南開大學
Nankai University

计算机学院

并行程序设计 **pthread** 作业

ANN-bp 的并行优化

姓名：林语盈

学号：2012174

专业：计算机科学与技术

2022 年 5 月 4 日

摘要

本文首先介绍期末大作业的选题与本次作业的选题。接下来，介绍算法的设计与实现。最后，介绍不同平台、不同数据规模下的实验和结果分析。

项目源代码链接:https://github.com/AldebaranL/Parallel-programming-Homework

关键字: ANN-bp; SIMD; pthread

目录

1 问题描述

1.1 期末选题

ANN，Artificial Network，人工神经网络，泛指由大量的处理单元 (神经元) 互相连接而形成的复杂网络结构，是对人脑组织结构和运行机制的某种抽象、简化和模拟。BP，Back Propagation，反向传播，是 ANN 网络的计算算法。

ANN 可以有很多应用场景，在期末大作业中，拟将其运用于简单的特征分类实际问题中，如根据花期等特征对植物的分类，环境污染照片的分类，或基于已知词嵌入的文本情感分类等。

- 问题输入: 训练样本及其标签数据矩阵
- 问题输出: 模型的参数矩阵，当输入新的训练样本，可用以计算其预测值。

ANN 的模型结构，如图1.1所示

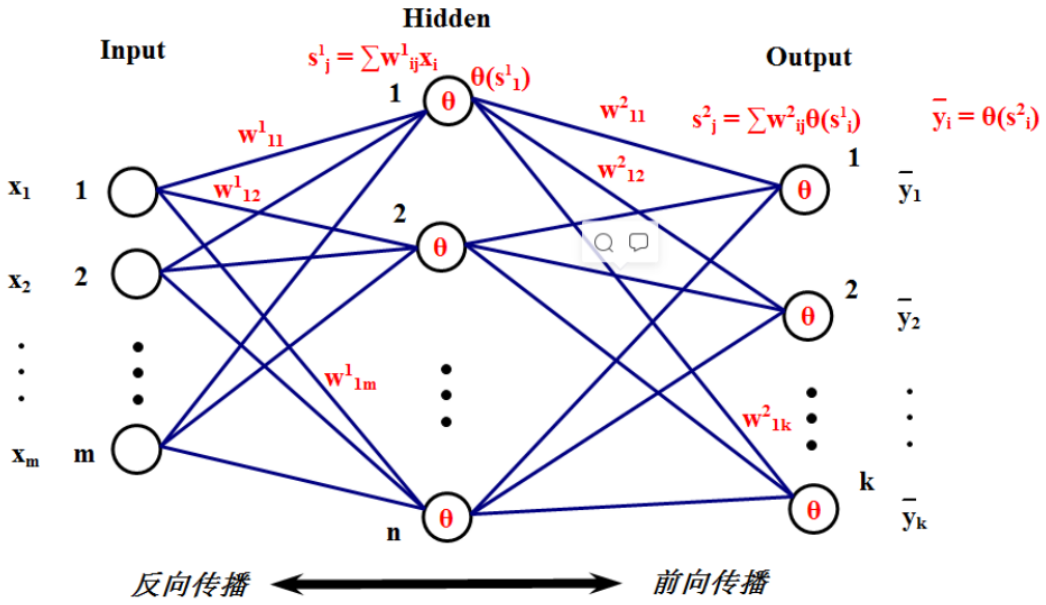

图 1.1: ANN 模型结构

其中，输入层维度为 m，输出层维度为 k，为了简化，假设仅有一个隐藏层，维度为 n。模型的参数分为两部分，W_1 为一个 m*n 的权值矩阵，用于计算输入层到隐藏层，W_2 为一个 n*k 的权值矩阵，用于计算隐藏层到输出层。此外，需要维护一个大小相同的导数矩阵，用于更新权值。

1.2 本次题目选题

本次实验将针对 ANN 计算过程中前向计算与反向传播的参数求导及其更新过程进行优化，这部分的算法描述如下。

1.2.1 前向传播

前向传播即由输入样本矩阵计算出对应的输出标签矩阵，其中每一层的计算即参数矩阵相乘，再加之偏置。

$$\begin{aligned} \boldsymbol{z}^{(l)} &= W^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{x}^{(l)} &= f\left(\boldsymbol{z}^{(l)}\right) \end{aligned} \tag{1}$$

Algorithm 1 前向传播

```
function PREDICT(Face)
    hiddenLayer ← W1×X + bias
    outputLayer ← f(W2 * hiddenLayer + bias)
    return outputLayer
end function
```

其中的 f 函数, 即激活函数, 可能有多种选择, 它的加入是为增加非线性性。常见的激活函数包括 sigmoid 和 tanh, 本实验中采用 sigmoid 函数。

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{2}$$

1.2.2 反向传播与参数更新

对于一个训练数据 $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$，首先计算其代价函数:

$$\begin{aligned} E_{(i)} &= \frac{1}{2}\left\|\boldsymbol{y}^{(i)} - \boldsymbol{o}^{(i)}\right\| \\ &= \frac{1}{2}\sum_{k=1}^{n_L}\left(y_k^{(i)} - o_k^{(i)}\right)^2 \end{aligned} \tag{3}$$

所有数据的平均代价函数为:

$$E_{total} = \frac{1}{N}\sum_{i=1}^{N} E_{(i)} \tag{4}$$

采用梯度下降法更新参数:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \mu\frac{\partial E_{\text{total}}}{\partial W^{(l)}} \\ &= W^{(l)} - \frac{\mu}{N}\sum_{i=1}^{N}\frac{\partial E_{(i)}}{\partial W^{(l)}} \\ \boldsymbol{b}^{(l)} &= \boldsymbol{b}^{(l)} - \mu\frac{\partial E_{\text{total}}}{\partial \boldsymbol{b}^{(l)}} \\ &= \boldsymbol{b}^{(l)} - \frac{\mu}{N}\sum_{i=1}^{N}\frac{\partial E_{(i)}}{\boldsymbol{b}^{(l)}} \end{aligned} \tag{5}$$

现计算其中的偏导数，记 $\delta_i^{(l)} \equiv \frac{\partial E}{\partial z_i^{(l)}}$，根据求导的链式法则，用向量形式可表示为:

$$\begin{aligned} \delta_i^{(L)} &= -\left(y_i - a_i^{(L)}\right) f'\left(z_i^{(L)}\right) && (1 \le i \le n_L) \\ \frac{\partial E}{\partial w_{ij}^{(L)}} &= \delta_i^{(L)} a_j^{(L-1)} && (1 \le i \le n_L, 1 \le j \le n_{L-1}) \end{aligned} \tag{6}$$

其中的 L 为神经网络的层数，为了简化，程序中取 L=2

Algorithm 2 反向传播训练参数

```
1: function TRAIN
2:     while 未达到迭代次数或收敛条件 do
3:         TrainVec()
4:         //更新 Weights 与 Bias
5:         for each layer reversely do
6:             Weights− = studyRate ∗ DeltaWeights
7:             Bias− = studyRate ∗ DeltaBias
8:             TrainVec()
9:         end for
10:    end while
11: end function
12:
13: function TRAINVEC
14:    //正向传播，计算 LayerOutput
15:    for each layer do
16:        LayerOutput ← sigmoid(LayerInput ∗ Weight + Bias)
17:    end for
18:    //求导，即计算 DeltaBias 与 DeltaWeights
19:    for each layer reversely do
20:        DeltaBias = (−0.1) ∗ (Label − LayerOutput) ∗ LayerOutput ∗ (1 − LayerOutput)
21:        DeltaWeights = DeltaBias ∗ LayerInput
22:    end for
23: end function
```

