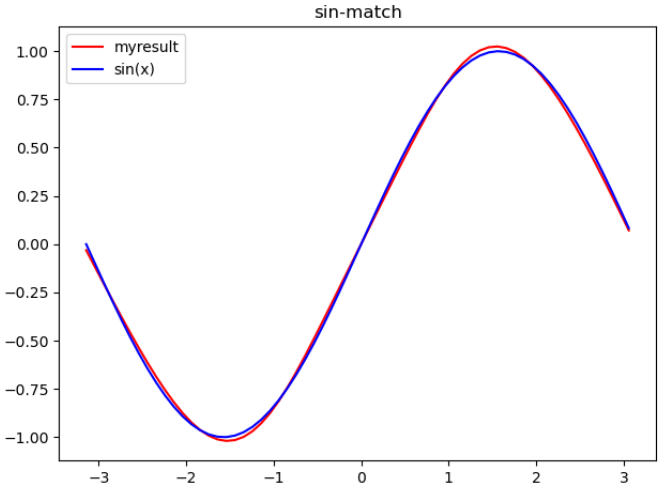
智能系统原理与开发 Lab1-part1

18302010018 俞哲轩

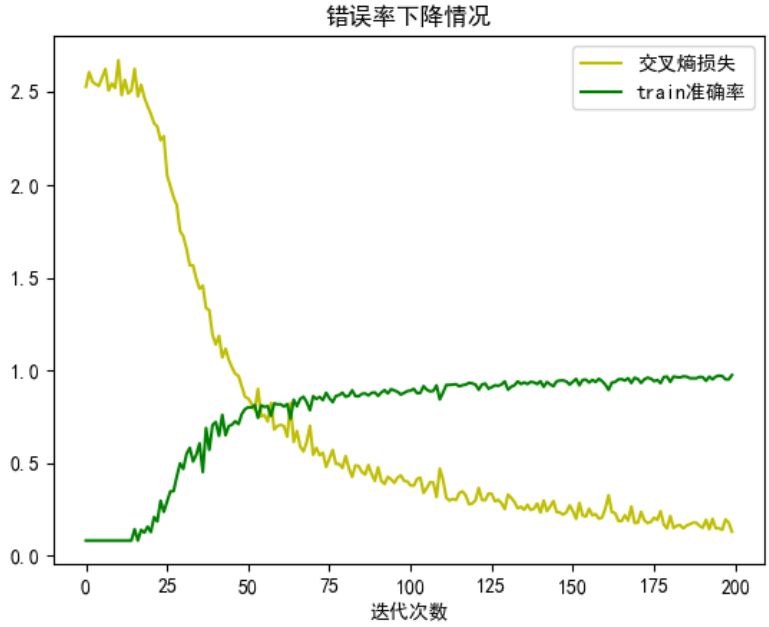
- 实验结果
  - sin函数拟合
  - 12个手写汉字分类
- 代码基本架构
  - Network
  - Layer
  - Neuron
  - Util
- 不同网络架构、网络参数的实验比较
  - 隐层节点个数的影响
  - leanring rate的影响
  - batch的影响
- 数据增强
  - 第一步
  - 第二步
- 正则项
- 更换激活函数和初始化方式
  - Tanh & LeCun/Xavier
  - LeRU & LeCun/Xavier
  - LeRU & 随机初始化
  - 浅谈sigmoid、LeRU、tanh三者的对比
- 对反向传播算法的理解
  - 整体理解
  - 反向传播的推导

实验结果

sin函数拟合



12个手写汉字分类



代码基本架构

```
lab1-part1 — -zsh — 90×25
Last login: Sat Oct 31 13:04:29 on ttys001
(base) yuzhexuan@Yu-MacBook-Pro lab1-part1 % tree
.
├── lab1-part1.iml
├── src
|   ├── Classification.java
|   ├── Layer.java
|   ├── Network.java
|   ├── Neuron.java
|   ├── Regression.java
|   ├── Test.java
|   └── Util.java
└── train
    ├── 1
    |   ├── 1.bmp
    |   ├── 10.bmp
    |   ├── 100.bmp
    |   ├── 101.bmp
    |   ├── 102.bmp
    |   ├── 103.bmp
    |   ├── 104.bmp
    |   ├── 105.bmp
    |   ├── 106.bmp
    |   ├── 107.bmp
    |   ├── 108.bmp
```

Network

实现神经网络类，字段包括网络层数、每层神经网络，方法包括向前传播、向后传播算法以及softmax

```
public class Network implements Serializable {
  int size;
  List<Layer> layers;

  public Network(int[] structure, boolean classification, double weightLR,
double biasLR)
```

```
  public void forward(double[] input) {
    Layer layer = layers.get(0);
    Neuron[] neurons = layer.neurons;
    for (int i = 0; i < neurons.length; i++) {
      neurons[i].output = input[i]; // Input Layer
    }
    for (int i = 1; i < size; i++) {
      layers.get(i).forward();
    }
  }

  public void backward(double[] desired) {
    Layer layer = layers.get(size - 1);
    Neuron[] neurons = layer.neurons;
    for (int i = 0; i < neurons.length; i++) {
      neurons[i].delta = desired[i] - neurons[i].output; // Output Layer
    }
    for (int i = layers.size() - 1; i > 0; i--) {
      layers.get(i).backward(); // Output Layer & Hidden Layer
    }
  }

  void softmax() {
    Layer layer = layers.get(size - 1);
    Neuron[] neurons = layer.neurons;
    double totalOutput = 0;
    for (Neuron neuron : neurons) {
      totalOutput += Math.exp(neuron.output);
    }
    for (Neuron neuron : neurons) {
      neuron.output = Math.exp(neuron.output) / totalOutput;
    }
  }
}
```

