**第五次会议**

**1.论文分享：**

**李强：**

**篇名：**卷积神经网络综述（计算机学报）

**重要内容：**1.**网中网结构（NIN）简述？**

答：网中网结构使用微型神经网络代替传统CNN的卷积过程，微神经网络主要是采用MLP模型，MLP (1).采用BP算法进行训练。(2)是一种深度模型，具有特征重用的思想。(3).MLP卷积层能够处理更复杂的线性问题，提取更加抽象的特征

2**.空间变换网络（STNs）简述？**

答：CNN会收到数据在空间上多样性的影响，因此提出空间变换网络来解决此问题。STNs可用于输入层，也可插入到卷积层或其他层后边，能够自适应的对数据进行空间变换和对齐，使得CNN模型对平移、缩放、旋转或其他的变换保持不变性。

**3.有监督学习和无监督学习的区别？**

答：有监督学习：对具有概念标记（分类）的训练样本进行学习，以尽可能的对训练样本集外的数据进行标记预测。因此，训练样本的歧义性低。如分类。

无监督学习：对没有概念标记（分类）的训练样本进行学习，以发现训练样本中的结构性知识。因此，训练样本的歧义性高，聚类就是典型的无监督学习。

CNN既可以进行有监督训练，也可以进行无监督训练，有监督学习需要大量有标注的样本，无监督学习需要训练的参数多，因此提出一个只需调整一个超参数的无监督学习算法---稀疏滤波。

**4.常用开源框架以及特点？**

答：Caffe基于C++语言，可在CPU和GPU上运行，支持MATLAB和python接口；

Torch基于Lua语言，可在CPU和GPU上运行，但在CPU中运行优于其他框架。对于Torch 的时域卷积，其输入长度可变，非常有助于自然语言任务，但是没有python接口；

Theano基于Python，可在CPU和GPU上运行，但在GPU上运行小网络，Theano较优；在 GPU上训练和部署RNN模型，其性能最好；

Tensorflow主要基于C++,可在CPU和GPU上运行.在单个GPU上运行不如其他几个架构。

**徐聪：**

**篇名：**基于视觉手势识别的研究 —综述（电子学报）

**内容：**从手势建模 、手势分析和手势识别等三个方面综述了基于视觉手势识别的研究现状及其应用。

**手势建模：基于3D模型的手势建模：**分体模型、网格模型 、几何模型以及骨架模型。

**基于表观的手势模型：**基于2D灰度图像、基于可变形2D模板、基于图像属性、基于 运动参数。

**手势分析：特征检测：任务：**估计选定的手势模型的参数。

**定位技术：**基于颜色定位 、基于运动定位 、多模式定位。

**图像特征基元：**灰度图像、二值影像、区域、边界及轮廓或者指尖。

**模型参数估计：3D模型：两组参数：**角度参数 (关节角度等)和直线参数(指骨长度和手 掌尺度等)。

**两个环节：**初始参数估计(其中，3D人手模型都假设直线参数 是预知的，则关节角度问题都转换为逆运动学问题 （病态））和参数随时间的更新（常用的策略是卡尔 曼滤波和预测）。

**基于表观的手势模型：**1.模型视图序列；2.几何矩描述或者特征分解；3 模板节点的均值m和它们的方差v.通过在训 练 集上进行主成分分析（PCA）。4.平移运动参数 , 旋转运动参数 ,以及图像变形参数等等。

**手势识别：**把模型参数空间里的轨迹 (或点) 分类到该空间里某个子集的过程。（静态手势对应着模 型参数空间里一个点 ,而动态手势则对应着模型参数空间里的一条轨迹）。

**静态手势识别算法：**基于经典参数聚类技术的识别和基于非线性聚类技术的识别。

**动态手势：**时间及空间上下文.

**动态手势识别技术:**基于隐马尔可夫模型(Hidden Markov Models , HMM)的识别 , 基于动态时间规整 (Dy2namic Time Warping , DTW) 的识别 , 基于压缩时间轴的识别。

**结论：**基于3D手(臂)模型的手势模型适合于给所有手势建模,而基于表观的手势模型通常只试用于 给交流性手势。成功的手势识别策略应该考虑手势的时间-空间上下文,即考虑手势的语法规 则。语法规则既要反映手势的语言学特征,又要反映手势的空间特征。手势识别将增强现有的 人机交互模式 ,从而实现更直接、更自然、更和谐的人机接口,并且促成一些新的应用。

**张巧：**

1. **算法讨论：**

**快速排序：** 快速排序首先找到一个基准，假设以第一个元素作为基准（k），然后先从右向左搜索，如果发现比k小，则和k交换，然后从左向右搜索，如果发现比k大，则和k交换，一直到左边大于右边，此时k左边的都比它小，而右边的都比它大，此时k的位置就是排好序后应该在的位置，此时k将数组划分为左右两部分，可以递归采用该方法进行。快排的交换使排序成为不稳定的。

**Quick Sort（by python）**

import time

def quick\_sort(data\_list, i, j):

"""快速排序函数，入参为待排列表，待排起始位置与终点位置"""

start = i #start记录初始列表起始位置

end = j #end记录初始列表终点位置

k = data\_list[i] #k为阀值，默认为列表起点

while j != i:

while j != i: #从后向前遍历列表值与k值比较，第一个比k小的值赋给i位置

if data\_list[j] >= k : j -= 1

elif j == i : break #结束判断

else :

data\_list[i] = data\_list[j]

i += 1

break

while j != i: #再从前向后遍历列表值与k值比较，第一个比k大的值赋给j位置

if data\_list[i] <= k : i += 1

elif j == i : break #结束判断

else :

data\_list[j] = data\_list[i]

j -= 1

break

data\_list[i] = k #将k值赋值给其在列表的最终位置

if end - start == 1: #终止条件

return 0

if start != i and i != start + 1: #递归排序阀值左侧元素，且当其元素小于2时不做任何处理

quick\_sort(data\_list, start, i - 1)

if end != j and j != end - 1: #递归排序阀值右侧元素，且当其元素小于2时不做任何处理

quick\_sort(data\_list, i + 1, end)

def main():

