人工神经网络的训练与推理研究

周宝航 2120190442

一、问题简介

作为应用最为广泛的机器学习算法，神经网络模型具有较好的数据拟合能力。人工神经网络（ANN）模拟人类神经元的工作方式，通过神经元节点之间的信息传递与激活，达到输出任务目标的目的。人工神经网络由包含不同数目神经元节点的隐含层组成，且不同层之间的神经元一一相连接。当数据被输入网络时，模型通过每个隐含层向前传递信息并输出目标值。我们称这个过程为：前向传播算法（推理）。但初始网络并不能很好地预测任务目标值，我们需要根据模型的预测值与真实值进行误差计算并反向传回各个隐含层的梯度来更新参数。我们称这个过程为：反向传播算法（训练）。

对于人工神经网络的前向传播与反向传播过程，我们一般将它们抽象为矩阵运算。因此，神经网络模型软件实现的瓶颈在于矩阵运算的速度。我们在本次实验中主要针对矩阵乘法进行并行化设计，达到加速人工神经网络推理与训练的目的。我们使用 MNIST（手写数字识别）数据集进行模型训练来完成分类任务。在实验设计中，我们考虑并行化算法对模型性能的影响以及不同问题规模上的表现。这点体现在：神经网络具有不同的规模或者数据批次的大小时，并行化算法性能的表现。

二、算法描述与实现

1. 人工神经网络的推理与训练

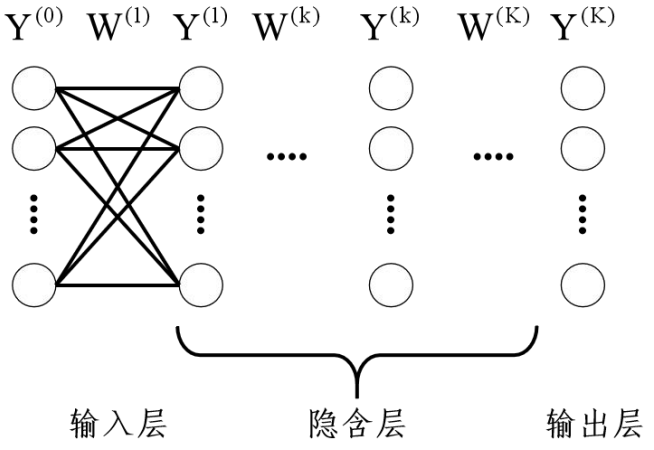

图 1 人工神经网络模型

上图为人工神经网络的简化模型，其中 $Y^{(0)}$ 表示输入数据，$W^{(k)}$ 表示网络层

神经元之间的连接权重，$\mathrm{Y}^{(k)}$ 表示神经元输出数据经过激活函数后的数值。一般每个网络层还会有一个偏置项 $\mathrm{b}^{(k)}$ 来更好地拟合数据。我们在网络层中常使用的激活函数为：$\sigma(z)=\dfrac{1}{1+\exp(-z)}$。前向传播的算法流程如下：

$$\text{For}\quad k \leftarrow 1,2,...,K$$

$$\begin{aligned}\mathrm{net}^{(k)} &\leftarrow \mathrm{W}^{(\mathrm{k})}\mathrm{Y}^{(k-1)}+\mathrm{b}^{(k)}\\ \mathrm{Y}^{(k)} &\leftarrow \sigma^{(k)}(\mathrm{net}^{(k)})\end{aligned}\quad 。$$

当经过前向传播得到预测值 $\mathrm{Y}^{(K)}$ 后，我们需要评估预测值与真实值 T 之间的误差。在分类问题中，交叉熵损失函数是最常使用的一种误差评价方式，其定义为：$\mathrm{L}=-\sum_{\mathrm{i}=1}^{m}\mathrm{T}_{(\mathrm{i})}*\log(\mathrm{Y}_{(\mathrm{i})}^{(K)})+(1-\mathrm{T}_{(\mathrm{i})})*\log(1-\mathrm{Y}_{(\mathrm{i})}^{(K)})$。通过计算误差函数值后，我们可以进行反向传播计算各层的梯度来更新参数值。反向传播的算法流程如下：

$$\text{For}\quad k \leftarrow K,K-1,...,1$$

$$\begin{aligned}\mathrm{S}^{(k)} &\leftarrow \begin{cases}\sigma'^{(K)}(\mathrm{net}^{(K)})\bullet(\mathrm{Y}^{(k)}-\mathrm{T}), & k=K\\ \sigma'^{(K)}(\mathrm{net}^{(K)})\bullet(\mathrm{W}^{(k+1)T}\mathrm{S}^{(k+1)}), & k\neq K\end{cases}\\ \frac{\partial \mathrm{L}}{\partial \mathrm{W}^{(k)}} &\leftarrow \mathrm{S}^{(k)}\mathrm{Y}^{(k-1)T}\\ \frac{\partial \mathrm{L}}{\partial \mathrm{b}^{(k)}} &\leftarrow \mathrm{S}^{(k)}\end{aligned}\quad ,$$

