南開大學
Nankai University

计算机学院

并行程序设计 **SIMD** 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 4 月 6 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

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

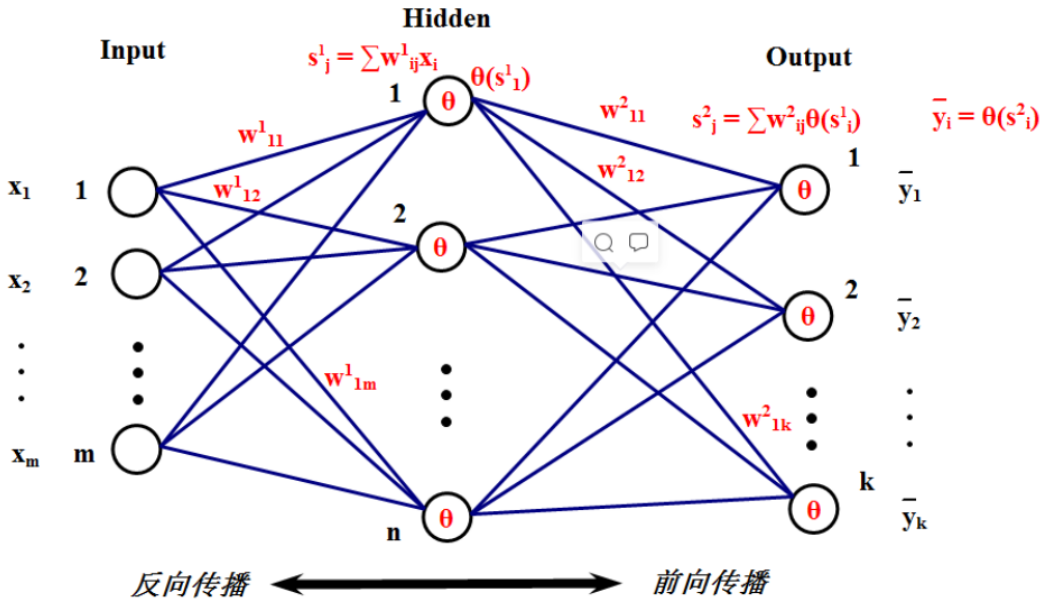

图 1.1: ANN 模型结构

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。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：

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

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。同时，使用 perf 可查看程序对应的指令数并做对比分析。

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。问题中共有三大类循环可以优化。

1. train 函数中的参数更新循环可直对最内层接进行向量化。
2. 前向传播的循环对于 weights 按列访问，首先转置的开销也非常大，故需要首先手动将数据打包为向量，再进行后续向量化操作。
3. 计算导数 (更新量) 的循环可先将大循环拆解成三部分，依次改为按行访问，再进行向量运算。