"""快速排序程序入口"""

with open("./dig\_data.txt", "r+", encoding="utf-8") as fp:

#读取列表数据

content = fp.read()

data\_list = eval(content) #字符串转化为相应的变量

start\_time = time.time() #计时

quick\_sort(data\_list, 0, len(data\_list) - 1) #调用快排函数

print(time.time() - start\_time) #打印所耗时间

print(data\_list) #打印排完序的列表

if \_\_name\_\_ == "\_\_main\_\_":

main()

**堆排序：**堆排序是把数组看作堆，第i个结点的孩子结点为第2\*i+1和2\*i+2个结点（不超出数组长度前提下），堆排序的第一步是建堆，然后是取堆顶元素然后调整堆。建堆的过程是自底向上不断调整达成的，这样当调整某个结点时，其左节点和右结点已经是满足条件的，此时如果两个子结点不需要动，则整个子树不需要动，如果调整，则父结点交换到子结点位置，再以此结点继续调整。以大顶堆为例，建立好堆后堆顶元素为最大值，此时取堆顶元素即使堆顶元素和最后一个元素交换，最大的元素处于数组最后，此时调整小了一个长度的堆，然后再取堆顶和倒数第二个元素交换，依次类推，完成数据的非递减排序。堆排序的主要时间花在初始建堆期间，建好堆后，堆这种数据结构以及它奇妙的特征，使得找到数列中最大的数字这样的操作只需要O(1)的时间复杂度，维护需要logn的时间复杂度。堆排序不适宜于记录数较少的文件。

**Heap Sort（by python）**

import time

def big\_endian(arr, start, end):

root = start

while True:

child = root \* 2 + 1 # 左孩子

if child > end: # 孩子比最后一个节点还大 也就意味着最后一个叶子节点了 就得跳出去一次循环已经调整完毕

break

if child + 1 <= end and arr[child] < arr[child + 1]: # 为了始终让其跟子元素的较大值比较 如果右边大就左换右，左边大的话就默认

child += 1

if arr[root] < arr[child]: # 父节点小于子节点直接换位置 同时坐标也得换这样下次循环可以准确判断是否为最底层是不是调整完毕

arr[root], arr[child] = arr[child], arr[root]

root = child

else: # 父子节点顺序正常 直接过

break

def heap\_sort(arr):

# 无序区大根堆排序

first = len(arr) // 2 - 1

for start in range(first, -1, -1): # 从下到上，从右到左对每个节点进调整 循环得到非叶子节点

big\_endian(arr, start, len(arr) - 1) # 去调整所有的节点

for end in range(len(arr) - 1, 0, -1):

arr[0], arr[end] = arr[end], arr[0] # 顶部尾部互换位置

big\_endian(arr, 0, end - 1) # 重新调整子节点的顺序 从顶开始调整

return arr

def main():

#堆排序程序入口

with open("./dig\_data.txt", "r+", encoding="utf-8") as fp:

#读取列表数据

content = fp.read()

data\_list = eval(content) #字符串转化为相应的变量

start\_time = time.time()

print(heap\_sort(data\_list)) # 原地排序

print(time.time() - start\_time)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Heap Sort 1（by C++）**

#include <iostream>

#include <string>

#include <fstream>

#include <time.h>

#define SIZE 10000

using namespace std;

void AdjustDown(int \*a, int k,int len)

{

a[0] = a[k];

for (int i = 2 \* k; i <= len; i \*= 2)

{

if (i < len&&a[i] < a[i + 1])

i++;

if (a[0] >= a[i]) break;

else

{

a[k] = a[i];

k = i;

}

}

a[k] = a[0];

}

void BuildMaxHeap(int \*a,int len)

{

for (int i = len / 2; i > 0; i--)

{

AdjustDown(a, i, len);

}

}

void HeapSort(int \*a, int len)

{

BuildMaxHeap(a, len);

for (int j = len;j>0;j--)

{

printf("%d\t", a[1]);

int temp;

temp = a[j]; a[j] = a[1]; a[1] = temp;

AdjustDown(a, 1, j - 1);

}

}

**Heap Sort 2（by C++）**

int\* readData()

{

ifstream readFile("dig\_data.txt");

//ifstream readFile("test.txt");

string s;

int test[10001];

test[0] = 0;

static int i = 0;

const char \*aplit = ",";

char \*next\_token = NULL;

while (getline(readFile,s))

{

char \*c\_s = (char\*)s.c\_str();

char \*p = strtok\_s(c\_s, aplit, &next\_token);//strtok()

while (p)

{

int a = atoi(p); //char字符串转int

test[++i] = a;

p = strtok\_s(NULL, aplit, &next\_token);

}

}

return test;

}

int main()

{

int \*test = readData();

int aa[10001];

for (int i = 0; i < SIZE+1; i++)

{

aa[i] = \*test;

test++;

}

clock\_t start, end;

double cost;

start = clock();

HeapSort(aa, SIZE+1);

end = clock();

cout << "计算时间为：" <<(end-start)<<"毫秒";

cin.get();

cin.get();

}