Layer

实现神经元层类，字段包括神经元的个数、包含的神经元、前一神经元层、后一神经元层和学习率，方法包括向前传播和向后传播算法

```
public class Layer implements Serializable {
  //static final double LAMBDA = 0.00001;
  //static final double lambdaLR = 0.005;
  Random rd = new Random();
  int quantity;
  Neuron[] neurons;
  Layer backLayer;
```

```
  Layer nextLayer;
  double weightLR;
  double biasLR;

  public Layer(int quantity, Layer backLayer, boolean classification, double
weightLR, double biasLR)

  void forward() {
    Neuron[] backLayerNeurons = backLayer.neurons;
    for (Neuron neuron : neurons) {
      double sum = 0;
      for (int i = 0; i < neuron.weights.length; i++) {
        sum += neuron.weights[i] * backLayerNeurons[i].output;
      }
      sum += neuron.bias;
      if (nextLayer == null) { // Output Layer
        neuron.output = sum;
      } else { // Hidden Layer
        neuron.output = Neuron.sigmoid(sum);
        //neuron.output = Neuron.tanh(sum);
        //neuron.output = Neuron.LeRU(sum);
      }
    }
  }

  void backward() {
    Neuron[] backLayerNeurons = backLayer.neurons;
    Neuron[] nextLayerNeurons;
    if (nextLayer == null) { // Output Layer
      for (Neuron neuron : neurons) {
        double gradient = 1;
        gradient *= neuron.delta;
        for (int i = 0; i < neuron.weights.length; i++) {
          //neuron.weights[i] *= lambdaLR * (1 - LAMBDA);
          neuron.weights[i] += weightLR * gradient *
backLayerNeurons[i].output;
        }
        neuron.bias += biasLR * gradient;
      }
    } else { // Hidden Layer
      nextLayerNeurons = nextLayer.neurons;
      for (int i = 0; i < neurons.length; i++) {
        double gradient = 0;
        for (Neuron nextLayerNeuron : nextLayerNeurons) {
          gradient += nextLayerNeuron.delta * nextLayerNeuron.weights[i];
        }
        gradient *= Neuron.derivativeSigmoid(neurons[i].output);
        //gradient *= Neuron.derivativeTanh(neurons[i].output);
        //gradient *= Neuron.derivativeLeRU(neurons[i].output);
```

```
        neurons[i].delta = gradient;
        for (int j = 0; j < neurons[i].weights.length; j++) {
          //neurons[i].weights[i] *= lambdaLR * (1 - LAMBDA);
          neurons[i].weights[j] += weightLR * gradient *
backLayerNeurons[j].output;
        }
        neurons[i].bias += biasLR * gradient;
      }
    }
  }
}
```

Neuron

实现神经元类，字段包括神经元的权重、偏移量、输出和改变量，方法包括各种激活函数和导函数

```
public class Neuron implements Serializable {
  Random rd = new Random();
  double bias;
  double[] weights;
  double output;
  double delta;

  public Neuron()
  public Neuron(double bias, int backLayerQuantity, boolean classification)
  static double sigmoid(double x)
  static double derivativeSigmoid(double x)
  static double tanh(double x)
  static double derivativeTanh(double x)
  static double LeRU(double x)
  static double derivativeLeRU(double x)
}
```

Util

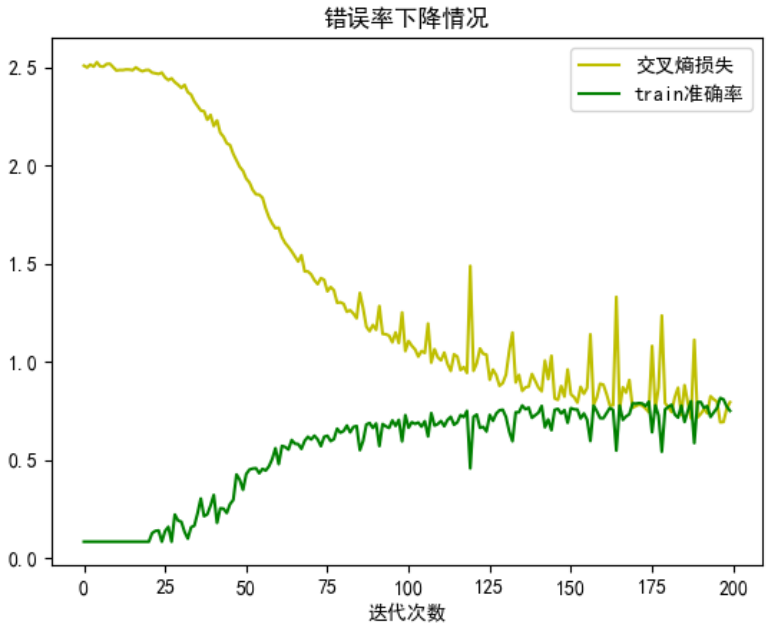
实现工具类，方法包括保存/加载网络、等间隔采样、随机采样和读取图片内容

```
public class Util {
  static Random rd = new Random();
  public static void saveNetwork(String path, Network network)
  public static Network loadNetwork(String path)
  public static double[] linspace(double start, double end, int total)
  public static double[] random(double start, double end, int total)
  public static double[] imgInfo(String src, String position, int offset)
}
```