至此我们计算出了各网络层的梯度值。为了更新网络权重，我们一般采用随机梯度下降的方法，并通过引入超参数——学习率 α 来控制网络更新的程度。具体流程如下：

$$\text{For}\quad k \leftarrow K,K-1,...,1$$

$$\begin{aligned}\mathrm{W}^{(k)} &\leftarrow \mathrm{W}^{(k)}-\alpha\frac{\partial \mathrm{L}}{\partial \mathrm{W}^{(k)}}\\ \mathrm{b}^{(k)} &\leftarrow \mathrm{b}^{(k)}-\alpha\frac{\partial \mathrm{L}}{\partial \mathrm{b}^{(k)}}\end{aligned}\quad 。$$

上面介绍的是神经网络一次迭代训练的算法流程。实际中，我们需要多次迭代训练网络才能使损失值降到相对低的大小。同时，我们可以发现：矩阵乘法运算占到了算法流程中的很大一部分。因此，我们对该部分运算进行并行化设计对提升性能会非常有帮助。

下面是我们使用 C++定义的神经网络类：

```
class NeuralNetwork
{
public:
    NeuralNetwork();
    ~NeuralNetwork();
    NeuralNetwork(vector<int> hiddens, double lr, QString mode);
    // 训练方法
    void train(matrix x, matrix y);
    // 预测方法
    matrix predict(matrix x);
    // 损失函数
    double loss(matrix y, matrix _y);
private:
    // 学习率
    double p_lr;
    // 网络层数
    int n_layers;
    // 网络层参数
    vector<matrix> layers;
    // 运算加速
    QString p_mode;
    // 前向传播算法
    vector<vector<matrix>> forward(matrix input);
    // 反向传播算法
    void backward(matrix output, vector<vector<matrix>> cache);
    // 初始化网络层参数
    matrix initializer_of_layer(int m, int n);
    // 激活函数
    matrix sigmoid(matrix z, bool flag);
};
```

该类中定义了网络的基本参数，包括：学习率、网络每层的权重。而网络的前向传播算法与反向传播算法具体实现可见“附录—前向与反向传播算法的实现”。通过 train 方法，我们传入训练数据样本和标签，对网络进行一次迭代的训练。为了直观展示模型是否收敛，我们定义了 loss 函数来计算预测值与真实值之间的误差。在实验中，我们为了对比并行化算法与普通算法的性能，通过 p_mode 属性决定网络类是否使用并行化算法加速运行。

2. 矩阵乘法并行算法

考虑到矩阵运算是主要处理方式，我们对矩阵的各种操作进行抽象封装。其主要定义如下：

```
class matrix
{
```

```
3.  public:
4.      matrix(int row, int col, const double init);
5.      matrix(const matrix& B);
6.      // 索引取值
7.      double& operator() (int i, int j);
8.      const double operator() (int i, int j) const;
9.      // 矩阵赋值
10.     matrix& operator =(matrix&& B);
11.     // 矩阵元素判等
12.     matrix operator ==(matrix B);
13.     // 减法运算
14.     matrix operator -(double a);
15.     matrix operator -(matrix a);
16.     // 加法运算
17.     matrix operator +(matrix a);
18.     // 元素乘法运算
19.     matrix operator *(double a);
20.     matrix operator *(matrix a);
21.     // 索引运算
22.     matrix index(vector<int> idx);
23.     // 最大值索引
24.     matrix argmax(int axis);
25.     // 矩阵乘法
26.     matrix mult(const matrix &B);
27.     // SIMD 加速
28.     matrix sse_mult(const matrix &B);
29.     // 矩阵转置
30.     matrix transpose();
31. private:
32.     double **p_matrix;
33.     int p_row, p_col;
34.     void sse_mult_kernel(double **c, double **a, double **b, int row, int co
    l, int b_col);
35. };
```

上面的矩阵类包含了矩阵的基本操作，如：矩阵乘法、点乘、转置、元素相加等等。在为矩阵申请内存时，我们采用_mm_malloc 函数申请对齐地址内存。为了加速矩阵乘法运算，我们还使用 SSE 重新设计了矩阵乘法的流程。