2 算法的设计与实现

2.1 实验分析与设计

2.1.1 程序正确性的判断

为判断程序的正确性，采用固定的输入矩阵，并观察 lossfunction 是否收敛与收敛速度。此外，由于神经网络的运算对浮点数精度的要求不高，全部采用 float 型进行存储，对于不同运算次序精度对结果的不同影响，在此问题中也无须考虑。

2.1.2 实验问题规模的设置

根据并行部分的特性，这里隐藏层层数均为 1，ANN 迭代次数 numEpoch 均为 128。这里，由于迭代仅等同于重复的函数调用，可看作问题时间的等倍放大，也使计时的差异更加明显，但因为其与并行部分无关，故在本次实验中保持为 128 不变。使用随机梯度下降，即使用每个样本数据更新参数。改变不同输入层结点数、输出层结点数、隐藏层结点数，测试程序运行时间。接下来，改变不同训练样本数，即训练样本类别数 * 每个类别训练样本数，测试程序运行时间。

2.1.3 程序时间复杂度分析

理论上来说，程序的时间复杂度为:

$$T = O(numEpoch*(numInput*numHidden1+numHidden1*numOutput)*numClass*numSamplePerClass) \tag{7}$$

理论上来说，4 个数的向量化最大带来 4 倍的加速比，加上数据打包解包等开销，加速比可能会略小于 4。n 个线程最多带来 n 倍的加速比，加上同步开销与线程初始化与创建的开销，加速比可能会略小于 n。

2.1.4 程序并行化的设计思路

为保证相同的问题规模，不采用 loss 函数是否收敛为终止条件，而设置特定的迭代次数做为终止条件。由于迭代的不同 minibatch 更新参数有前后依赖关系，并且在一个 batch 的训练中不同层前向传播与反向传播的层间也有依赖关系，故这里数据划分的方式采用在每层的矩阵运算中对矩阵的行或列划分，进行公式的计算。问题中共有三大类循环可以优化。

1. 前向传播的循环, 可以在同一层内按照 weights 矩阵的不同列进行划分，为避免数据访问的冲突，每个线程计算不同列，在每层计算后需要同步。
2. 计算导数 (更新量) 的循环，由于不存在访问冲突的问题，可以选择在同一层内按照 weights 矩阵的不同行或列进行划分，在每层计算后需要同步。
3. 更新 weights 的循环，类似的，在同一层内按照 weights 矩阵的不同列进行划分，在每层计算后需要同步。

由于仅在内层矩阵运算中进行任务的划分，这里使用静态的方式创建线程，在 train 函数初始即创建线程并通过信号量进入等待状态，在每次需要线程计算时从主线程唤醒子线程。同步的实现用到了信号量和 barrier，在每个网络层前向传播或反向传播结束时使用 barrier 进行同步，并用 sem 进行主、子线程的唤醒。
实现了两种并行化的方式，分别是将整个 train 写入 threadFunc、仅将 3 个计算部分写入 threadFunc 并通过 sem 唤醒，分别对比了这几种设计的运行时间。此外，在对列（行）划分时，分别使用连续、不连续列的划分方式，对比了这几种设计的运行时间。

2.1.5 pthread+SIMD

在 pthread 的基础上，将内层循环进行 sse 与 noen 优化，并测试运行实际进行对比。

2.2 朴素算法

由于完整代码过长，本报告中展示的代码仅为进行优化的核心代码。如上章所述，设计 ANN 类。

朴素算法，$ANN_2.cpp$

```
1 void ANN_2::train (const int num_sample, float** _trainMat, float** _labelMat)
2 {
3     printf ("begin training\n");
4     float thre = 1e-2;
5     float *avr_X = new float[num_each_layer[0]];
```

```
    float *avr_Y = new float[num_each_layer[num_layers + 1]];

    for (int epoch = 0; epoch < num_epoch; epoch++)
    {
        if (epoch % 50 == 0)
        {
            printf ("round%d:\n", epoch);
        }
        int index = 0;

        while (index < num_sample)
        {
            for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] = 0.0;
            for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                avr_Y[Y_i] = 0.0;

            for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
            {
                layers[num_layers]->delta[j] = 0.0;
            }

            for (int batch_i = 0; batch_i < batch_size && index < num_sample;
                batch_i++, index++)
                //默认batch_size=1, 即采用随机梯度下降法, 每次使用全部样本训练并更新参数
            {
                //前向传播
                predict (_trainMat[index]);
                //计算loss, 即最后一层的delta,即该minibatch中所有loss的平均值
                for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
                {
                    //均方误差损失函数,batch内取平均
                    layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j])
                        * layers[num_layers]->derivative_activation_function
                        (layers[num_layers]->output_nodes[j]);
                    //交叉熵损失函数
                    //layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j]);
                }
                // printf ("finish cal error\n");
                for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] +=
                    _trainMat[index][X_i];
                for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                    avr_Y[Y_i] += _labelMat[index][Y_i];
            }

            //delta在batch内取平均, avr_X、avr_Y分别为本个batch的输入、输出向量的平均值
            if (index % batch_size == 0)
            {
```

```
46                  for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
47                  {
48                      if (batch_size == 0) printf ("wrong!\n");
49                      layers[num_layers]->delta[j] /= batch_size;
50                      //for(int
                            i=0;i<5;i++)printf("delta=%f\n",layers[num_layers]->delta[i]);
51                      avr_Y[j] /= batch_size;
52                  }
53                  for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                        batch_size;
54
55              }
56              else
57              {
58                  for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
59                  {
60                      if (index % batch_size == 0) printf ("wrong!\n");
61                      layers[num_layers]->delta[j] /= (index % batch_size);
62                      avr_Y[j] /= (index % batch_size);
63                  }
64                  for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                        (index % batch_size);
65              }
66             // printf("index:%d\n",index);
67             //计算loss func, 仅用于输出
68              if ( index >= num_sample)
69              {
70                  static int tt = 0;
71                  float loss = 0.0;
72                  for (int t = 0; t < num_each_layer[num_layers + 1]; ++t)
73                  {
74                      loss += (layers[num_layers]->output_nodes[t] - avr_Y[t]) *
                            (layers[num_layers]->output_nodes[t] - avr_Y[t]);
75                  }
76                  printf ("第%d次训练: %0.12f\n", tt++, loss);
77              }
78
79              //反向传播更新参数
80              back_propagation (avr_X, avr_Y);
81              //printf ("finish bp with index:%d\n",index);
82          }
83
84          // display();
85      }
86      printf ("finish training\n");
87      delete[]avr_X;
88      delete[]avr_Y;
89  }
90
```

```
91  void ANN_2::predict (float* in)
92  {
93      layers[0]->_forward (in);
94      for (int i = 1; i <= num_layers; i++)
95      {
96          layers[i]->_forward (layers[i - 1]->output_nodes);
97      }
98  }
99
100 void ANN_2::back_propagation (float* X, float * Y)
101 {
102     //计算每层的delta,行访问优化
103     for (int i = num_layers - 1; i >= 0; i--)
104     {
105         float *error = new float[num_each_layer[i + 1]];
106         for (int j = 0; j < num_each_layer[i + 1]; j++) error[j] = 0.0;
107         for (int k = 0; k < num_each_layer[i + 2]; k++)
108         {
109             for (int j = 0; j < num_each_layer[i + 1]; j++)
110             {
111                 error[j] += layers[i + 1]->weights[k][j] * layers[i + 1]->delta[k];
112             }
113         }
114         for (int j = 0; j < num_each_layer[i + 1]; j++)
115         {
116             layers[i]->delta[j] = error[j] *
                    layers[num_layers]->derivative_activation_function
                    (layers[i]->output_nodes[j]);
117         }
118         delete[] error;
119     }
120
121     //反向传播, weights和bias更新
122     for (int k = 0; k < num_each_layer[1]; k++)
123     {
124         for (int j = 0; j < num_each_layer[0]; j++)
125         {
126             layers[0]->weights[k][j] -= study_rate * X[j] * layers[0]->delta[k];
127         }
128         layers[0]->bias[k] -= study_rate * layers[0]->delta[k];
129     }
130     for (int i = 1; i <= num_layers; i++)
131     {
132         for (int k = 0; k < num_each_layer[i + 1]; k++)
133         {
134             for (int j = 0; j < num_each_layer[i]; j++)
135             {
136                 layers[i]->weights[k][j] -= study_rate * layers[i -
                        1]->output_nodes[j] * layers[i]->delta[k];
```

```
137             }
138             layers[i]->bias[k] -= study_rate * layers[i]->delta[k];
139         }
140     }
141 }
```