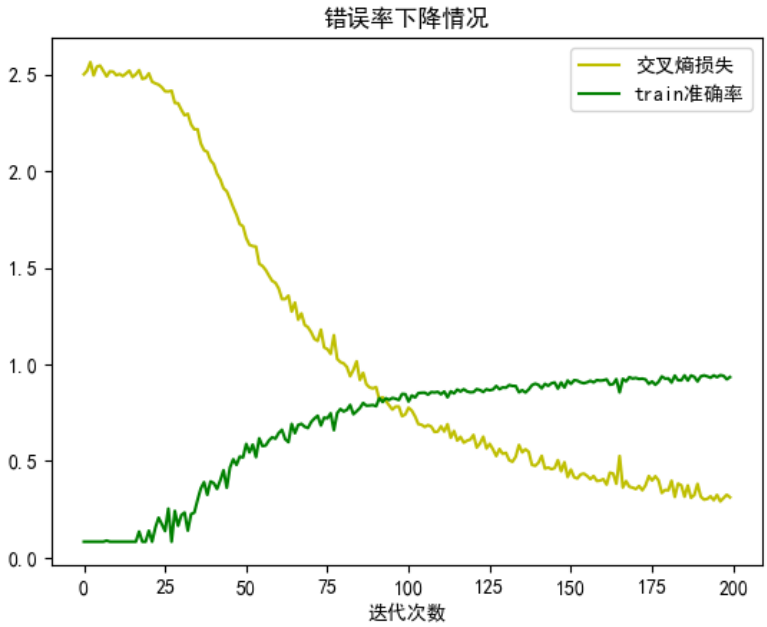
不同网络架构、网络参数的实验比较

隐层节点个数的影响

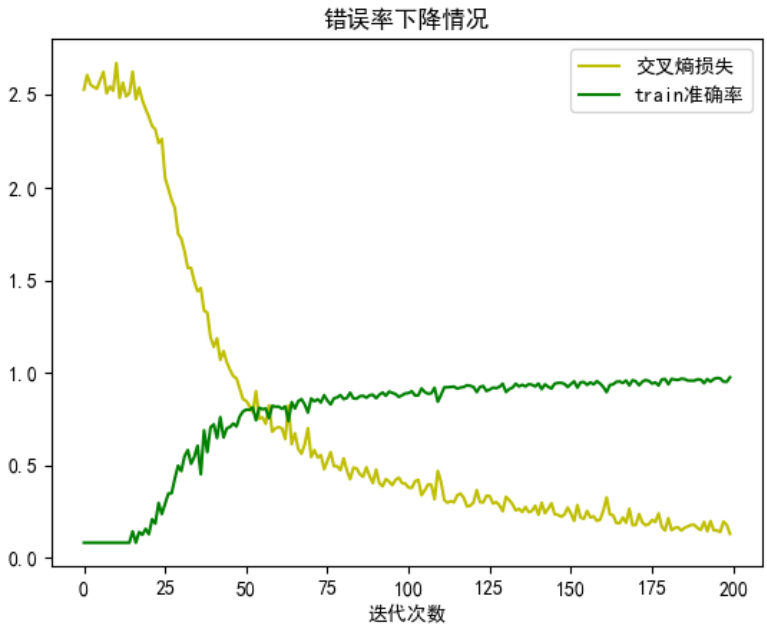
> 1个隐层4个节点 学习率0.05 batch32
>
> 验证集准确率：51%



> 1个隐层8个节点 学习率0.05 batch32
>
> 验证集准确率：71%

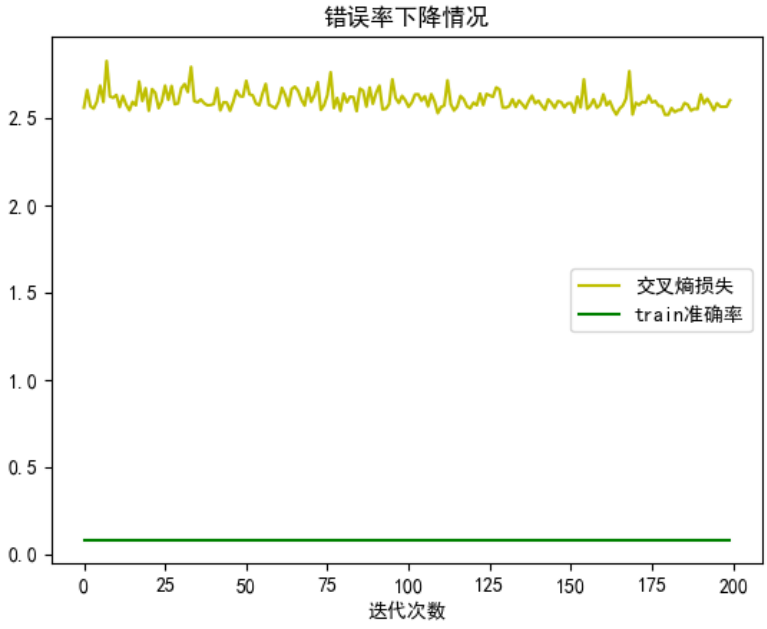


> 1个隐层64个节点 学习率0.05 batch32（标准模型）
>
> 验证集准确率：83%



由此可以看出，节点越多，收敛越快，即学习能力越强；当节点个数不够多时，可能无法收敛，即无法有效学习出结果。

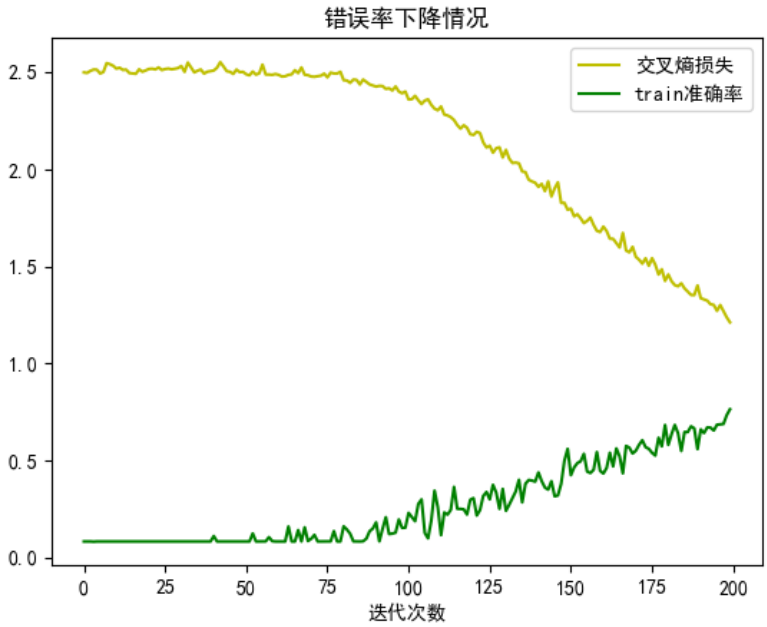
> 2个隐层各128个节点 学习率0.05 batch32
>
> 验证集正确率：0%



