Back-Propagation-Neural-Network

本项目是基于C++实现基础BP神经网络，有助于深入理解BP神经网络原理。

是对项目GavinTechStudio/bpnn_with_cpp的代码重构。

项目结构

```
.
├── CMakeLists.txt
├── README.md
├── data
│   ├── testdata.txt
│   └── traindata.txt
├── docs
│   └── formula.md
├── img
│   └── net-info.png
├── lib
│   ├── Config.h
│   ├── Net.cpp
│   ├── Net.h
│   ├── Utils.cpp
│   └── Utils.h
└── main.cpp

4 directories, 12 files
```

主要文件

- Net：网络具体实现
- Config：网络参数设置
- Utils：工具类
  - 数据加载
  - 激活函数
- main：网络具体应用

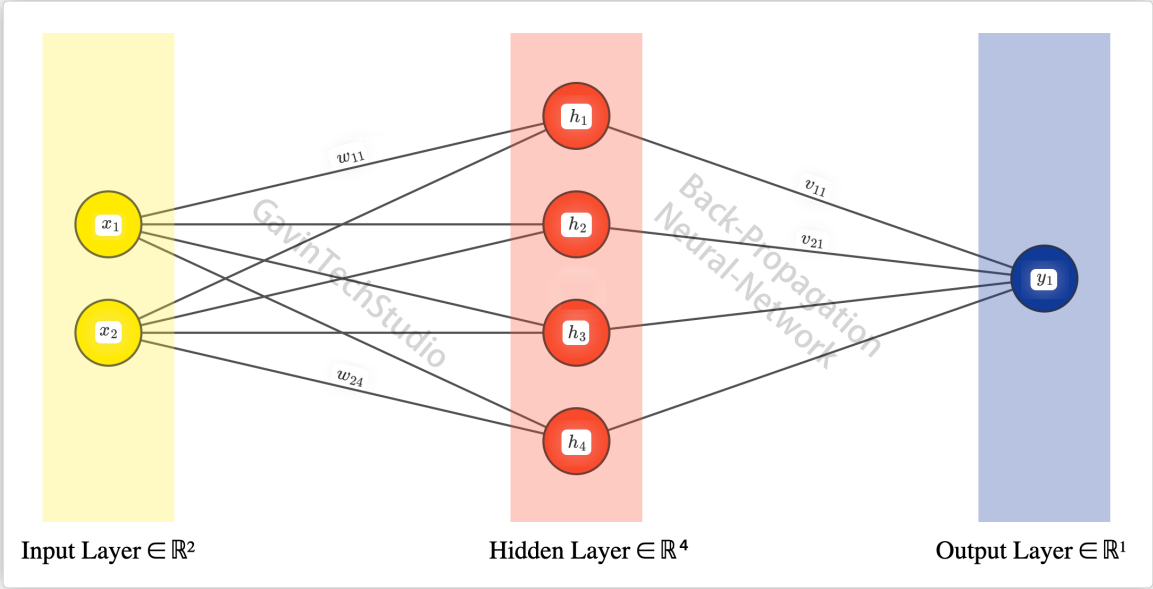
训练原理

具体公式推导请看视频讲解彻底搞懂BP神经网络 理论推导+代码实现（C++）哔哩哔哩bilibili

本部分文档包含大量数学公式，由于GitHub的README页面不支持数学公式渲染，推荐以下阅读方式：

1. 如果您使用的是Chrome、Edge、Firefox等浏览器，可以安装插件MathJax Plugin for Github（需要访问chrome web store）。
2. 使用**PDF**的方式进行阅读（**对网络无要求**）。
3. 使用**预渲染的静态网页**进行阅读（**对网络要求高，推荐**）。

0. 神经网络结构图



1. Forward（前向传播）

1.1 输入层向隐藏层传播

$$h_j = \sigma(\sum_i x_i w_{ij} - \beta_j)$$

其中h_j为第j个隐藏层节点的值，x_i为第i个输入层节点的值，w_{ij}为第i个输入层节点到第j个隐藏层节点的权重，β_j为第j个隐藏层节点偏置值，$\sigma(x)$为**Sigmoid**激活函数，后续也将继续用这个表达，其表达式如下

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

本项目中的代码实现如下：

```
for (size_t j = 0; j < Config::HIDENODE; ++j) {
    double sum = 0;
    for (size_t i = 0; i < Config::INNODE; ++i) {
        sum += inputLayer[i]->value *
               inputLayer[i]->weight[j];
    }
    sum -= hideLayer[j]->bias;

    hideLayer[j]->value = Utils::sigmoid(sum);
}
```

1.2 隐藏层向输出层传播

$$\hat{y_k} = \sigma(\sum_j h_j v_{jk} - \lambda_k)$$

其中$\hat{y_k}$为第k个输出层节点的值（预测值），h_j为第j个隐藏层节点的值，v_{jk}为第j个隐藏层节点到第k个输出层节点的权重，λ_k为第k个输出层节点的偏置值，$\sigma(x)$为激活函数。