朴素算法，Layer.cpp

```
1  void Layer::_forward (float * input_nodes)
2  {
3      for (int i = 0; i < num_output; i++)
4      {
5          output_nodes[i] = 0.0;
6          for (int j = 0; j < num_input; j++)
7          {
8              output_nodes[i] += weights[i][j] * input_nodes[j];
9          }
10         output_nodes[i] += bias[i];
11         output_nodes[i] = activation_function (output_nodes[i]);
12         if(output_nodes[i]==NAN) printf("!!!!!!!!");
13     }
14 }
```

2.3 pthread 信号量唤醒

pthread，使用静态的创建方式，仅将 3 个计算部分写入 threadFunc 并通过 sem 唤醒。

pthread 信号量唤醒

```
1  void* ANN_pthread::threadFunc_sem (void *param)
2  {
3      /* 这是仅将计算部分纳入的threadFunc函数，被train调用
4       * 对前向传播和反向传播每层的矩阵运算分配给不同线程
5       * 注意总体的循环要与主线程保持同步，在需要计算的部分被主线程唤醒（通过sem）
6       */
7      threadParam_t *p = (threadParam_t*) param;
8      int t_id = p -> t_id;
9      ANN_pthread * class_p = p->class_pointer;
10
11     for (int epoch = 0; epoch < class_p->num_epoch; epoch++)
12     {
13         int sample_index = 0;
14         while (sample_index < NUM_SAMPLE)
15         {
16             for (int batch_i = 0; batch_i < class_p->batch_size; batch_i++)
17             {
18                 if (sample_index >= NUM_SAMPLE)
19                 {
20                     break;
```

```
21                  }
22                  //前向传播
23
24                  sem_wait (& (sem_before_fw[t_id]) );
                        //阻塞,等待主线程（操作自己专属的信号量）
25
26                  int max_n = class_p->num_each_layer[1];
27
28                  //按weights矩阵的行进行划分, 每个线程执行连续的一部分行, 避免class_p->layers[0]-
29                  int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
30                  for (int i = t_id * my_n; i < std::min (max_n, (t_id + 1) *my_n);
                        i += 1)
31                  {
32                      class_p->layers[0]->output_nodes[i] = 0.0;
33
34                      for(int j = 0; j < class_p->num_each_layer[0]; j += 1)
35                      {
36
37                          class_p->layers[0]->output_nodes[i] +=
                                class_p->layers[0]->weights[i][j] *
                                p->sampleMat[sample_index][j];
38
39                      }
40                      class_p->layers[0]->output_nodes[i] +=
                            class_p->layers[0]->bias[i];
41                      class_p->layers[0]->output_nodes[i] =
                            class_p->layers[0]->activation_function
                            (class_p->layers[0]->output_nodes[i]);
42                  }
43                  for (int layers_i = 1; layers_i <= class_p->num_layers; layers_i++)
44                  {
45
46                      int max_n = class_p->num_each_layer[layers_i + 1];
47
48                      //按weights矩阵的行进行划分
49                      int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                            max_n / NUM_THREADS + 1;
50                      for (int i = t_id * my_n; i < std::min (max_n, (t_id + 1)
                            *my_n); i += 1)
51                      {
52                          class_p->layers[layers_i]->output_nodes[i] = 0.0;
53
54                          for (int j = 0; j < class_p->num_each_layer[layers_i]; j
                                += 1)
55                          {
56                              class_p->layers[layers_i]->output_nodes[i] +=
                                    class_p->layers[layers_i]->weights[i][j] *
                                    class_p->layers[layers_i - 1]->output_nodes[j];
```

```
57                      }
58                      class_p->layers[layers_i]->output_nodes[i] +=
                            class_p->layers[layers_i]->bias[i];
59                      class_p->layers[layers_i]->output_nodes[i] =
                            class_p->layers[layers_i]->activation_function
                            (class_p->layers[layers_i]->output_nodes[i]);
60                  }
61              }
62              sem_post (&sem_main_after_fw);//唤醒主线程
63              sample_index++;
64          }
65
66          //反向传播, 计算每层的delta
67          sem_wait (&sem_before_bp[t_id]);
                //阻塞, 等待主线程 (操作自己专属的信号量)
68          for (int i = class_p->num_layers - 1; i >= 0; i--)
69          {
70              float *error = new float[class_p->num_each_layer[i + 1]];
71              for (int j = t_id; j < class_p->num_each_layer[i + 1]; j +=
                    NUM_THREADS) error[j] = 0.0;
72              for (int k = 0; k < class_p->num_each_layer[i + 2]; k++)
73              {
74
75                  int max_n = class_p->num_each_layer[i + 1];
76                  //按weights矩阵的列进行划分, 也可按照k (按weights矩阵的行) 进行划分
77                  int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
78                  for (int j = t_id * my_n; j < std::min (max_n, (t_id + 1)
                        *my_n); j += 1)
79                  {
80                      error[j] += class_p->layers[i + 1]->weights[k][j] *
                            class_p->layers[i + 1]->delta[k];
81                  }
82              }
83              int max_n = class_p->num_each_layer[i + 1];
84              int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                    max_n / NUM_THREADS + 1;
85              for (int j = t_id * my_n; j < std::min (max_n, (t_id + 1) *my_n);
                    j += 1)
86              {
87                  class_p->layers[i]->delta[j] = error[j] *
                        class_p->layers[i]->derivative_activation_function
                        (class_p->layers[i]->output_nodes[j]);
88              }
89              delete[] error;
90          }
91
92          //反向传播, weights和bias更新
93          //按weights矩阵的行进行划分
```

```
94              int max_n = class_p->num_each_layer[1];
95              int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS : max_n /
                    NUM_THREADS + 1;
96              for (int k = t_id * my_n; k < std::min (max_n, (t_id + 1) *my_n); k +=