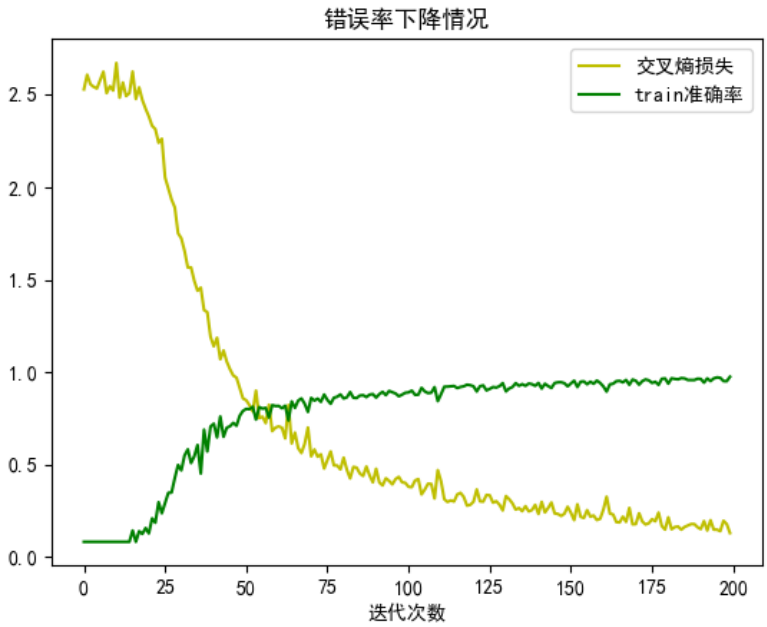
网络的层数越多理论上学习能力就会越强，精度越高，但也使网络更加复杂，大大增加了网络的训练时间，而且输入值过大，可能引起反向传播的时候梯度消失的情况，导致网络学习不出来；也极有可能产生过拟合的情况，训练结果反而不如网络结构较简单的网络。

leanring rate的影响

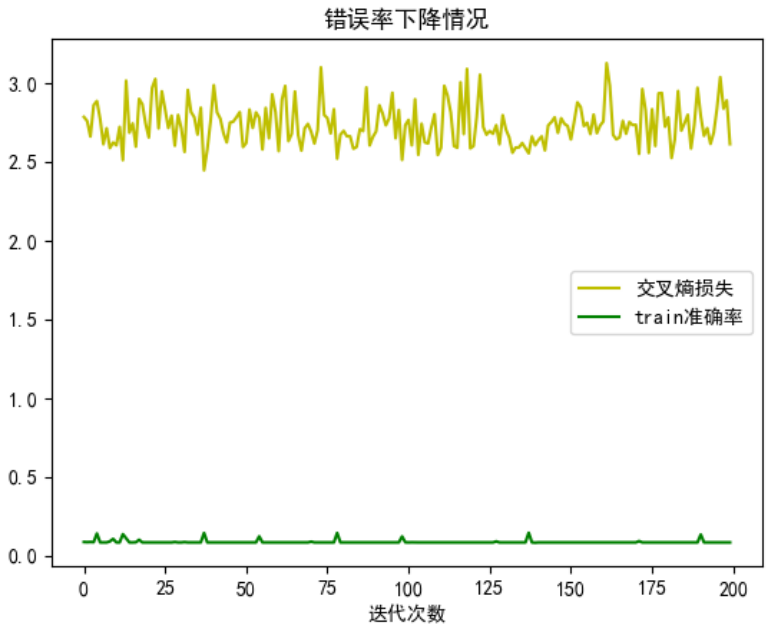
> 1个隐层64个节点 学习率0.01 batch32



> 1个隐层64个节点 学习率0.05 batch32（标准模型）
>
> 验证集准确率：83%



> 1个隐层64个节点 学习率0.8 batch32



在梯度下降优化的过程中，如果学习率设置的过小，会使得模型优化速度变得很慢；如果学习率设置的过大，则容易导致模型过拟合。

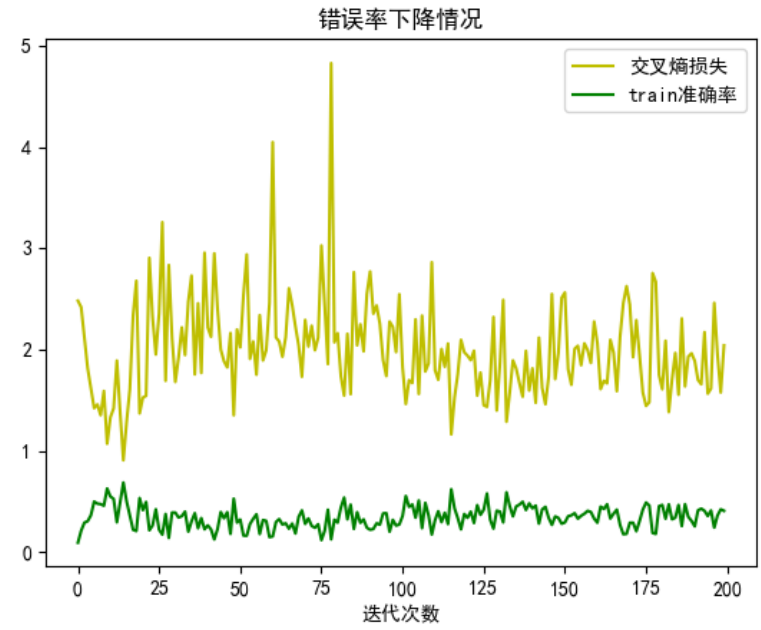
比如一开始设置的学习率为0.01，导致模型迭代的前75次Error值也没有太大的变化，梯度下降极其缓慢；标准模型的学习率为0.05；如果大于0.05，我们可以看到在收敛的过程中有较大波动，如果进一步上升至0.8，则不会收敛。

一个合理的学习率既可以影响学习速度，又可以影响学习效果：

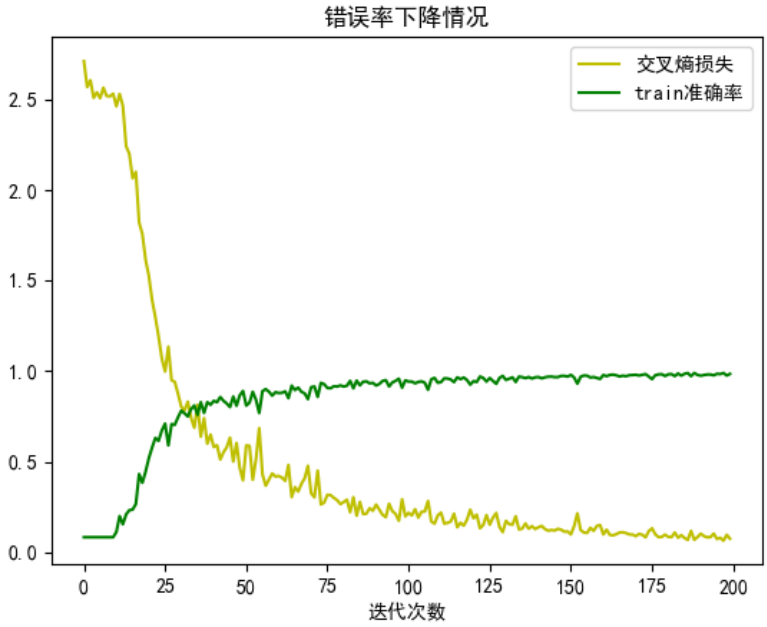
学习率	迭代总次数	平均误差（绝对值）
0.001	20000000	1.02%
0.001	50000000	0.24%
0.005	20000000	0.15%
0.02	10000000	0.24%
0.02	20000000	0.12%
0.03	10000000	0.14%
0.05	10000000	0.28%
0.1	10000000	0.70%