2.1.5 对齐与不对齐

分别测试对齐与不对齐的存储对程序性能的影响

2.2 朴素算法

如上章所述，设计 ANN-bp 类，包含 train 与 trainVec 方法，这里仅展示本次进行优化的核心代码。

朴素算法

```
void ANN::train(const int _sampleNum, float** _trainMat, float** _labelMat)
{
  float thre = 1e-2;
  for (int i = 0; i < _sampleNum; ++i) {
    train_vec(_trainMat[i], _labelMat[i], i);
  }
  for(int tt = 0; tt < MAXTT; tt++){
    if(!isNotConver_(_sampleNum, _labelMat, thre)) break;
    //调整权值
    for (int index = 0; index < _sampleNum; ++index) {
      for (int i = 0; i < numNodesInputLayer; ++i) {
        for (int j = 0; j < numNodesHiddenLayer; ++j) {
          weights[0][i][j] -= studyRate * allDeltaWeights[index][0][i][j];
        }
      }
      for (int i = 0; i < numNodesHiddenLayer; ++i) {
        for (int j = 0; j < numNodesOutputLayer; ++j) {
          weights[1][i][j] -= studyRate * allDeltaWeights[index][1][i][j];
```

```
19          }
20        }
21      }
22
23      for (int index = 0; index < _sampleNum; ++index) {
24        for (int i = 0; i < numNodesHiddenLayer; ++i) {
25          bias[0][i] -= studyRate * allDeltaBias[index][0][i];
26        }
27        for (int i = 0; i < numNodesOutputLayer; ++i) {
28          bias[1][i] -= studyRate * allDeltaBias[index][1][i];
29        }
30      }
31
32      for (int i = 0; i < _sampleNum; ++i) {
33        train_vec(_trainMat[i], _labelMat[i], i);
34      }
35    }
36
37    //printf("训练权值和偏置成功了! \n");
38  }
39
40  void ANN::train_vec(const float* _trainVec, const float* _labelVec, int index)
41  {
42    //计算各隐藏层结点的输出
43    for (int i = 0; i < numNodesHiddenLayer; ++i) {
44      float z = 0.0;
45      for (int j = 0; j < numNodesInputLayer; ++j) {
46        z += _trainVec[j] * weights[0][j][i];
47      }
48      z += bias[0][i];
49      hidenLayerOutput[i] = sigmoid(z);
50
51    }
52
53    //计算输出层结点的输出值
54    for (int i = 0; i < numNodesOutputLayer; ++i) {
55      float z = 0.0;
56      for (int j = 0; j < numNodesHiddenLayer; ++j) {
57        z += hidenLayerOutput[j] * weights[1][j][i];
58      }
59      z += bias[1][i];
60      outputLayerOutput[i] = sigmoid(z);
61      outputMat[index][i] = outputLayerOutput[i];
62    }
63
64    //计算偏置及权重更新量, 但不更新
65
66    for (int j = 0; j < numNodesOutputLayer; ++j) {
67      allDeltaBias[index][1][j] = (-0.1) * (_labelVec[j] - outputLayerOutput[j]) *
```

```
        outputLayerOutput[j]
68        * (1 - outputLayerOutput[j]);
69      for (int i = 0; i < numNodesHiddenLayer; ++i) {
70        allDeltaWeights[index][1][i][j] = allDeltaBias[index][1][j] *
              hidenLayerOutput[i];
71      }
72    }
73    for (int j = 0; j < numNodesHiddenLayer; ++j) {
74      float z = 0.0;
75      for (int k = 0; k < numNodesOutputLayer; ++k) {
76        z += weights[1][j][k] * allDeltaBias[index][1][k];
77      }
78      allDeltaBias[index][0][j] = z * hidenLayerOutput[j] * (1 -
          hidenLayerOutput[j]);
79      for (int i = 0; i < numNodesInputLayer; ++i) {
80        allDeltaWeights[index][0][i][j] = allDeltaBias[index][0][j] * _trainVec[i];
81      }
82    }
83  }
```