本项目中的代码实现如下：

```
for (size_t k = 0; k < Config::OUTNODE; ++k) {
    double sum = 0;
    for (size_t j = 0; j < Config::HIDENODE; ++j) {
        sum += hideLayer[j]->value *
               hideLayer[j]->weight[k];
    }
    sum -= outputLayer[k]->bias;

    outputLayer[k]->value = Utils::sigmoid(sum);
}
```

2. 计算损失函数（Loss Function）

损失函数定义如下：

$$Loss = \frac{1}{2}\sum_{k}(y_k - \hat{y_k})^2$$

其中y_k为第k个输出层节点的目标值（真实值），$\hat{y_k}$为第k个输出层节点的值（预测值）。

本项目中的代码实现如下：

```
double loss = 0.f;

for (size_t k = 0; k < Config::OUTNODE; ++k) {
    double tmp = std::fabs(outputLayer[k]->value - out[k]);
    los += tmp * tmp / 2;
}
```

3. Backward（反向传播）

利用梯度下降法进行优化。

3.1 计算$\Delta\lambda_k$（输出层节点偏置值的修正值）

其计算公式如下（激活函数为Sigmoid时）：

$$\Delta\lambda_k = -\eta(y_k - \hat{y_k})\hat{y_k}(1 - \hat{y_k})$$

其中η为学习率（其余变量上方已出现过不再进行标注）。

本项目中的代码实现如下：

```
for (size_t k = 0; k < Config::OUTNODE; ++k) {
    double bias_delta =
        -(out[k] - outputLayer[k]->value) *
        outputLayer[k]->value *
        (1.0 - outputLayer[k]->value);

    outputLayer[k]->bias_delta += bias_delta;
}
```

3.2 计算Δv_{jk}（隐藏层节点到输出层节点权重的修正值）

其计算公式如下（激活函数为Sigmoid时）：

$$\Delta v_{jk} = \eta(y_k - \hat{y}_k)\hat{y}_k(1 - \hat{y}_k)h_j$$

其中h_j为第j个隐藏层节点的值（其余变量上方已出现过不再进行标注）。

本项目中的代码实现如下：

```
for (size_t j = 0; j < Config::HIDENODE; ++j) {
    for (size_t k = 0; k < Config::OUTNODE; ++k) {
        double weight_delta =
            (out[k] - outputLayer[k]->value) *
            outputLayer[k]->value *
            (1.0 - outputLayer[k]->value) *
            hideLayer[j]->value;

        hideLayer[j]->weight_delta[k] += weight_delta;
    }
}
```

3.3 计算$\Delta \beta_j$（隐藏层节点偏置值的修正值）

其计算公式如下（激活函数为Sigmoid时）：

$$\Delta \beta_j = -\eta \sum_k (y_k - \hat{y}_k)\hat{y}_k(1 - \hat{y}_k)v_{jk}h_j(1 - h_j)$$

其中v_{jk}为第j个隐藏层节点到第k个输出层节点的权重（其余变量上方已出现过不再进行标注）。

本项目中的代码实现如下：

```
for (size_t j = 0; j < Config::HIDENODE; ++j) {
    double bias_delta = 0.f;
    for (size_t k = 0; k < Config::OUTNODE; ++k) {
        bias_delta +=
            -(out[k] - outputLayer[k]->value) *
            outputLayer[k]->value *
            (1.0 - outputLayer[k]->value) *
            hideLayer[j]->weight[k];
    }
    bias_delta *=
        hideLayer[j]->value *
        (1.0 - hideLayer[j]->value);

    hideLayer[j]->bias_delta += bias_delta;
}
```

3.4 计算Δw_{ij}（输入层节点到隐藏层节点权重的修正值）

其计算公式如下（激活函数为Sigmoid时）：

$$\Delta w_{ij} = \eta \sum_k (y_k - \hat{y}_k)\hat{y}_k(1 - \hat{y}_k)v_{jk}h_j(1 - h_j)x_i$$

其中x_i为第i个输入层节点的值（其余变量上方已出现过不再进行标注）。

本项目中的代码实现如下：

```
for (size_t i = 0; i < Config::INNODE; ++i) {
    for (size_t j = 0; j < Config::HIDENODE; ++j) {
        double weight_delta = 0.f;
        for (size_t k = 0; k < Config::OUTNODE; ++k) {
            weight_delta +=
                (out[k] - outputLayer[k]->value) *
                outputLayer[k]->value *
                (1.0 - outputLayer[k]->value) *
                hideLayer[j]->weight[k];
        }
        weight_delta *=
            hideLayer[j]->value *
            (1.0 - hideLayer[j]->value) *
            inputLayer[i]->value;

        inputLayer[i]->weight_delta[j] += weight_delta;
    }
}
```