batch的影响

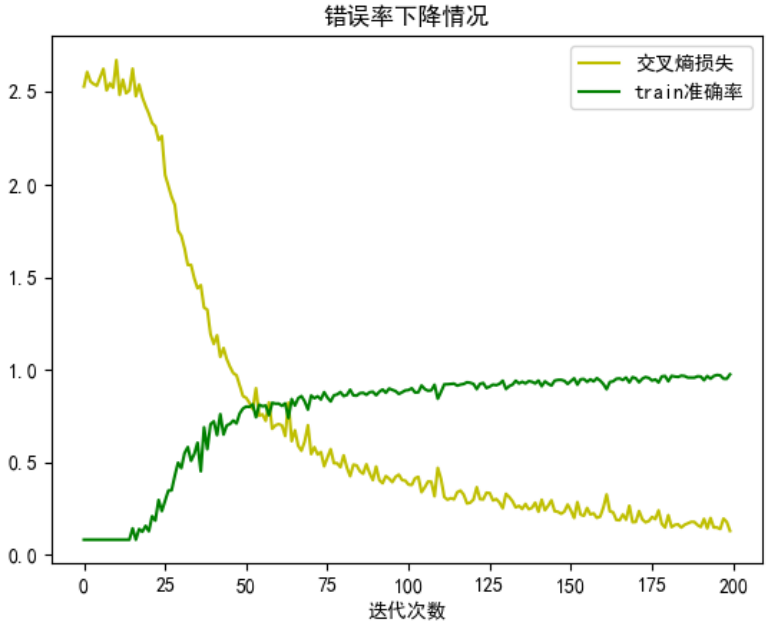
> 1个隐层64个节点 学习率0.05 batch1



> 1个隐层64个节点 学习率0.05 batch16



> 1个隐层64个节点 学习率0.05 batch32（标准模型）
>
> 验证集准确率：83%



标准BP算法每次更新只针对单个样例，参数更新得非常频繁，而且对不同样例进行更新的效果可能出现**“抵消”**现象。因此，为了达到同样的累计误差极小点，标准BP算法往往需要进行**更多次的迭代**。累积BP算法，即batch，直接针对累积误差最小化，它在读取整个训练集D一遍后才对参数进行更新，其参数更新频率低得多。

通过实验我们发现在batch大小为1时，模型在前200次迭代时**不能良好收敛**；batch大小为16时，模型已经可以收敛，但**波动较大**；batch大小在16以上时，模型可以**较平稳的收敛**，且收敛的迭代次数会随着batch的增大降低。

这也表明了，当batch的大小太小时，每次修正梯度都是以各自样本的梯度方向修正，随机性太强；随着batch增大，可以降低达到精度要求的迭代次数；但是当batch过大了时候，虽然一次迭代时间会下降，但是达到精度要求所需要的迭代数量反而会再次上升。

在实际任务中，累计误差下降到一定程度之后，进一步下降会非常缓慢，这时应该采用标准BP算法以更快获得更好的解。

数据增强

> 通过已有的数据的几何变换，扩充数据集
>
> 增加训练的数据量，提高模型的泛化能力

由于每种文字仅有620张图片，数据量较小，很可能制约了模型分类的准确性，所以想到通过合理的方法来扩充数据集，因而采用了在计算机视觉领域常用的方法：数据增强。

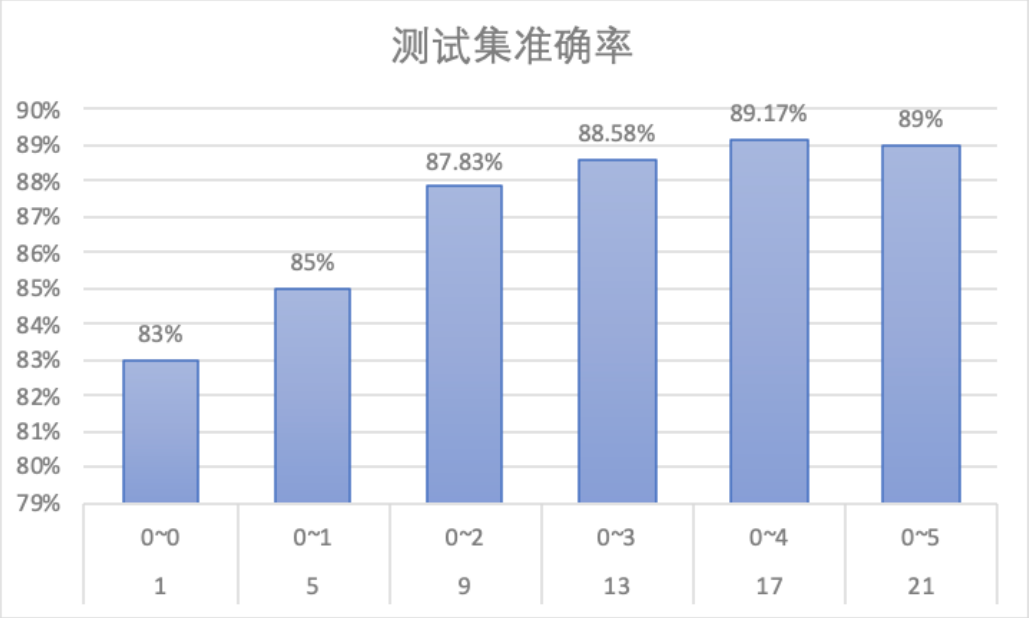
在分析了数据集之后，发现大部分手写文字，都位于图片的中央，且边缘四周会有留白，所以决定采用平移方法（Translations）进行数据增强。

