知行一

技术探讨、社区精华

一个C++14模板元实现的深度学习神经网络模板类，支持任意层数

2017年8月3日 | 杨清风 | 1条评论

构造编译期矩阵以及数据传递代码，headonly
搜遍了github，在模板元这块机器学习还是空白，正好是个填补，我接下来会逐渐丰富这个库（倒是有几个模板元数学运算库，都很简陋）
大量的矩阵运算用模板元进行有几个让人非常惬意的优势，也发觉模板元其实很适合这种编程
（不知道是否唯有C++才有的优势，数学专用语言不算在内，比如m、r这些）：
1、永远不用担心数组越界，也不用写检查数组越界的代码
2、矩阵运算不用检查行列是否匹配，行列的要求通过模板函数参数就能限定了
3、快，只有cpper才懂的快
代码在这里 https://github.com/bowdar/DeepLearning

先看使用方法，过程极其简单

```
#include "NeuralNetwork.hpp"
int main()
{
    /// 1. 创建一个输入层，两个隐含层，一个输出层的神经网络
    mtl::BPNeuralNet<20, 30, 20, 2> mynn;

    /// 2. 初始化
    mynn.init(0.001, 0.8);

    /// 3. 输入
    mtl::Matrix<double, 1, 20> inMatrix;
    mtl::Matrix<double, 1, 2> outMatrix;
    ///    录入你的矩阵数据...

    /// 4. 训练
    mynn.train(inMatrix, outMatrix, 100);

    /// 5. 仿真
```

```
    mynn.simulate(inMatrix, outMatrix);
}
```

模板类的申明，开头是用来迭代整形模板参数的UnpackInts，根据Index取值，没有使用TypeList
代码使用到的矩阵模板类和数学公式就没贴了

```
/// Unpack ints from variadic template
/// The compile-time integer array, following RCInt is reverse of ints e.g.
///    G{5, 3, 2, 4, 2} the (0, G) is 2 and (4, G) is 5
template<int N, int... Tail>
struct RCInt;
template<int N, int Tail>
struct RCInt<N, Tail>
{
    enum { value = Tail };
};
template<int N, int Head, int... Tail>
struct RCInt<N, Head, Tail...>
{
    enum { value = (N == sizeof...(Tail)) ? Head : RCInt<N, Tail...>::value };
};
template<int N, int... Ints>
struct UnpackInts
{
    enum { value = RCInt<(int)sizeof...(Ints) - N - 1, Ints...>::value };
};

/// Type helper
template<typename I, int... Layers> struct NNType;
template<std::size_t... I, int... Layers>
struct NNType<std::index_sequence<I...>, Layers...>
{
    typedef /// Weights type
    std::tuple<
            Matrix<
                    double,
                    UnpackInts<I, Layers...>::value,
                    UnpackInts<I + 1, Layers...>::value
            >...
    > Weights;

    typedef /// Thresholds type
    std::tuple<
            Matrix<
                    double,
                    1,
                    UnpackInts<I + 1, Layers...>::value
            >...
    > Thresholds;
};

/// The neural network class
template<int... Layers>
class BPNeuralNet
{
```

```
    static const int N = sizeof...(Layers);
    using InMatrix = Matrix<double, 1, UnpackInts<0, Layers...>::value>;
    using OutMatrix = Matrix<double, 1, UnpackInts<N - 1, Layers...>::value>;
public:
    void init();

    template<class LX, class LY, class W, class T>
    void forward(LX& layerX, LY& layerY, W& weight, T& threshold);
    template<class LX, class LY, class W, class T, class DX, class DY>
    void reverse(LX& layerX, LY& layerY, W& weight, T& threshold, DX& deltaX, DY& delta

    template<std::size_t... I>
    bool train(const InMatrix& input, const OutMatrix& output, int times, std::index_se
    bool train(const InMatrix& input, const OutMatrix& output, int times = 1)
    {
        return train(input, output, times, std::make_index_sequence<N - 1>());
    }

    template<std::size_t... I>
    void simulate(const InMatrix& input, OutMatrix& output, std::index_sequence<I...>);
    void simulate(const InMatrix& input, OutMatrix& output)
    {
        simulate(input, output, std::make_index_sequence<N - 1>());
    }

public:
    std::tuple<Matrix<double, 1, Layers>...> m_layers;
    typename NNType<std::make_index_sequence<N - 1>, Layers...>::Weights m_weights;
    typename NNType<std::make_index_sequence<N - 1>, Layers...>::Thresholds m_threshold
    std::tuple<Matrix<double, 1, Layers>...> m_deltas;
    OutMatrix m_aberrmx;

public:
    double m_aberration = 0.001;
    double m_learnrate = 0.1;
};
```