                    1)
97              {
98                  for (int j = 0; j < class_p->num_each_layer[0]; j += 1)
99                  {
100                     class_p->layers[0]->weights[k][j] -= class_p->study_rate *
                            p->sampleMat[sample_index - 1][j] *
                            class_p->layers[0]->delta[k];
101                 }
102                 class_p->layers[0]->bias[k] -=  class_p->study_rate *
                        class_p->layers[0]->delta[k];
103             }
104             for (int i = 1; i <= class_p->num_layers; i++)
105             {
106                 int max_n = class_p->num_each_layer[i + 1];
107                 int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
108                 for (int k = t_id * my_n; k < std::min (max_n, (t_id + 1) *my_n);
                        k += 1)
109                 {
110                     for (int j = 0; j < class_p->num_each_layer[i]; j += 1)
111                     {
112                         class_p->layers[i]->weights[k][j] -= class_p->study_rate *
                                class_p->layers[i - 1]->output_nodes[j] *
                                class_p->layers[i]->delta[k];
113                     }
114                     class_p->layers[i]->bias[k] -=  class_p->study_rate *
                            class_p->layers[i]->delta[k];
115                 }
116             }
117             //唤醒主线程
118             sem_post (&sem_main_after_bp);
119         }
120     }
121     //线程退出
122     pthread_exit (NULL);
123 }
124
125 void ANN_pthread::train (const int _num_sample, float** _trainMat, float**
        _labelMat)
126 {
127     //构建线程输入参数
128     creat_params();
129     for (int i = 0; i < NUM_THREADS; i++)
130     {
131         params[i].class_pointer = this;
```

```
132             for (int j = 0; j < _num_sample; j++)
133             {
134                 for (int k = 0; k < num_each_layer[0]; k++)
135                     params[i].sampleMat[j][k] = _trainMat[j][k];
136             }
137         }
138 
139         //信号量初始化,赋值为0即开始就进入等待状态
140         int res = 0;
141         for (int i = 0; i < NUM_THREADS; i++)
142         {
143             res += sem_init (& (sem_before_fw[i]), 0, 0);
144             res += sem_init (& (sem_before_bp[i]), 0, 0);
145         }
146         res += sem_init (&sem_main_after_fw, 0, 0);
147         res += sem_init (&sem_main_after_bp, 0, 0);
148         if (res != 0) printf ("init sem failed!\n");
149 
150         //创建线程,静态
151         for (int t_id = 0; t_id < NUM_THREADS; t_id++)
152         {
153             params[t_id].t_id = t_id;
154             //使用类方法作为线程函数需进行强制转换, test为另外声明的静态函数, 指向threadFunc_sem
155             pthread_create (&handles[t_id], NULL, &test, (void*) & (params[t_id]) );
156         }
157 
158         float thre = 1e-2;
159 
160         for (int epoch = 0; epoch < num_epoch; epoch++)
161         {
162             if (epoch % 50 == 0)
163                 printf ("round%d:\n", epoch);
164             int sample_index = 0;
165             while (sample_index < _num_sample)
166             {
167                 for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
168                 {
169                     layers[num_layers]->delta[j] = 0.0;
170                 }
171 
172                 for (int batch_i = 0; batch_i < batch_size; batch_i++)
                        //默认batch_size=1, 即采用随机梯度下降法, 每次使用全部样本训练并更新参数
173                 {
174                     if (sample_index >= _num_sample) break;
175                     //前向传播, 使用sem唤醒子线程
176                     for (int t_i = 0; t_i < NUM_THREADS; t_i++)
177                     {
178                         sem_post (& (sem_before_fw[t_i]) );
179                     }
```

```
180                 for (int t_i = 0; t_i < NUM_THREADS; t_i++)
181                 {
182                     sem_wait (&sem_main_after_fw);
183                 }
184                 //计算loss, 即最后一层的delta,即该minibatch中所有loss的平均值
185                 for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
186                 {
187                     //均方误差损失函数
188                     layers[num_layers]->delta[j] +=
                            (layers[num_layers]->output_nodes[j] -
                            _labelMat[sample_index][j]) *
                            layers[num_layers]->derivative_activation_function
                            (layers[num_layers]->output_nodes[j]);
189                     //交叉熵损失函数
190                     //layers[num_layers]->delta[j] +=
                            (layers[num_layers]->output_nodes[j] -
                            _labelMat[sample_index][j]);
191                 }
192                 sample_index++;
193             }
194             //batch内取平均
195             if (sample_index % batch_size == 0)
196             {
197                 for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
198                 {
199                     layers[num_layers]->delta[j] /= batch_size;
200                 }
201             }
202             else
203             {
204                 for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
205                 {
206                     layers[num_layers]->delta[j] /= sample_index % batch_size;
207                 }
208             }
209             //反向传播并更新参数, 使用sem唤醒子线程
210             for (int t_i = 0; t_i < NUM_THREADS; t_i++)
211             {
212                 sem_post (&sem_before_bp[t_i]);
213             }
214             for (int t_i = 0; t_i < NUM_THREADS; t_i++)
215             {
216                 sem_wait (&sem_main_after_bp);
217             }
218         }
219     }
220     for (int t_id = 0; t_id < NUM_THREADS; t_id++)
221     {
222         pthread_join (handles[t_id], NULL);
```

```
223     }
224     printf ("finish pthread_join\n");
225 
226     //delete线程输入参数
227     delet_params();
228 }
```