第一步

在原图片的基础上，向上、下、左、右四个方向分别平移**n**个单位像素，故扩充后的数据集变为原有的**5倍**，分类准确率也有显著的提高（每项均经过三次实验，取平均值）：

第二步

在第一步的基础上，向上、下、左、右四个方向依次平移0到n个单位像素，故扩充后的数据集变为原有的$4*n+1$**倍**，分类准确率随n的增大有先增大后减小的趋势（每项均经过三次实验，取平均值）：



具体实现代码如下：

```
/**
* @param position: translation position
* @param offset: offset pixel
*/
```

```
private static int[] convertImageToArray(BufferedImage bf, String position, int
offset) {
  int width = bf.getWidth();
  int height = bf.getHeight();
  int[] data = new int[width * height];
  switch (position) {
    case "center":
      bf.getRGB(0, 0, width, height, data, 0, width);
      break;
    case "left":
      bf.getRGB(offset, 0, width - offset, height, data, offset, width);
      break;
    case "right":
      bf.getRGB(0, 0, width - offset, height, data, 0, width);
      break;
    case "up":
      bf.getRGB(0, offset, width, height - offset, data, 0, width);
      break;
    case "down":
      bf.getRGB(0, 0, width, height - offset, data, 0, width);
      break;
  }
  return data;
}
```

正则项

在误差目标函数中增加一个用于描述网络复杂度的部分，例如权重与偏移量的平方和，则误差目标函数变为：

$$Error = \lambda \frac{1}{m} \sum_{k=1}^{m} Error_k + (1-\lambda) \sum_i w_i^2$$

更新权重的方式如下：

```
static final double LAMBDA = 0.00001;
static final double lambdaLR = 0.005;

neurons[i].weights[i] *= lambdaLR * (1 - LAMBDA);
neurons[i].weights[j] += weightLR * gradient * backLayerNeurons[j].output;
```

更换激活函数和初始化方式

除了**sigmoid函数**以外，还尝试了其它激活函数，如**LeRU函数**、**tanh函数**等。

sigmoid函数及其求导：

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

$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = f(x) * (1 - f(x))$

```
static double sigmoid(double x) { return 1 / (1 + Math.exp(-x)); }
static double derivativeSigmoid(double x) { return x * (1 - x); }
```

LeRU函数及其求导：

$f(x) = 0(x \leq 0),\ f(x) = x(x > 0)$

$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 0(x \leq 0),\ \frac{\mathrm{d}f(x)}{\mathrm{d}x} = 1(x > 0)$

```
static double LeRU(double x) { return Math.max(0, x); }
static double derivativeLeRU(double x) { return x > 0 ? 1 : 0; }
```

tanh函数及其求导：

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

$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 1 - f^2(x)$

```
static double tanh(double x) { return (Math.exp(x) - Math.exp(-x) / Math.exp(x)
+ Math.exp(-x)); }
static double derivativeTanh(double x) { return 1 - x * x; }
```

在初始化方面，尝试了**LeCun初始化**和**Xavier初始化**。

LeCun初始化：

将各层神经元的权重初始化为服从$weight \sim N(0,\ \mu^2)$的正态分布的值，其中$\mu^2 = quantity_{layer-1}$，即前一层的神经元个数，保证每一层的输出平均值为0，尽可能的让输入和输出服从相同的分布，这样就能够避免后面层的激活函数的输出值趋向于0。

```
weights[i] = rd.nextGaussian() / Math.sqrt(backLayerQuantity);
```

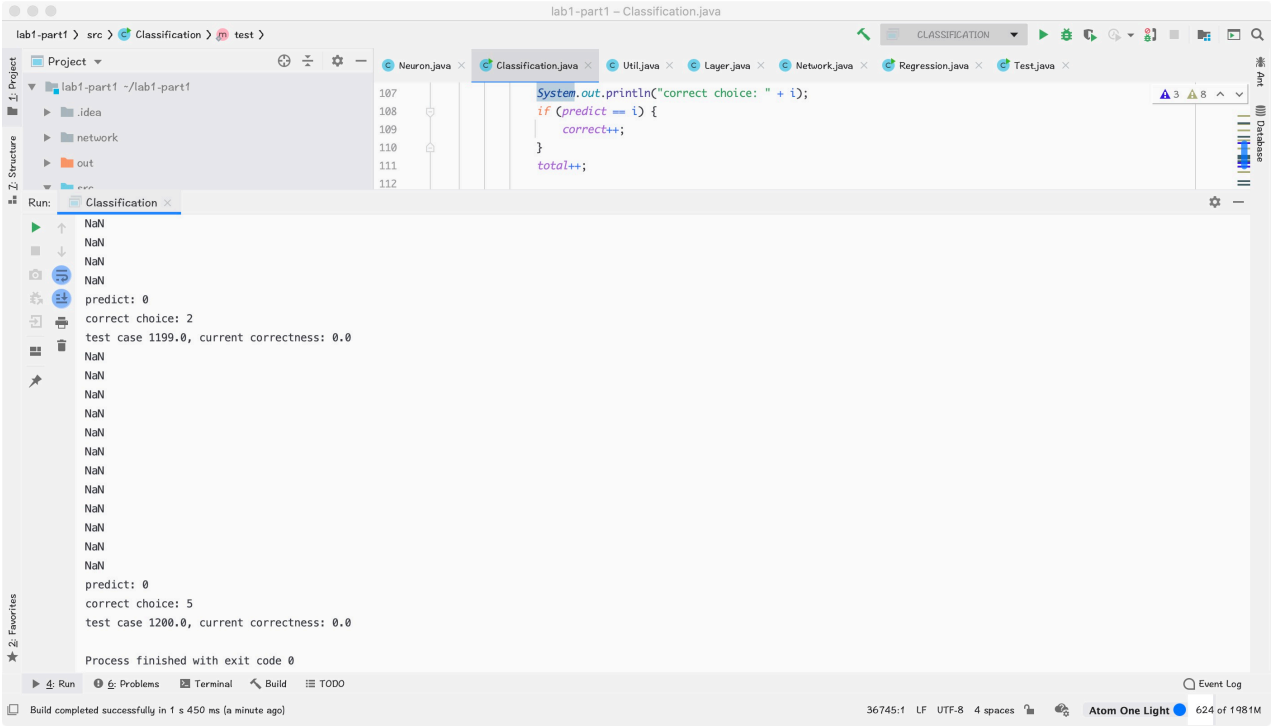
Xavier初始化：

将各层神经元的权重初始化为服从$weight \sim N(0,\ 2 * \mu^2)$的正态分布的值，其中$\mu^2 = quantity_{layer-1} + quantity_{layer}$，即前一层的神经元个数，保证每一层的输出平均值为0，尽可能的让输入和输出服从相同的分布，这样就能够避免后面层的激活函数的输出值趋向于0。

```
weights[i] = rd.nextGaussian() / Math.sqrt(backLayerQuantity + quantity);
```