2.3 cache 优化与循环展开算法

根据行主次序存储的特性，改变某些循环的运算次序，在 train 函数中，使用了 4 路循环展开，代码如下。

cache 优化与循环展开算法

```
1  void ANN_SIMD::train_cache(const int _sampleNum, float** _trainMat, float**
       _labelMat)
2  {
3      ...
4    for(int tt = 0; tt < MAXTT; tt++){
5      //if(!isNotConver_(_sampleNum, _labelMat, thre)) break;
6      //调整权值
7      for (int index = 0; index < _sampleNum; ++index) {
8        for (int i = 0; i < numNodesInputLayer; ++i) {
9          for (int j = 0; j < numNodesHiddenLayer; j+=4) {
10                       //四路循环展开
11                       weights[0][i][j] -= studyRate *
                             allDeltaWeights[index][0][i][j];
12                       weights[0][i][j+1] -= studyRate *
                             allDeltaWeights[index][0][i][j+1];
13                       weights[0][i][j+2] -= studyRate *
                             allDeltaWeights[index][0][i][j+2];
14                       weights[0][i][j+3] -= studyRate *
                             allDeltaWeights[index][0][i][j+3];
15         }
16       }
17             ...//基本思路相同，即按行访问，并全部四路展开。
```

```
18      }
19          ...
20    }
21  }
22  void ANN_SIMD::train_vec_cache(const float* _trainVec, const float* _labelVec, int
        index)
23  {
24    //计算各隐藏层结点的输出
25    for(int i=0;i<numNodesHiddenLayer;i++){
26          hidenLayerOutput[i]=0.0;
27    }
28    //对weights,拆分为按行访问weights[0]
29    for (int j = 0; j < numNodesInputLayer; ++j) {
30          for (int i = 0; i < numNodesHiddenLayer; ++i) {
31        hidenLayerOutput[i] += _trainVec[j] * weights[0][j][i];
32      }
33    }
34    for(int i=0;i<numNodesHiddenLayer;i++){
35          hidenLayerOutput[i]+=bias[0][i];
36          hidenLayerOutput[i]=sigmoid(hidenLayerOutput[i]);
37    }
38
39    //计算输出层结点的输出值
40      ...
41    //计算偏置及权重更新量, 但不更新, 改为按行访问
42    for (int j = 0; j < numNodesOutputLayer; ++j) {
43      allDeltaBias[index][1][j] = (-0.1) * (_labelVec[j] - outputLayerOutput[j]) *
            outputLayerOutput[j]
44        * (1 - outputLayerOutput[j]);
45    }
46      for (int i = 0; i < numNodesHiddenLayer; ++i) {
47          for (int j = 0; j < numNodesOutputLayer; ++j) {
48        allDeltaWeights[index][1][i][j] = allDeltaBias[index][1][j] *
              hidenLayerOutput[i];
49      }
50    }
51    for (int j = 0; j < numNodesHiddenLayer; ++j) {
52      float z = 0.0;
53      //按行访问(k)
54      for (int k = 0; k < numNodesOutputLayer; ++k) {
55        z += weights[1][j][k] * allDeltaBias[index][1][k];
56      }
57      allDeltaBias[index][0][j] = z * hidenLayerOutput[j] * (1 -
            hidenLayerOutput[j]);
58    }
59      for (int i = 0; i < numNodesInputLayer; ++i) {
60          for (int j = 0; j < numNodesHiddenLayer; ++j) {
61        allDeltaWeights[index][0][i][j] = allDeltaBias[index][0][j] * _trainVec[i];
62      }
```

```
63    }
64 }
```

2.4 SSE 算法

与 neon 类似，使用向量化每 4 个 float 一次操作，具体细节如注释所述，以下是不对齐的 sse 实现，具体代码过长，这里仅展示几个代表性的循环。对齐的 SSE 算法与之相似，在动态分配时指定对齐并将 ld 与 st 指令改为对齐。