2.4 pthread 全部写入 threadFunc

train 全部写入 threadFunc，使用 barrier 在每次计算完一层后进行同步。

train 全部写入 threadFunc

```
1  void* ANN_pthread::threadFunc_barrier (void *param)
2  {
3      /* 这是仅将全部train纳入的threadFunc函数，被train_2调用
4       * 数据划分与threadFunc_sem函数相同
5       * 在需要同步的位置使用barrier进行同步
6       */
7      threadParam_t *p = (threadParam_t*) param;
8      int t_id = p -> t_id;
9      ANN_pthread * class_p = p->class_pointer;
10 
11     float thre = 1e-2;
12 
13     for (int epoch = 0; epoch < class_p->num_epoch; epoch++)
14     {
15         if (epoch % 50 == 0)
16             printf ("round%d:\n", epoch);
17         int sample_index = 0;
18         while (sample_index < NUM_SAMPLE)
19         {
20             if (t_id == 0)
21                 for (int j = 0; j < class_p->num_each_layer[class_p->num_layers +
                       1]; j++)
22                 {
23                     class_p->layers[class_p->num_layers]->delta[j] = 0.0;
24                 }
25 
26             for (int batch_i = 0; batch_i < class_p->batch_size; batch_i++)
27             {
28                 if (sample_index >= NUM_SAMPLE) break;
29                 //前向传播
30                 int max_n = class_p->num_each_layer[1];
31                 int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                       max_n / NUM_THREADS + 1;
32                 for (int i = t_id * my_n; i < std::min (max_n, (t_id + 1) *my_n);
                       i += 1)
33                 {
```

```
34
35                  class_p->layers[0]->output_nodes[i] = 0.0;
36                  int max_n = class_p->num_each_layer[0];
37                  for (int j = 0; j < class_p->num_each_layer[0]; j += 1)
38                  {
39                      class_p->layers[0]->output_nodes[i] +=
                            class_p->layers[0]->weights[i][j] *
                            p->sampleMat[sample_index][j];
40                  }
41                  class_p->layers[0]->output_nodes[i] +=
                        class_p->layers[0]->bias[i];
42                  class_p->layers[0]->output_nodes[i] =
                        class_p->layers[0]->activation_function
                        (class_p->layers[0]->output_nodes[i]);
43              }
44               pthread_barrier_wait (&barrier_fw[0]);
45
46              for (int layers_i = 1; layers_i <= class_p->num_layers; layers_i++)
47              {
48
49                  int max_n = class_p->num_each_layer[layers_i + 1];
50                  int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
51                  for (int i = t_id * my_n; i < std::min (max_n, (t_id + 1)
                        *my_n); i += 1)
52                  {
53                      class_p->layers[layers_i]->output_nodes[i] = 0.0;
54                      for (int j = 0; j < class_p->num_each_layer[layers_i]; j
                            += 1)
55                      {
56                          class_p->layers[layers_i]->output_nodes[i] +=
                                class_p->layers[layers_i]->weights[i][j] *
                                class_p->layers[layers_i - 1]->output_nodes[j];
57                      }
58                      class_p->layers[layers_i]->output_nodes[i] +=
                            class_p->layers[layers_i]->bias[i];
59                      class_p->layers[layers_i]->output_nodes[i] =
                            class_p->layers[layers_i]->activation_function
                            (class_p->layers[layers_i]->output_nodes[i]);
60                  }
61                  //每层均需同步
62                   pthread_barrier_wait (&barrier_fw[layers_i]);
63              }
64              sample_index++;
65          }
66          //计算loss, 即最后一层的delta,即该minibatch中所有loss的平均值, 由0号线程执行此部分
67          if (t_id == 0)
68          {
69              for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
```

```
70                  {
71                      //均方误差损失函数
72                      layers[num_layers]->delta[j] +=
                            (layers[num_layers]->output_nodes[j] -
                            LABEL_MAT[sample_index][j]) *
                            layers[num_layers]->derivative_activation_function
                            (layers[num_layers]->output_nodes[j]);
73
74                  }
75                  sample_index++;
76                  if (sample_index % class_p->batch_size == 0)
77                  {
78                      for (int j = 0; j <
                            class_p->num_each_layer[class_p->num_layers + 1]; j++)
79                      {
80                          class_p->layers[class_p->num_layers]->delta[j] /=
                                class_p->batch_size;
81                      }
82                  }
83                  else
84                  {
85                      for (int j = 0; j <
                            class_p->num_each_layer[class_p->num_layers + 1]; j++)
86                      {
87                          class_p->layers[num_layers]->delta[j] /= sample_index %
                                class_p->batch_size;
88                      }
89                  }
90              }
91              //同步, 等待0号线程完成
92              pthread_barrier_wait (&barrier_before_bp);
93
94              //反向传播更新参数
95              for (int i = class_p->num_layers - 1; i >= 0; i--)
96              {
97                  float *error = new float[class_p->num_each_layer[i + 1]];
98                  for (int j = t_id; j < class_p->num_each_layer[i + 1]; j +=
                        NUM_THREADS) error[j] = 0.0;
99                  for (int k = 0; k < class_p->num_each_layer[i + 2]; k++)
100                 {
101                     int max_n = class_p->num_each_layer[i + 1];
102                     int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                            max_n / NUM_THREADS + 1;
103                     for (int j = t_id * my_n; j < std::min (max_n, (t_id + 1)
                            *my_n); j += 1)
104                     {
105                         error[j] += class_p->layers[i + 1]->weights[k][j] *
                                class_p->layers[i + 1]->delta[k];
106                     }
```

```
107                 }
108                 int max_n = class_p->num_each_layer[i + 1];
109                 int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
110                 for (int j = t_id * my_n; j < std::min (max_n, (t_id + 1) *my_n);
                        j += 1)
111                 {
112                     class_p->layers[i]->delta[j] = error[j] *
                            class_p->layers[i]->derivative_activation_function
                            (class_p->layers[i]->output_nodes[j]);
113                 }
114                 delete[] error;
115                  //每层均需同步
116                   pthread_barrier_wait (&barrier_delta[i]);
117             }
118
119             //反向传播, weights和bias更新
120
121             int max_n = class_p->num_each_layer[1];
122             int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS : max_n /
                    NUM_THREADS + 1;
123             for (int k = t_id * my_n; k < std::min (max_n, (t_id + 1) *my_n); k +=
                    1)
124             {
125                 for (int j = 0; j < class_p->num_each_layer[0]; j += 1)
126                 {
127                     class_p->layers[0]->weights[k][j] -= class_p->study_rate *
                            p->sampleMat[sample_index - 1][j] *
                            class_p->layers[0]->delta[k];
128                 }
129                 class_p->layers[0]->bias[k] -=  class_p->study_rate *
                        class_p->layers[0]->delta[k];
130             }
131             pthread_barrier_wait (&barrier_bp[0]);
132             for (int i = 1; i <= class_p->num_layers; i++)
133             {
134                 int max_n = class_p->num_each_layer[i + 1];
135                 int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
136                 for (int k = t_id * my_n; k < std::min (max_n, (t_id + 1) *my_n);
                        k += 1)
137                 {
138                     for (int j = 0; j < class_p->num_each_layer[i]; j += 1)
139                     {
140                         class_p->layers[i]->weights[k][j] -= class_p->study_rate *
                                class_p->layers[i - 1]->output_nodes[j] *
                                class_p->layers[i]->delta[k];
141                     }
142                     class_p->layers[i]->bias[k] -=  class_p->study_rate *
```

```
                        class_p->layers[i]->delta[k];
143                 }
144                  //每层均需同步
145                 pthread_barrier_wait (&barrier_bp[i]);
146             }
147         }
148     }
149     pthread_exit (NULL);
150
151 }
152 void ANN_pthread::train_2 (const int _num_sample, float** _trainMat, float**
        _labelMat)
153 {
154     //创建线程输入参数
155     creat_params();
156     for (int i = 0; i < NUM_THREADS; i++)
157     {
158         //输入当前类的指针, 否则不为本对象!
159         params[i].class_pointer = this;
160         for (int j = 0; j < _num_sample; j++)
161         {
162             for (int k = 0; k < num_each_layer[0]; k++)
163                 params[i].sampleMat[j][k] = _trainMat[j][k];
164         }
165     }
166     //barrier初始化
167     for(int layer_i=0;layer_i<num_layers;layer_i++){
168         pthread_barrier_init (&barrier_fw[layer_i], NULL, NUM_THREADS);
169         pthread_barrier_init (&barrier_bp[layer_i], NULL, NUM_THREADS);
170         pthread_barrier_init (&barrier_delta[layer_i], NULL, NUM_THREADS);
171     }
172     pthread_barrier_init (&barrier_before_bp, NULL, NUM_THREADS);
173     printf("finish init");
174
175     //创建线程,静态
176     for (int t_id = 0; t_id < NUM_THREADS; t_id++)
177     {
178         params[t_id].t_id = t_id;
179         //使用类方法作为线程函数需进行强制转换, func_threadFunc_barrier为另外声明的静态函数
180         pthread_create (&handles[t_id], NULL, &func_threadFunc_barrier, (void*) &
                (params[t_id]) );
181     }
182
183     //等待线程结束
184     for (int t_id = 0; t_id < NUM_THREADS; t_id++)
185     {
186         pthread_join (handles[t_id], NULL);
187     }
188     delet_params();
```

```
189 }
```