Tanh & LeCun/Xavier

Xavier初始化最先被提出来的时候，搭配的是tanh激活函数，以期望该神经元层输出平均值为0，且输出的方差为输入的方差，所以一开始尝试采用了tanh函数与Xavier搭配的方式：



~~（训练效果异常效果感人）~~

究其原因，可能是初始化的权重值大于先前随机初始化的权重值，且tanh函数的求导，最大值为1，反向传播的幅度远大于sigmoid函数，容易将权重值更改的很大，导致运算数NaN💥。

LeRU & LeCun/Xavier

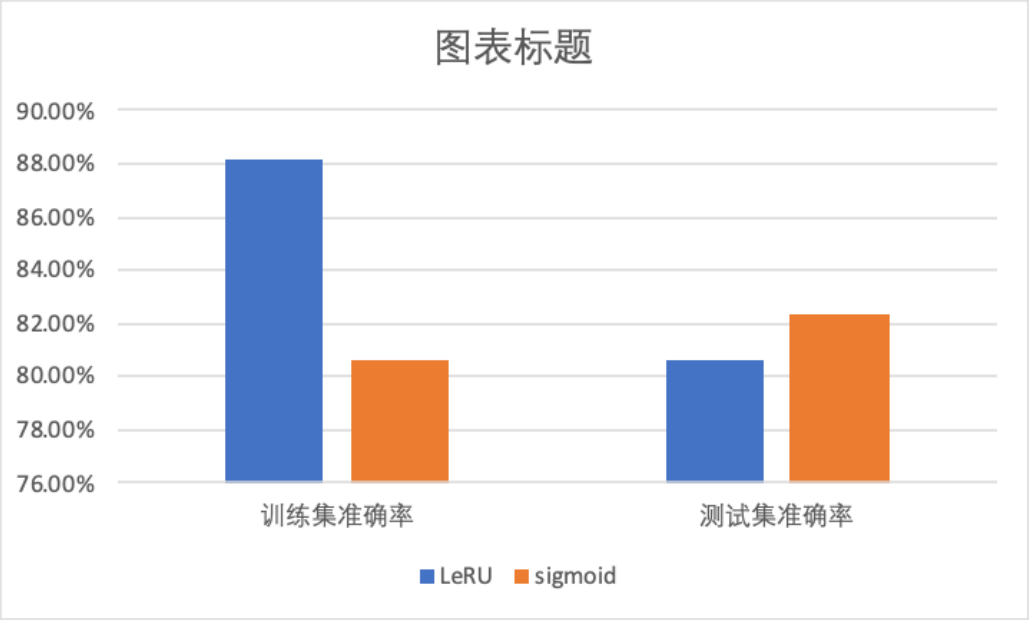
在LeRU函数配合Xavier初始化方法的时候，也会有类似的问题：LeRU函数的导数最大值也为1，尽管在反向传播中可以很好的将梯度从后向前传播，但是也容易将权重值更改的很大。

经测试，在采用LeRU作为激活函数的时候，`weight learning rate`通常是sigmoid激活函数的`weight learning rate`的**40%~50%**。

LeRU & 随机初始化

为了检测LeRU和sigmoid两种激活函数对于分类问题的准确率是否有提升，进行了对照实验。

实验中的权重和偏移量均为随机初始化，只有激活函数和learning rate的区别（每项均经过三次实验，取平均值）：



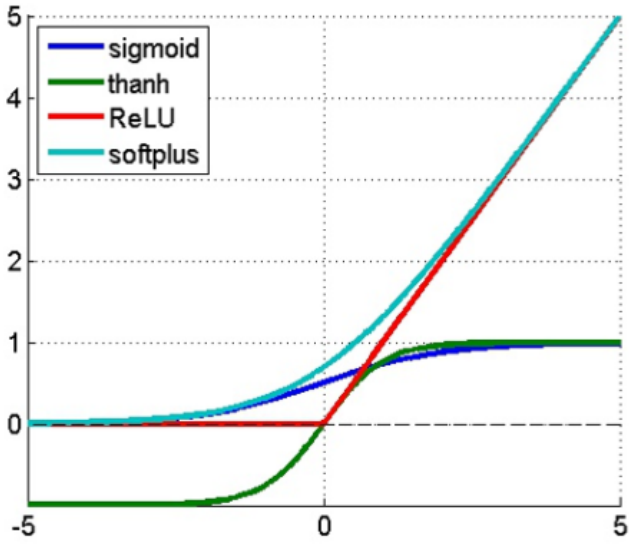
从图中我们可以看出，在训练集准确率上，LeRU激活函数明显高于sigmoid激活函数，但是在测试集准确率上，LeRU激活函数略低于sigmoid激活函数。

这是因为：LeRU激活函数由于导数取值比sigmoid激活函数大，在训练初期，能**更加"原汁原味"**地反向传播调整权重值和偏移量值，所以能**更加快速**的调整到一个具有合理的权重值和偏移量值的网络——即LeRU激活函数训练初期的速度**优于**sigmoid激活函数。

但是对于分类的准确率来说，两者并没有显著的差别，甚至sigmoid激活函数要**略优于**LeRU激活函数。

综上所述，更换激活函数和权重初始化方式，效果更多的是让**训练速度更快**，而非**训练效果更优**，想要达到更优的训练效果，需要更加复杂的网络优化，譬如改变网络结构、使用优化器等。