模板类的实现

```
template<int... Layers>
void BPNeuralNet<Layers...>::init()
{
    for_each(m_weights, [](auto& weight) mutable
    {
        weight.random(0, 1);
    });

    for_each(m_thresholds, [](auto& threshold) mutable
    {
        threshold.random(0, 1);
    });
}

template<int... Layers>
template<class LX, class LY, class W, class T>
void BPNeuralNet<Layers...>::forward(LX& layerX, LY& layerY, W& weight, T& threshold)
{
```

```
    layerY.multiply(layerX, weight); /// layerY = layerX * weight
    layerY += threshold;
    layerY.foreach(logsig);
};

template<int... Layers>
template<class LX, class LY, class W, class T, class DX, class DY>
void BPNeuralNet<Layers...>::reverse(LX& layerX, LY& layerY, W& weight, T& threshold, D
{
    weight.adjustW(layerX, deltaY, m_learnrate);
    threshold.adjustT(deltaY, m_learnrate);
    /// 计算delta
    deltaX.multtrans(weight, deltaY);
    layerY.foreach(dlogsig);
    deltaX.hadamard(layerX);
};

template<int... Layers>
template<std::size_t... I>
bool BPNeuralNet<Layers...>::train(const InMatrix& input, const OutMatrix& output, int
{
    /// 1. 输入归一化
    auto& layer0 = std::get<0>(m_layers);
    layer0 = input;
    layer0.normaliz1();
    auto& layerN = std::get<N - 1>(m_layers);
    auto& deltaN = std::get<N - 1>(m_deltas);
    for(int i = 0; i < times; ++i)
    {
        /// 2. 正向传播
        using expander = int[];
        expander {(forward(std::get<I>(m_layers),
                           std::get<I + 1>(m_layers),
                           std::get<I>(m_weights),
                           std::get<I>(m_thresholds)),
                           0)...};
        /// 3. 判断误差
        double aberration = m_aberrmx.subtract(output, layerN).squariance() / 2;
        if (aberration < m_aberration) return true;
        /// 4. 反向修正
        deltaN.hadamard(m_aberrmx, layerN.foreach(dlogsig));
        expander {(reverse(std::get<N - I - 2>(m_layers),
                           std::get<N - I - 1>(m_layers),
                           std::get<N - I - 2>(m_weights),
                           std::get<N - I - 2>(m_thresholds),
                           std::get<N - I - 2>(m_deltas),
                           std::get<N - I - 1>(m_deltas)),
                           0)...};
    }
    return false;
}

template<int... Layers>
template<std::size_t... I>
void BPNeuralNet<Layers...>::simulate(const InMatrix& input, OutMatrix& output, std::in
{
    /// 1. 输入归一化
    auto& layer0 = std::get<0>(m_layers);
    layer0 = input;
```

```
78      layer0.normaliz1();
79      /// 2. 正向传播
80      using expander = int[];
81      expander {(forward(std::get<I>(m_layers),
82                         std::get<I + 1>(m_layers),
83                         std::get<I>(m_weights),
84                         std::get<I>(m_thresholds)),
85                         0)...};
86      /// 3. 输出结果
87      output = std::get<N - 1>(m_layers);
88  }
```


《一个C++14模板元实现的深度学习神经网络模板类，支持任意层数》有1个想法

Ethanm

2017年11月5日 上午11:16

赞赞赞！