矩阵乘法是一个叠加过程，所以我们将待相乘的 A、B 两矩阵按照 A 取四行、B 取四列的划分，将这两块矩阵的运算采用 SSE 指令集函数进行加速。然后，A、B 两个矩阵的全部划分均复用这个加速函数，即可完成 A、B 矩阵相乘运算。SSE 加速矩阵核运算的伪代码如下：

输入：输出矩阵 C、输入矩阵 A、输入矩阵 B、矩阵核起始行 row、起始列 col

For **n** in [0, 1]:

a_**n** = _mm_load_pd(A[0]) // 从 A 矩阵的一列取两个元素

For **k** in [0,1,2,3]:

b_**k** = _mm_set1_pd(B[**k**]) // 从 B 矩阵的一列取两个元素

t_**n**_**k** += a_**n** * b_**k** //A、B 矩阵对应位置相乘

For **k** in [0,1,2,3]:

_mm_store_pd(c[col+**k**][row], t_0_**k**) // 对应输出至矩阵 C 的相应位置

_mm_store_pd(c[col+**k**][row+2], t_1_**k**)// 对应输出至矩阵 C 的相应位置

输出：矩阵 C

上述算法实现可见“附录—SSE 矩阵核算法的实现”。而实现整个矩阵乘法的代码如下：

```
matrix matrix::sse_mult(const matrix &B)
{
    if (p_col != B.p_row) return *this;
    matrix tmp(p_row, B.p_col, 0);
    double *ta[2];
    ta[0] = (double*)_mm_malloc(sizeof(double)*2*p_col, 16);
    ta[1] = (double*)_mm_malloc(sizeof(double)*2*p_col, 16);
    int i = 0, j = 0, k;
    do {
        k = 0; i = 0;
        do {
            ta[0][k]   = p_matrix[i][j];
            ta[1][k++] = p_matrix[i][j+2];
            ta[0][k]   = p_matrix[i][j+1];
            ta[1][k++] = p_matrix[i++][j+3];
        } while (i < p_col);
        i = 0;
        do {
             this->sse_mult_kernel(tmp.p_matrix, ta, B.p_matrix, j, i, B.p_co
l);
            i += 4;
        } while (i < B.p_col);
        j += 4;
    } while (j < p_row);
    _mm_free(ta[0]);
    _mm_free(ta[1]);
    return tmp;
}
```

上面的函数通过调用矩阵核运算 sse_mult_kernel 函数来完成对整个矩阵的乘法运算。

三、实验结果与分析

实验中，我们针对不同问题规模下算法性能的表现进行了分析。对于神经网络模型，我们主要从模型和数据的角度来讨论问题规模，主要包括：网络的隐含层节点数量、网络的隐含层数量以及每次迭代输入网络的数据量。因为随机梯度下降法收敛时间较长、抖动的比较严重，我们在实验中采用批量梯度下降法。该方法选取一小批数据输入模型，然后采用前向与反向传播算法进行训练。由于神经网络的训练过程比较长，我们只固定迭代 60 次，然后比较不同问题规模下串、并行算法的性能。