浅谈sigmoid、LeRU、tanh三者的对比

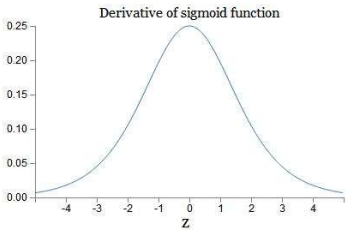


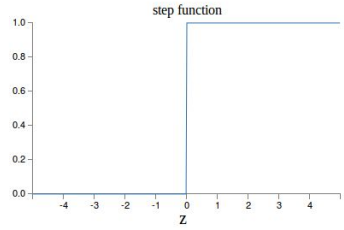
1. sigmoid在压缩数据幅度方面有优势（因为输出在0到1之间），对于深度网络，使用sigmoid可以保证数据幅度不会有问题，这样数据幅度稳住了就不会出现太大的失误；但是sigmoid存在梯度消失的问题，在反向传播上有劣势，所以在优化的过程中存在不足；
2. ReLU不会对数据做幅度压缩，所以如果数据的幅度不断扩张，那么模型的层数越深，幅度的扩张也会越厉害，最终会影响模型的表现；但是ReLU在反向传导方面可以很好地将"原汁原味"的梯度传

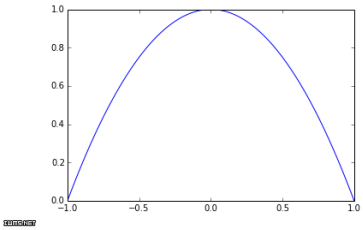
到后面，这样在学习的过程中可以更好地发挥出来，加快初期的学习速度；

3. tanh的以原点成中心对称，当数据足够大、选择合适的初始化值，可以使得输出的平均值为0，更加有利于提高训练效率；由于sigmoid和ReLU输出是在0-1之间，总是正数，在训练过程中参数的梯度值为同一符号，这样更新的时候容易出现震荡现象（zigzag），不容易到达最优值。

综上所述，sigmoid对比ReLU而言，前向更优；ReLU对比sigmoid而言，后向更优。







对反向传播算法的理解

整体理解

BP算法的工作流程（伪代码）：

```
input:
训练集D = {(x_k, y_k), k in (1, m)}
学习率r
procedure:
1.在（0, 1）范围内随机初始化网络中的所有连接权重和偏移量
2.repeat
3.  for all (x_k, y_k) in D do
4.    计算输出y_k
5.    计算输出层神经元的梯度项
6.    计算隐层神经元的梯度项
7.    更新权重和偏移量
8.  end for
9.until 达到停止条件
ouput: 权重和偏移量稳定的神经网络
```

BP算法的目标，是要**最小化**训练集D上的累积误差$Error = \frac{1}{m}\sum_{k=1}^{m} Error_k$，标准BP算法每次仅针对一个训练样例更新权重和偏移量。

如果推导出基于累积误差最小化的更新规则，就得到了累计误差反向传播算法，即引入了batch。

累积BP算法与标准BP算法都分常用，一般来说，标准BP算法每次更新只针对单个样例，参数更新得非常频繁，而且对不同样例进行更新的效果可能出现“抵消”现象。因此，为了达到同样的累计误差极小点，标准BP算法往往需要进行更多次的迭代。累积BP算法直接针对累积误差最小化，它在读取整个训练集D一遍后才对参数进行更新，其参数更新频率低得多。但在很多任务中，累计误差下降到一定程度之后，进一步下降会非常缓慢，这时标准BP算法往往会更快获得更好的解，尤其是在训练集D非常大时更明显。

[Hornik et al., 1989]证明，只需一个包含足够多神经元的隐层，多层前馈网络就能以任意精度逼近任意复杂度的连续函数。然而，如何设置隐层神经元的个数仍是个未决问题，实际应用中通常靠“试错法”调整。

因为BP网络其强大的表示能力，它经常遭遇过拟合的问题，其训练误差持续降低，但是测试误差却可能上升。有**两种策略**常用来缓解BP网络的过拟合。**第一种策略是“早停”（early stopping）**：将数据分成训练集和验证集，训练集用来计算梯度、更新权重和偏移量，验证集用来估计误差；如果训练集误差降低但是验证集误差升高，则停止训练，返回具有最小验证集误差的网络；**第二种策略是“正则化”（regularization）**：在误差目标函数中增加一个用于描述网络复杂度的部分，例如权重与偏移量的平方和，则误差目标函数变为：

$$Error = \lambda \frac{1}{m} \sum_{k=1}^{m} Error_k + (1-\lambda) \sum_i w_i^2$$

反向传播的推导

$$Error = \frac{1}{2} \sum_{i \in output} (d_i - y_i)^2$$

$$w_{ji} = w_{ji} - r \frac{\partial Error}{\partial w_{ji}}$$

$$net_j = \sum_i w_{ji} x_{ji}$$

$$\frac{\partial Error}{\partial w_{ji}} = \frac{\partial Error}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \frac{\partial Error}{\partial net_j} \frac{\partial \sum_i w_{ji} x_{ji}}{\partial w_{ji}} = \frac{\partial Error}{\partial net_j} x_{ji}$$，x_{ji}为节点i的输出值

输出层：

$$\frac{\partial Error}{\partial net_j} = \frac{\partial Error}{\partial y_j} \frac{\partial y_j}{\partial net_j}$$

$$\frac{\partial Error}{\partial y_j} = \frac{\partial \frac{1}{2} \sum_{i \in output} (d_i - y_i)^2}{\partial y_j} = -(d_i - y_i)$$

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial sigmoid(net_j)}{\partial net_j} = y_j(1-y_j)$$

$$\frac{\partial Error}{\partial net_j} = -(d_i - y_i) y_j (1-y_j)$$

令$\delta_j = -\frac{\partial Error}{\partial net_j}$

则$\delta_j = (d_i - y_i) y_j (1-y_j)$

δ_j可复用，在拟合问题中，输出层不需要经过sigmoid激活，所以反向传播也不需要求导

隐层：

$$\frac{\partial Error}{\partial net_j} = \sum_{k \in downstream(j)} \frac{\partial Error}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial net_k}{\partial a_j} \frac{\partial a_j}{\partial net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k w_{kj} \frac{\partial a_j}{\partial net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k w_{kj} a_j (1 - a_j)$$

$$= -a_j (1 - a_j) \sum_{k \in downstream(j)} \delta_k w_{kj}$$

令$\delta_j = -\frac{\partial Error}{\partial net_j}$

则$\delta_j = a_j (1 - a_j) \sum_{k \in downstream(j)} \delta_k w_{kj}$