2.5 pthread+SIMD

在 pthread 的 sem 唤醒的基础上将内层循环进行四路向量化，这里展示 sse 的代码，neon 与之同理。

train 全部写入 threadFunc

```
1  void* ANN_pthread::threadFunc_sem_SIMD (void *param)
2  {
3      threadParam_t *p = (threadParam_t*) param;
4      int t_id = p -> t_id;
5      ANN_pthread * class_p = p->class_pointer;
6
7      for (int epoch = 0; epoch < class_p->num_epoch; epoch++)
8      {
9          int sample_index = 0;
10         while (sample_index < NUM_SAMPLE)
11         {
12             for (int batch_i = 0; batch_i < class_p->batch_size; batch_i++)
13             {
14                 if (sample_index >= NUM_SAMPLE)
15                 {
16                     break;
17                 }
18                 sem_wait (& (sem_before_fw[t_id]) ); //
                       阻塞,等待主线程（操作自己专属的信号量）
19                 for (int i = 0; i < class_p->num_each_layer[1]; i++)
20                 {
21                     class_p->layers[0]->output_nodes[i] = 0.0;
22                     int max_n = class_p->num_each_layer[0];
23
24                     __m128 ans = _mm_setzero_ps();
25                     for (int j = t_id * max_n / NUM_THREADS; j < min (max_n, (t_id
                           + 1) *max_n / NUM_THREADS); j += 4)
26                     {
27                         __m128 t1, t2;
28                         //将内部循环改为4位的向量运算
29                         t1 = _mm_loadu_ps (class_p->layers[0]->weights[i] + j);
30                         t2 = _mm_loadu_ps (p->sampleMat[sample_index] + j);
31                         t1 = _mm_mul_ps (t1, t2);
32
33                         ans = _mm_add_ps (ans, t1);
34                     }
35                     // 4个局部和相加
36                     ans = _mm_hadd_ps (ans, ans);
37                     ans = _mm_hadd_ps (ans, ans);
38                     //标量存储
```

```
39                  float z;
40                  _mm_store_ss (&z, ans);
41                  class_p->layers[0]->output_nodes[i] += z;
42                  class_p->layers[0]->output_nodes[i] +=
                        class_p->layers[0]->bias[i];
43                  class_p->layers[0]->output_nodes[i] =
                        class_p->layers[0]->activation_function
                        (class_p->layers[0]->output_nodes[i]);
44              }
45              for (int layers_i = 1; layers_i <= class_p->num_layers; layers_i++)
46              {
47
48                  for (int i = 0; i < class_p->num_each_layer[layers_i + 1]; i++)
49                  {
50                      class_p->layers[layers_i]->output_nodes[i] = 0.0;
51                      int max_n = class_p->num_each_layer[layers_i];
52                      __m128 ans = _mm_setzero_ps();
53                      for (int j = t_id * max_n / NUM_THREADS; j < min (max_n,
                            (t_id + 1) *max_n / NUM_THREADS); j += 4)
54                      {
55                          __m128 t1, t2;
56                          //将内部循环改为4位的向量运算
57                          t1 = _mm_loadu_ps
                                (class_p->layers[layers_i]->weights[i] + j);
58                          t2 = _mm_loadu_ps (p->sampleMat[sample_index] + j);
59                          t1 = _mm_mul_ps (t1, t2);
60
61                          ans = _mm_add_ps (ans, t1);
62                      }
63                      // 4个局部和相加
64                      ans = _mm_hadd_ps (ans, ans);
65                      ans = _mm_hadd_ps (ans, ans);
66                      //标量存储
67                      float z;
68                      _mm_store_ss (&z, ans);
69                      class_p->layers[layers_i]->output_nodes[i] += z;
70
71                      class_p->layers[layers_i]->output_nodes[i] +=
                            class_p->layers[layers_i]->bias[i];
72                      class_p->layers[layers_i]->output_nodes[i] =
                            class_p->layers[layers_i]->activation_function
                            (class_p->layers[layers_i]->output_nodes[i]);
73                  }
74              }
75              sem_post (&sem_main_after_fw);
76              sample_index++;
77          }
78          sem_wait (&sem_before_bp[t_id]); //
                阻塞, 等待主线程 (操作自己专属的信号量)
```

```
79              //计算每层的delta
80              for (int i = class_p->num_layers - 1; i >= 0; i--)
81              {
82                  float *error = new float[class_p->num_each_layer[i + 1]];
83                  for (int j = t_id; j < class_p->num_each_layer[i + 1]; j +=
                        NUM_THREADS) error[j] = 0.0;
84                  for (int k = 0; k < class_p->num_each_layer[i + 2]; k++)
85                  {
86                      int max_n = class_p->num_each_layer[i + 1];
87                      __m128 t3 = _mm_set1_ps (class_p->layers[i + 1]->delta[k]);
88                      for (int j = t_id * max_n / NUM_THREADS; j < min (max_n, (t_id
                            + 1) *max_n / NUM_THREADS); j += 4)
89                      {
90                          __m128 t1, t2;
91                          t1 = _mm_loadu_ps (error + j);
92                          t2 = _mm_loadu_ps (class_p->layers[i + 1]->weights[k] + j);
93                          //product=t1*t2*t3, 向量对位相乘
94                          t2 = _mm_mul_ps (t3, t2);
95                          t1 = _mm_add_ps (t1, t2);
96                          _mm_storeu_ps (error + j, t1);
97                      }
98                  }
99                  int max_n = class_p->num_each_layer[i + 1];
100                 int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS :
                        max_n / NUM_THREADS + 1;
101                 for (int j = t_id * my_n; j < std::min (max_n, (t_id + 1) *my_n);
                        j += 1)
102                 {
103                     class_p->layers[i]->delta[j] = error[j] *
                            class_p->layers[i]->derivative_activation_function
                            (class_p->layers[i]->output_nodes[j]);
104                 }
105                 delete[] error;
106             }
107             //反向传播, weights和bias更新
108             for (int k = 0; k < class_p->num_each_layer[1]; k++)
109             {
110                 int max_n = class_p->num_each_layer[0];
111                 __m128 sr = _mm_set1_ps (class_p->study_rate);
112                 __m128 t2 = _mm_set1_ps (class_p->layers[0]->delta[k]);
113                 for (int j = t_id * max_n / NUM_THREADS; j < min (max_n, (t_id +
                        1) *max_n / NUM_THREADS); j += 4)
114                 {
115                     __m128 t1, t3, product;
116                     t1 = _mm_loadu_ps (class_p->layers[0]->weights[k] + j);
117                     t3 = _mm_loadu_ps (p->sampleMat[sample_index - 1] + j);
118                     //product=sr*t2*t3, 向量对位相乘
119                     product = _mm_mul_ps (t3, t2);
120                     product = _mm_mul_ps (product, sr);
```

```
                    t1 = _mm_sub_ps (t1, product);
                    _mm_storeu_ps (class_p->layers[0]->weights[k] + j, t1);
                }
                class_p->layers[0]->bias[k] -= class_p->study_rate *
                    class_p->layers[0]->delta[k];
            }
            for (int i = 1; i <= class_p->num_layers; i++)
            {
                for (int k = 0; k < class_p->num_each_layer[i + 1]; k++)
                {
                    int max_n = class_p->num_each_layer[i];
                    __m128 sr = _mm_set1_ps (class_p->study_rate);
                    __m128 t2 = _mm_set1_ps (class_p->layers[i]->delta[k]);
                    for (int j = t_id * max_n / NUM_THREADS; j < min (max_n, (t_id
                        + 1) *max_n / NUM_THREADS); j += 4)
                    {
                        __m128 t1, t3, product;
                        t1 = _mm_loadu_ps (class_p->layers[i]->weights[k] + j);
                        t3 = _mm_loadu_ps (class_p->layers[i - 1]->output_nodes +
                            j);
                        //product=sr*t2*t3, 向量对位相乘
                        product = _mm_mul_ps (t3, t2);
                        product = _mm_mul_ps (product, sr);
                        t1 = _mm_sub_ps (t1, product);

                        _mm_storeu_ps (class_p->layers[i]->weights[k] + j, t1);
                    }
                    class_p->layers[i]->bias[k] -= class_p->study_rate *
                        class_p->layers[i]->delta[k];
                }
            }
            sem_post (&sem_main_after_bp);
        }
    }
    pthread_exit (NULL);
}
```