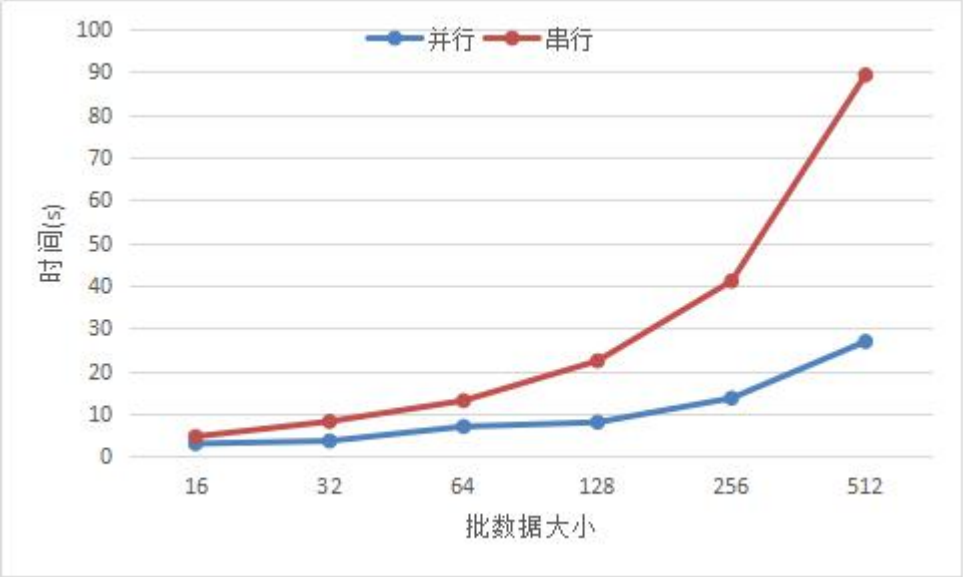

图 2 不同批数据大小对训练时间的影响

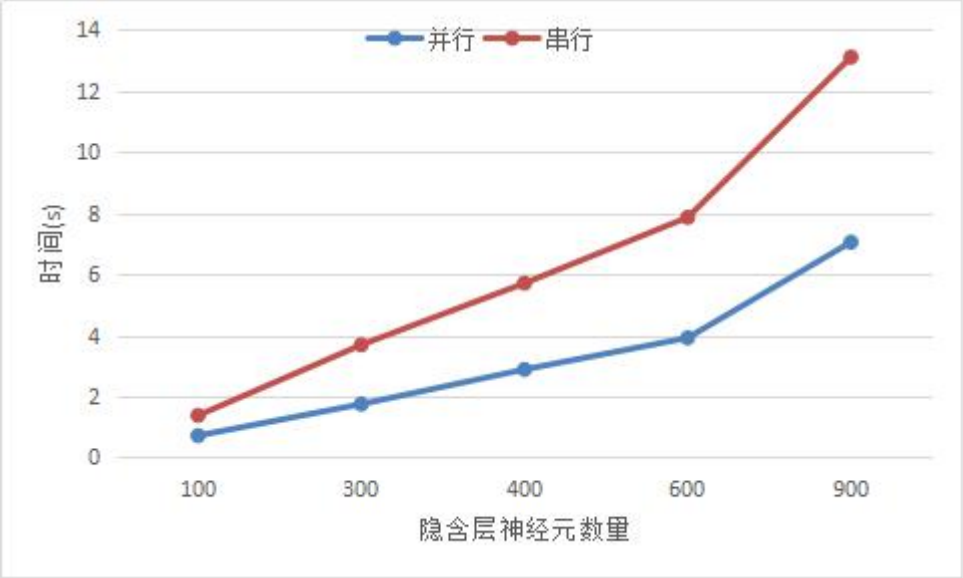

图 3 隐含层神经元数量对训练时间的影响

图 2 所示实验结果为：不同批数据大小下算法性能的表现对比。我们在实验

中设置的网络结构为：输入层：784 个节点；隐含层：900 个节点；输出层：10 个节点。在该网络结构下，我们考虑不同批数据大小下算法性能的表现。由图 2 可以看出：随着批数据量的增多，并行算法支持下的神经网络需要相对更少的时间来进行训练。这点也体现了 SIMD 编程的优势，随着计算量的增加，并行化算法的计算时间增加的比较缓慢。在实际应用中，采用并行化计算的方式对于神经网络的训练与推理有着很好的加速效果。

图 3 所示实验结果为：不同隐含层神经元数量下并行化算法的性能表现。我们在实验中设置的网络结构为：输入层：784 个节点；隐含层：n 个节点；输出层：10 个节点。在不同的网络结构下，我们均采用批数据量为：64 的小样本集来训练网络。我们可以看到：随着隐含层神经元数量的增加，并行算法的消耗时间较串行算法更少。但是，相对于批数据大小对模型计算时间的影响，隐含层神经元的数量对模型计算时间的影响更大。这点主要体现为：网络节点的增多意味着矩阵乘法的耗时更长。虽然并行化算法一定程度上加速了计算效率，但是网络规模的不断增大，并行算法的性能会受到挑战。总体来说，并行算法对加速人工神经网络有着明显的效果。

对于计算精度的考虑，我们在实验中对比了串行与并行算法的损失值，发现两者均可以很好的收敛，且损失值之间几乎没有差距。这也表明并行化算法在没有损失计算精度的前提下加速了计算过程。这对实际工程中部署高性能的人工神经网络应用有着重要的指导意义。