sse 算法

```
1  void ANN_SIMD::train_sse(const int _sampleNum, float** _trainMat, float**
       _labelMat)
2  {
3      ...
4    for(int tt = 0; tt < MAXTT; tt++){
5      //if(!isNotConver_(_sampleNum, _labelMat, thre)) break;
6          //调整权值
7      __m128 sr = _mm_set1_ps(studyRate);
8          __m128 t1, t2;
9      for (int index = 0; index < _sampleNum; ++index) {
10       for (int i = 0; i < numNodesInputLayer; ++i) {
11         for (int j = 0; j < numNodesHiddenLayer-4; j+=4) {
12                     //将内部循环改为4位的向量运算
13             t1 = _mm_loadu_ps(allDeltaWeights[index][0][i] + j);
14             t1 = _mm_mul_ps(t1, sr);
15             t2 = _mm_loadu_ps(weights[0][i] + j);
16             t2 = _mm_sub_ps(t2,t1);
17                     //向量存
18                     _mm_storeu_ps(weights[0][i] + j, t2);
19           //weights[0][i][j] -= studyRate * allDeltaWeights[index][0][i][j];
20         }
21       }
22         ...//并行化方式类似的循环
23     }
24     ...
25   }
26   ...
27 }
28 void ANN_SIMD::train_vec_sse(const float* _trainVec, const float* _labelVec, int
       index)
29 {
30   //计算各隐藏层结点的输出
31     for (int i = 0; i < numNodesHiddenLayer; ++i) {
32         float z = 0.0;
33         __m128 sums = _mm_setzero_ps();
34         // 内层循环使用SSE(AVX)每次处理4(8)个float数据
35         for (int j = 0; j < numNodesInputLayer - 4; j += 4) {
36             __m128 vecs = _mm_loadu_ps(_trainVec + j);
```

```
37              // weights不连续，需各个单独设置，注意高低位
38              __m128 weights128 = _mm_set_ps(weights[0][j + 3][i], weights[0][j +
                    2][i], weights[0][j + 1][i], weights[0][j][i]);
39              // 融合乘加指令，也可以分开写
40              sums = _mm_fmadd_ps(vecs, weights128, sums);
41          }
42          // 4个局部和相加
43          sums = _mm_hadd_ps(sums, sums);
44          sums = _mm_hadd_ps(sums, sums);
45          //标量存储
46          _mm_store_ss(&z, sums);
47          hidenLayerOutput[i] = sigmoid(z);
48      }
49
50  //计算输出层结点的输出值,结构同上
51      ...
52
53  //计算偏置更新量allDeltaBias与allDeltaWeights[1]
54      ...
55      //计算偏置更新量allDeltaBias与allDeltaWeights[0]
56      //z数组用于存储中间结果
57      float *z=new float[numNodesHiddenLayer];
58  for (int j = numNodesHiddenLayer-1; j >= 0; j --) {
59          __m128 t1, t2, sum;
60          //初始置0
61          sum = _mm_setzero_ps();
62      for (int i = numNodesOutputLayer - 4; i >= 0; i -= 4){
63              t1 = _mm_loadu_ps(weights[1][j] + i);
64              t2 = _mm_loadu_ps(allDeltaBias[index][1] + i);
65              //向量乘
66              t1 = _mm_mul_ps(t1, t2);
67              //向量加
68              sum = _mm_add_ps(sum, t1);
69      }
70      //横向加，最终sum中每32位均为结果。
71      sum = _mm_hadd_ps(sum, sum);
72          sum = _mm_hadd_ps(sum, sum);
73          //存储32位的sum到z[j]
74          _mm_store_ss(z+j,sum);
75  }
76      for (int j = numNodesHiddenLayer-4; j >= 0; j -=4) {
77          //allDeltaBias[index][0][j] = z * hidenLayerOutput[j] * (1 -
                hidenLayerOutput[j]);
78          __m128 t1, t2,t3, product;
79          t1 = _mm_loadu_ps(z+j);
80          t2 = _mm_loadu_ps(hidenLayerOutput+j);
81          t3 = _mm_set1_ps(1);
82          t3 = _mm_sub_ps(t3,t2);
83          //product=t1*t2*t3，向量对位相乘
```

```
84          product = _mm_mul_ps(t1,t2);
85          product = _mm_mul_ps(product,t3);
86          _mm_storeu_ps(allDeltaBias[index][0]+j,product);
87      }
88      delete[]z;
89      for (int i = numNodesInputLayer-1; i>= 0; i --) {
90          for (int j = numNodesHiddenLayer -4;j>=0;j-= 4) {
91              __m128 a1, a2,pro;
92              //标量load
93              a1 = _mm_set1_ps (_trainVec[i]);
94              //向量乘
95              a2 = _mm_loadu_ps(allDeltaBias[index][0]+j);
96              pro =  _mm_mul_ps(a2, a1);
97              //存储
98              _mm_storeu_ps(allDeltaWeights[index][0][i] + j, pro);
99          }
100     }
101 }
```