3 实验和结果分析

因为笔记本为 AMD 架构，无法使用 Vtune 工具。对程序不同部分进行了较细粒度的计时对比，分析占用时间的关键循环。测试了不同算法设计、不同规模下、不同平台下的程序运行时间。

3.1 不同问题规模下串行算法与并行算法的对比

不同问题规模详细的定义如表1所示。其中，numInput、numOutput、numHidden1、numLayers、numClass、numSamplePerClass、numEpoch、batchSize 依次为输入层结点数、输出层结点数、隐藏层结点数、隐藏层层数、训练样本类别数、每个类别训练样本数、ANN 迭代次数、一次参数更新使用

表 1: 问题规模

	size1	size2	size3	size4	size5
numInput	1024	512	256	128	64
numOutput	1024	512	256	128	64
numHidden1	1024	512	256	128	64
numLayers	1	1	1	1	1
numClass	4	4	4	4	4
numSamplePerClass	16	16	16	16	16
numEpoch	128	128	128	128	128
batchSize	1	1	1	1	1

表 2: 不同问题规模下的程序运行时间 (单位：s)

	size1	size2	size3	size4	size5
串行	539.8456107	128.4103332	30.40502581	7.82814822	4.33231865
SIMD4 维向量	222.4107995	55.93146143	14.5320403	3.91522879	2.09871414
pthread_sem 4 线程	165.5424523	40.28941561	10.21026395	2.87439668	1.88398819
pthread+SIMD	92.33459881	24.94997974	7.07640235	2.23599507	1.39752083

的样本数目。在华为鲲鹏平台上的实际运行时间如表2所示。其对应的加速比如图3.2所示。

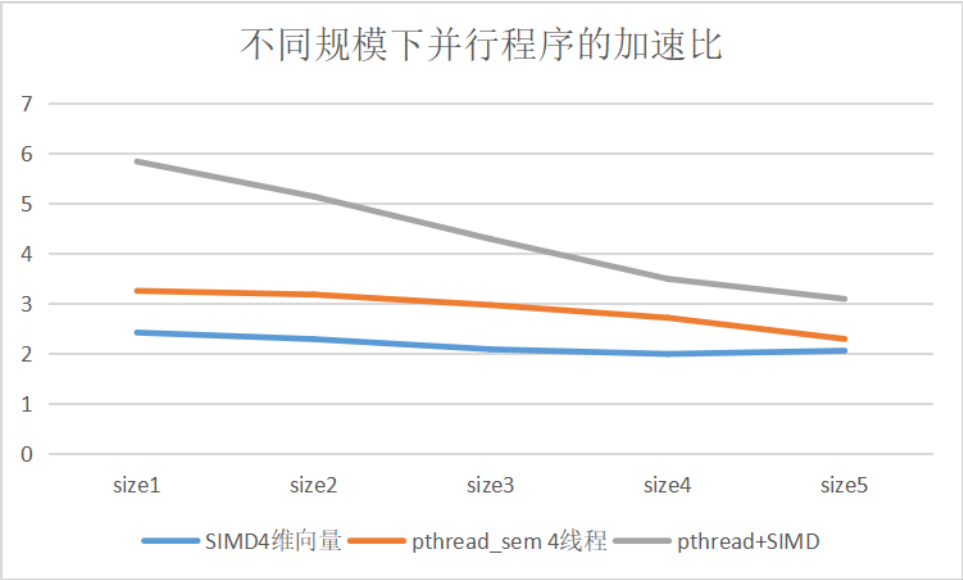

图 3.2: 不同问题规模下的并行程序加速比

根据并行部分的特性，这里隐藏层层数均为 1，ANN 迭代次数均为 128，由于迭代仅等同于重复的函数调用，可看作问题时间的等倍放大，也使计时的差异更加明显，但因为其与并行部分无关，故在本次实验中保持为 128 不变。测试了不同输入层结点数、输出层结点数、隐藏层结点数的程序运行时间。

理论上来说，程序的时间复杂度为:

$$T = O(t*(numInput*numHidden1+numHidden1*numOutput)*numClass*numSamplePerClass) \tag{8}$$

其中对总训练时间影响较大的是循环中每层的 numInput*numOutput，决定总时间的是其中较小的值，

表 3: 不同线程数的程序运行时间 (单位：s)

问题规模	size3							
子线程数	1	2	3	4	5	6	7	8
串行	30.36070599	30.40150117	30.35368941	30.40502581	30.35391538	30.42887226	30.36589808	30.43123709
pthread_sem	39.04147453	19.60434686	13.41327387	10.21026395	8.38806622	7.30424905	6.44540705	6.43263342
pthread+SIMD	22.77276208	12.00376612	9.61660997	7.07640235	6.43254567	6.37675082	6.43523545	6.35252352

故每次不同的规模中各层的维度设置相同。
通过多线程将其中的一层循环进行划分，理论上可以达到接近线程数的加速比。实际实验结果:

- 在问题规模较大时，与理论上基本吻合;
- 在规模较小时,由于线程创建的开销,加速比较小,并行的优势还不足以体现,这也符合预期。除了线程创建的开销,还包括构建线程输入参数的开销,其理论时间复杂度为 O(numSamples*numInput),在总时间中仅体现为常数的增加，但在问题规模较小时仍占有一定比例。