附录

前向与反向传播算法的实现

```
vector<vector<matrix>> NeuralNetwork::forward(matrix input)
{
    matrix tmp(1,1,0);
    vector<matrix> zs;
    vector<matrix> activations = {input.transpose()};
    vector<vector<matrix>> cache;
    for (int i = 0; i < layers.size(); i++) {
        // 循环每层权重，前向计算每层输出值
        if (this->p_mode == "normal")
            tmp = layers[i].mult(activations[activations.size() - 1]);
        else if (this->p_mode == "sse")
            tmp = layers[i].sse_mult(activations[activations.size() - 1]);
        zs.push_back(matrix(tmp));
        // 保存经过激活函数数值
        activations.push_back(sigmoid(tmp, false));
    }
    cache.push_back(zs);
```

```
    cache.push_back(activations);
    return cache;
}

void NeuralNetwork::backward(matrix output, vector<vector<matrix>> cache)
{
    int n_layers = this->n_layers;
    vector<matrix> zs = cache[0];
    vector<matrix> activations = cache[1];
    // 每层权重对应的梯度
    vector<matrix> nable_w;
    for (auto l : layers)
        nable_w.push_back(matrix(l.getRowSize(), l.getColSize(), 0));
    // 计算损失
    matrix cost = activations[n_layers - 1] - output;
    // 计算梯度
    matrix gradient = cost * sigmoid(zs[n_layers - 2], true);
    // 计算最后一层权重的梯度
    if (this->p_mode == "normal")
        nable_w[nable_w.size() - 1] = gradient.mult(activations[n_layers - 2].transpose());
    else if (this->p_mode == "sse")
         nable_w[nable_w.size() - 1] = gradient.sse_mult(activations[n_layers - 2].transpose());
    // 反向传播
    for (int i = 2; i <= layers.size(); i++) {
        // 反向传播梯度，并计算每层权重的梯度值
        if (this->p_mode == "normal") {
            gradient = layers[n_layers-i].transpose().mult(gradient)
                        * sigmoid(zs[n_layers-i-1], true);
             nable_w[n_layers-i-1] = gradient.mult(activations[n_layers-i-1].transpose());
        }
        else if (this->p_mode == "sse") {
            gradient = layers[n_layers-i].transpose().sse_mult(gradient)
                        * sigmoid(zs[n_layers-i-1], true);
             nable_w[n_layers-i-1] = gradient.sse_mult(activations[n_layers-i-1].transpose());
        }
    }
    // 利用梯度，更新每层权重
    for (int i = 0; i < layers.size(); i++) {
        layers[i] = layers[i] - nable_w[i] * p_lr;
    }
```

```
58. }
```

SSE 矩阵核算法的实现

```
1.  void matrix::sse_mult_kernel(double **c, double **a, double **b, int row, in
    t col, int b_col)
2.  {
3.      __m128d t01_0, t01_1, t01_2, t01_3;
4.      __m128d t23_0, t23_1, t23_2, t23_3;
5.      __m128d a0, a1, b0, b1, b2, b3;
6.      t01_0 = t01_1 = t01_2 = t01_3 = _mm_set1_pd(0);
7.      t23_0 = t23_1 = t23_2 = t23_3 = _mm_set1_pd(0);
8.      double *pb0(b[col]), *pb1(b[col+1]), *pb2(b[col+2]), *pb3(b[col+3]);
9.      double *pa0(a[0]), *pa1(a[1]);
10.     double *endb0 = pb0 + p_col;
11.     do {
12.         a0 = _mm_load_pd(pa0);
13.         a1 = _mm_load_pd(pa1);
14.
15.         b0 = _mm_set1_pd(*(pb0++));
16.         if (col+1 < b_col)
17.             b1 = _mm_set1_pd(*(pb1++));
18.         if (col+2 < b_col)
19.             b2 = _mm_set1_pd(*(pb2++));
20.         if (col+3 < b_col)
21.             b3 = _mm_set1_pd(*(pb3++));
22.         t01_0 += a0 * b0;
23.         if (col+1 < b_col)
24.             t01_1 += a0 * b1;
25.         if (col+2 < b_col)
26.             t01_2 += a0 * b2;
27.         if (col+3 < b_col)
28.             t01_3 += a0 * b3;
29.         t23_0 += a1 * b0;
30.         if (col+1 < b_col)
31.             t23_1 += a1 * b1;
32.         if (col+2 < b_col)
33.             t23_2 += a1 * b2;
34.         if (col+3 < b_col)
35.             t23_3 += a1 * b3;
36.         pa0 += 2;
37.         pa1 += 2;
38.     } while (pb0 != endb0);
39.     _mm_store_pd(&c[col][row], t01_0);
40.     _mm_store_pd(&c[col][row+2], t23_0);
```

```
    if (col+1 < b_col) {
        _mm_store_pd(&c[col+1][row], t01_1);
        _mm_store_pd(&c[col+1][row+2], t23_1);
    }
    if (col+2 < b_col) {
        _mm_store_pd(&c[col+2][row], t01_2);
        _mm_store_pd(&c[col+2][row+2], t23_2);
    }
    if (col+3 < b_col) {
        _mm_store_pd(&c[col+3][row], t01_3);
        _mm_store_pd(&c[col+3][row+2], t23_3);
    }
}
```