2.5 AVX 算法

与 sse 类似，每 8 个数同时操作，代码较长，这里仅展示一个循环作为示意。

avx 算法

```
1  void ANN_SIMD::train_avx(const int _sampleNum, float** _trainMat, float**
       _labelMat)
2  {
3      ...
4    for(int tt = 0; tt < MAXTT; tt++){
5      //调整权值
6      __m256 sr = _mm256_set1_ps(studyRate);
7          __m256 t1, t2;
8      for (int index = 0; index < _sampleNum; ++index) {
9        for (int i = 0; i < numNodesInputLayer; ++i) {
10         for (int j = 0; j < numNodesHiddenLayer - 8; j += 8) {
11                     //将内部循环改为8位的向量运算
12             t1 = _mm256_loadu_ps(allDeltaWeights[index][0][i] + j);
13             t1 = _mm256_mul_ps(t1, sr);
14             t2 = _mm256_loadu_ps(weights[0][i] + j);
15             t2 = _mm256_sub_ps(t2,t1);
16                     //向量存
17                     _mm256_storeu_ps(weights[0][i] + j, t2);
18           //weights[0][i][j] -= studyRate * allDeltaWeights[index][0][i][j];
19         }
20       }
21       ...
22     }
23     ...
```

```
  }
  ...
}
```

2.6 Noen 算法

使用向量化，每 4 个 float 一次操作，具体实现方法与 sse 类似，这里仅展示一个循环作为示意。

Noen 算法

```
void ANN_SIMD::train_neon(const int _sampleNum, float** _trainMat, float**
    _labelMat)
{
  ...
  for(int tt = 0; tt < MAXTT; tt++){
    if(!isNotConver_(_sampleNum, _labelMat, thre)) break;
    //调整权值
    float32x4_t sr = vmovq_n_f32(studyRate);
        float32x4_t t1, t2;
    for (int index = 0; index < _sampleNum; ++index) {
      for (int i = 0; i < numNodesInputLayer; ++i) {
        for (int j = 0; j < numNodesHiddenLayer-4; j+=4) {
          t1 = vld1q_f32(allDeltaWeights[index][0][i] + j);
            t1 = vmulq_f32(t1, sr);
            t2 = vld1q_f32(weights[0][i] + j);
            t2 = vsubq_f32(t2,t1);
                    vst1q_f32(weights[0][i] + j, t2);
          //weights[0][i][j] -= studyRate * allDeltaWeights[index][0][i][j];
        }
      }
            ...
    }
    ...
    }
}

void ANN_SIMD::train_vec_neon(const float* _trainVec, const float* _labelVec, int
    index)
{
    ...
}
```

表 1: 对齐是否对程序性能的影响

	对齐	不对齐
程序运行时间 (s)	3.92740	4.09173
问题规模	size4	
算法	SSE	

表 2: 问题规模

	size1	size2	size3	size4
numInput	256	128	128	128
numOutput	256	128	128	128
numHidden1	1024	1024	256	256
numLayers	1	1	1	1
numClass	8	8	8	8
numSamplePerClass	32	32	32	64
t	16	16	16	16

3 实验和结果分析

3.1 对齐与不对齐算法的程序运行时间，如表所示。的对比

分别测试了对齐与不对齐算法的程序运行时间，如表1所示, 经过实验发现，其运行时间相近。但理论上手动对齐应减少了一部分对齐的开销，应该更优，出现差异的原因可能是程序实际分配内存时本来就有一部分是对齐的所以差别不大。

3.2 不同问题规模下串行算法与 neon 并行算法的对比

不同问题规模详细的定义如表2所示。其中，numInput、numOutput、numHidden1、numLayers、numClass、numSamplePerClass、t 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。在华为鲲鹏平台上的实际运行时间如表3所示。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。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。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。接下来，又测试了不同训练样本数，即训练样本类别数 * 每个类别训练样本数，的程序运行时间。理论上来说，程序的时间复杂度为:

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

从中可以得出以下几个结论:

表 3: 华为鲲鹏平台不同问题规模下的程序运行时间对比

运行时间 (s)	size1	size2	size3	size4
串行算法	73.0008474	35.49578206	8.47293839	16.95243801
Neon 并行算法	39.706902	19.53183248	5.24248915	10.48018661
加速比	1.838492648	1.817329843	1.616205231	1.617570244

表 4: 不同并行策略对程序性能的影响

	串行	仅对更新进行向量化	仅对正向传播与求导计算进行向量化	全部 4 路向量化
程序运行时间 (s)	4.34234	4.09173	3.40656	2.85461
问题规模	size4			

表 5: 不同平台、不同问题规模下的程序运行时间对比

运行时间 (s)		size1	size2	size3	size4
ARM	串行算法	73.0008474	35.49578206	8.47293839	16.95243801
	Neon 并行算法	39.706902	19.53183248	5.24248915	10.48018661
	加速比	1.838492648	1.817329843	1.616205231	1.617570244
x86	串行算法	24.151	10.7332	2.30172	4.76322
	cache 与循环展开	14.7024	7.5379	1.93488	3.9638
	加速比	1.642656981	1.423897903	1.189593153	1.201680206
	SSE 并行算法	11.1215	5.59522	1.50242	3.18288
	加速比	2.171559592	1.918280246	1.53200836	1.496512592
	AVX 并行算法	9.2386	4.61232	1.12339	2.44927
	加速比	2.614140671	2.327071842	2.048905545	1.944750885

- 对于相同的算法，时间与问题规模的关系与理论上相同，基本符合预期。
- 对于并行与串行算法的比较，由于是 4 个浮点数的向量化，理论上来说至多提高 4 倍的性能，而实际的加速比维持在 1.6-1.85 之间，这可能是由于在函数中存在未进行并行化的条件分支语句、打包/解包的数据访存等额外开销造成的，基本上也是符合预期的。
- 对于加速比的影响因素，可以发现，加速比与样本数量规模的关系较小，而随着隐藏层规模的增大，加速比增大，这是由于代码中多个向量化的循环均与隐藏层的循环相关，这也是符合预期的。

3.3 不同编程策略的并行算法的对比

测试了更新部分的循环向量化与求导计算循环向量化对程序性能的影响。具体如表4所示。从中可以得出以下结论:

- 对更新过程的向量化与对权重正向传播计算的向量化相比，后者对性能的影响更加显著。

3.4 不同平台下，朴素算法、cache 与循环展开、SSE、AVX、neon 并行算法的对比

测试了不同平台下，朴素算法、cache 与循环展开、SSE、AVX、neon 并行算法以及不同规模的程序运行时间，如表5所示，其中的问题规模与上一小节定义相同。从中可以得出以下几个结论:

- 对于相同的算法，时间与问题规模的关系与理论上相同，基本符合预期。
- 相同问题规模下，arm 平台慢于 x86 平台。
- cache 优化加四路循环展开算法的加速比约在 1.2-1.6 之间，随着输入、输出、隐藏层规模的增大而增加。
- sse 并行算法的加速比约在 1.5-2.2 之间，随着输入、输出、隐藏层规模的增大而增加。除正向传播的循环之外的每个循环，都对最内层循环进行了对位的向量化运算的优化。正向传播的循环由于需要同时访问不同列的元素，首先将其手动访问与打包，需要更大的打包解包开销。理论上来

说，与 neon128 相同，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，结果是符合预期的。

- avx 并行算法理论上可达上限为 8 倍的加速比，但实际测试结果仅在 1.9-2.7 倍左右，并不理想。但经过其他规模的反复测试，发现在有些情况下加速比可达接近 6，因为算法整体结构循环较多，不同变量也比较复杂，故出现差异。

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 并行算法，达到一定加速比，接下来的作业中预计会做进一步的并行化

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

参考文献