleverer Hamiltonian-based proposal is that we need to compute gradients of our function. A gradient is just a generalization of the concept of derivative to more than one dimension. We can use gradient information to simulate the ball moving in a curved space. So, we are faced with a trade-off; each HMC step is more expensive to compute than a Metropolis-Hastings one but the probability of accepting that step is much higher with HMC than with Metropolis. For many problems, this compromise turns in favor of the HMC method, especially for complex ones. Another drawback with HMC methods is that to have really good sampling we need to specify a couple of parameters. When done by hand it takes some trial and error and also requires experience from the user, making this procedure a less universal inference engine than we may want. Luckily for us, PyMC3 comes with a relatively new method known as **No-U-Turn Sampler (NUTS)**. This method has proven very useful in providing the sampling efficiency of HMC methods, but without the need to manually adjust any knob.

Other MCMC methods There are plenty of MCMC methods out there and indeed people keep proposing newmethods, so if you think you can improve sampling methods there is a wide range of persons that will be interested in your ideas. Mentioning all of them and their advantages and drawbacks is completely out of the scope of this book. Nevertheless, there are a few worth mentioning because you may hear people talk about them, so it is nice

to at least have an idea of what are they talking about.

Another sampler that has been used extensively for molecular systems simulations is the **Replica Exchange** method, also known as **parallel tempering** or **Metropolis Coupled MCMC** (or MC3; maybe that's too many MCs). The basic idea of this method is to simulate different replicas in parallel. Each replica follows the Metropolis-Hastings algorithm. The only difference between replicas is that the value of a parameter called temperature (physics influence once more time!) controls the probability of accepting less probable positions. From time to time, the method attempts a swap between replicas. The swapping is also accepted/rejected according to the Metropolis-Hastings criteria, but this time taking into account both replicas' temperatures. The swapping between chains can be attempted between random chains but it is generally preferable to do it for neighboring replicas; that is, replicas with similar temperatures and hence a higher probability-of-acceptance ratio. The intuition for this method is that as we increase the temperature the probability of accepting the new proposed position increases, and decreases with lower and lower temperatures. Replicas at higher temperatures explore the system more freely; for these replicas the surface becomes effectively flatter and thus easier to explore. For a replica with infinite temperature, all states are equally likely. The exchange between replicas avoids replicas at low temperatures getting trapped in local minima. This method is well suited for exploring systems with multiple minima.

2.1.4 PyMC3 instroduction

第3章　深度学习

Goals to Achieve

1. pytorch basics
2. pytorch projects

§ 3.1 Pytorch

打印模型结构

pip install torchsummary

summary(model, (3, 32, 32))

print(model)