关于 pthread 与 SIMD 的讨论:

- 单纯使用 SIMD 理论上达到 4 倍加速比，但实际为 2.5 倍左右，具体在上一次实验中有所讨论。
- 使用 4 线程 pthread 理论上达到 4 倍加速比，实际也达到了将近 4 倍，这个结果还是比较理想的。
- 使用 pthread+SIMD，理论上来说，应该在 pthread 的基础上提高接近四倍的加速，考虑上次的因素，也应有 2 倍以上的加速比，但实际与无 SIMD 优化的结果十分相近，在问题规模较大是才达到将近 2 倍的额外加速。综合分析原因可能包括:
  1. 向量化的打包解包开销相对较大
  2. 已经将矩阵按列划分后，规模已经较小，以 size3 为例，若进行 8 线程的任务分配后，矩阵列仅为 512/8=64，再将其 4 向量化，仅经过 16 次重复，这已经很难体现出 SIMD 的优势

3.2 不同线程数对比

测试了不同线程数时, 朴素算法、pthreadsem、pthread+SIMD 的程序运行时间与加速比, 如表??所示，其中的问题规模定义如上节所述。其加速比如图3.3所示。

表 4: 不同平台下的程序运行时间与加速比（单位：s）

问题规模	size3	
平台	x86	ARM
串行	30.40502581	4.53168
SIMD	14.5320403	2.36929
SIMD 加速比	2.092275082	1.912674261
pthread_sem	10.21026395	1.99086
pthread_sem 加速比	2.977888325	2.276242428
pthread+SIMD	7.07640235	1.35049
pthread+SIMD 加速比	4.296678497	3.355582048

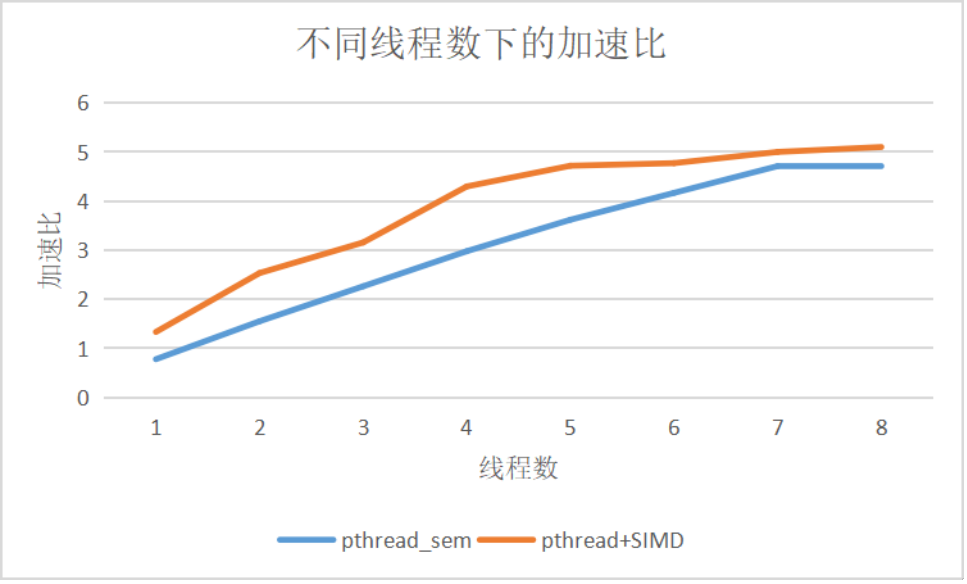

图 3.3: 不同问题规模下的并行程序加速比

可以发现，在不同线程数下，使用信号量实现的 pthread 基本达到了接近线程数的加速比，其结果还是比较理想的。最大分配 8 个线程，算上主线程，最多分配 7 个子线程，故子线程数为 7、8 时结果相近。

3.3 不同平台下对比

分别对比 x86 与 ARM 平台下的程序运行时间，理论上 ARM 应慢于 x86，但加速比应当接近，结果如表5所示，符合预期。

3.4 不同编程策略的并行算法的对比

以下实验均在 size3，4 线程，x86 平台下进行测试。

3.4.1 不同的同步实现方式

分别使用了 barrier 与 sem 实现，分别将全部 train 纳入循环、仅将待计算部分纳入循环并等待主线程唤醒，具体如上章所述。发现程序运行时间 barrier 略快于 sem，如表5所示。可以有以下几种原因:

表 5: 不同同步机制的程序运行时间

	串行	sem	barrier
运行时间 (s)	4.3755	2.10138	1.71972

表 6: 不同数据划分的运行时间（s）

	串行	线程连续读取	线程间隔读取
运行时间	4.3755	2.1013	2.5376

- 主线程同时参与了运算，一些循环的计算线程数增加了一。
- barrier 同步机制的系统开销小于 sem。
- 因为程序的架构不同，把全部 train 函数纳入线程函数减少了部分系统开销，之前过分依赖于主线程的部分被尽可能的分配。

可能的原因有两种:

3.4.2 不同的任务分配方式

对于同一列的任务划分，有以下两种方法，其执行时间具有差异，如表6所示，线程连续读取的运行时间快于线程间隔读取，这是因为局部的 cache 优化，是符合预期的。

线程连续读取

```
int max_n = class_p->num_each_layer[layers_i + 1];
int my_n = (max_n % NUM_THREADS == 0) ? max_n / NUM_THREADS : max_n / NUM_THREADS
    + 1;
for (int i = t_id * my_n; i < std::min (max_n, (t_id + 1) *my_n); i += 1)
```

线程间隔读取

```
for (int i = t_id; i < max_n; i += NUM_THREADS)
```

3.4.3 占用时间的关键循环

分别测试不同循环所占用的时间，如表7所示。从中可以得出以下结论:

- 前向传播的循环占用了最多的时间,但三个循环基本相当。实际上,这仅限于常数的增大,在 batchsize 取 1 的情况下,理论上这三个循环的时间复杂度相同,均为 numInput*numOutput*numLayers,结果符合预期，也因此，优化效果较好。

表 7: 占用时间的关键循环分析（单位：s）

线程编号	0	1	2	3	主线程
前向传播循环	0.589707	0.596950	0.608758	0.605900	0.704961
导数循环	0.473973	0.456292	0.455359	0.475792	等待子线程求导与 bp 时间：0.824249
反向传播循环	0.493160	0.503356	0.492519	0.474948	线程创建、数据创建时间：0.004694
总时间	1.55684	1.556598	1.556636	1.55664	1.57345

- 与预期相同，其他的计算部分对总时间影响较小，因其时间复杂度少了一层 numInput，这是合理的。
- 线程创建与其他初始化确实占用了一定时间，但远小于预期，基本可忽略不计。

4 总结

本文首先介绍期末大作业的选题与本次作业的选题。接下来，介绍算法的设计与实现。最后，介绍不同平台、不同数据规模下的实验和结果分析。实现了对于反向传播算法更新、求导的 SIMD 与多线程并行算法，达到了一定加速比。

项目源代码链接:https://github.com/AldebaranL/Parallel-programming-Homework

参考文献