Author: Zhibin Jiang
zhibin.jiang2023@gmail.com
This document is translated by AI, and the original language is **Simplified Chinese**

How to implement a neural network for handwritten digit recognition in Java

Perface

The data source comes from the **MNIST** handwritten database. There are 60,000 data items as the training set and 10,000 data items as the validation set. After training with the Adam optimizer and L1, L2 regularization, the accuracy rate can reach about 95% (not very high, **:D** , many aspects that can be optimized haven't been written well yet).

If you have just started to learn about neural networks like me, I strongly recommend you to watch these videos, and then you will understand the main content about backpropagation that I will talk about below.

These videos have helped me a lot during my learning period, and there are no obscure or difficult parts in them! (In the order of viewing)

1. [3B1B] But what is a neural network? | Deep learning chapter 1 - YouTube
2. [3B1B] Gradient descent, how neural networks learn | DL2 - YouTube
3. [3B1B] Backpropagation, step-by-step | DL3 - YouTube
4. Gradient Descent Algorithm
5. Backpropagation Algorithm
6. Backpropagation Visualization
7. Adam Optimizer
8. L1 and L2 Regularization

It should be noted in advance that in this document, many explanations of professional terms may be inadequate and may also be full of loopholes. Therefore, I hope you can understand. If possible, please help me correct them! In addition, this document will explain in great detail how to build a neural network for handwritten digit recognition from scratch, and it is quite long.

Finally, I sincerely hope that you can point out the logical errors in my writing and post your corrections in the **Issue**. Thank you.

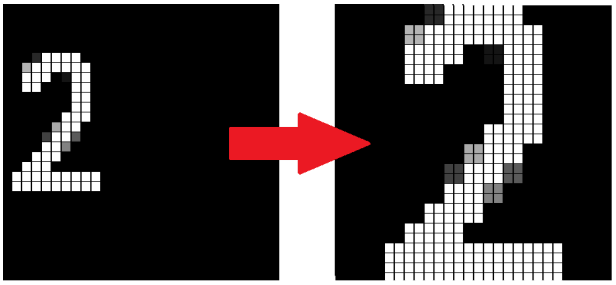
Image Processing

Firstly, whether it's the **training set** or the **validation set**, the handwritten images provided by **MNIST** are all $28 \times 28 = 784$ pixels. That means we need to use these 784 pixels as the input layer.

However, before these pixels are fed into the input layer, we first need to perform very simple preprocessing on the images.

> Why do we need to perform preprocessing? Because suppose I write a digit 2 that is very far to the left. If we directly predict it, the result will be very poor because the neural network is not very good at recognizing digits on the edge. Therefore, we preprocess the training set and also preprocess the digits on the drawing board so that they are concentrated in the middle. This will **significantly** improve the recognition accuracy!

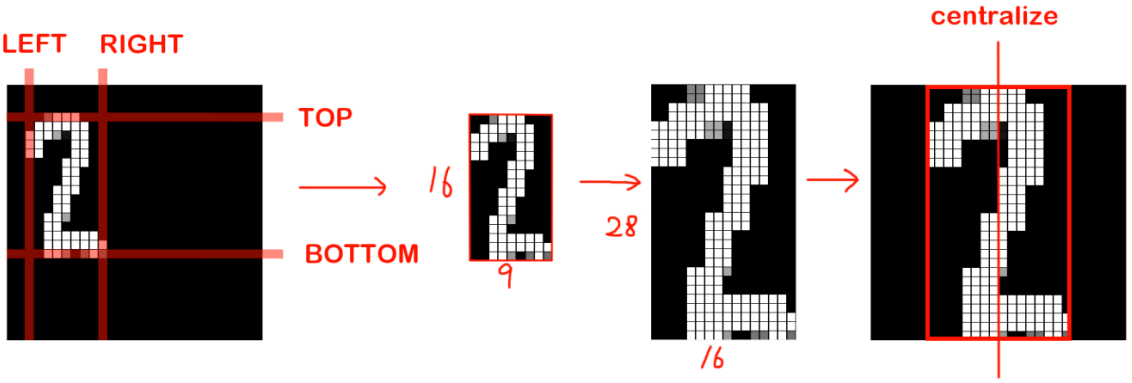
For example, I wrote the digit 2, but since it is off to the left, we not only need to move it to the middle but also enlarge it to $28 * 28$. Only in this way can we use a relatively unified standard for training and prediction.



For an input image, we first need to obtain its smallest rectangle, that is, from `left` to `right`, and from `top` to `bottom`. (As shown in the figure below)

Then we extract this image, whose width is `Math.abs(bottom - top) + 1` and length is `Math.abs(left - right) + 1`, and store it in array A.

```
int height = Math.abs(bottom - top) + 1;
int weight = Math.abs(left - right) + 1;
double[][] A = new double[height][weight]; // According to our example, the size
of array A should now be A[16][9].
```



Next, we enlarge the image in array A as much as possible. The longer side of array A is stretched to 28 (to match 28×28). In this example, the width of 16 is stretched to 28. Therefore,

$$k = \frac{28}{width_{old}}$$

$$width_{new} = width_{old} \times k = 16 \times \frac{28}{16} = 28$$

$$length_{new} = length_{origin} \times \frac{28}{weight_{origin}} = 9 \times \frac{28}{16} = 15.57 = (int)16$$

Here we regard the stretched image as

```
double[][] B
```

Then, we just need to center `B` to complete the initialization of the position.

Next, we need to normalize the data, that is, normalize all the values to the range of $[0.0, 1.0]$. This can enhance the representation.

In the entire 28×28 picture, assume that the minimum value of the **pixel** activation value (ranging from $[0, 1]$, where a white pixel is 1 and a black pixel is 0) is `min` and the maximum value is `max`. Meanwhile, make sure that the obtained minimum value `min` is not equal to 0.0 (black) of the canvas, but the minimum value greater than 0. Then for such a pixel value i_{old}, we normalize it as follows:

$$i_{new} = \frac{i_{old} - min}{max - min}$$

For example, now the minimum value in our image is 0.1 and the maximum value is 0.7. Then for any pixel i_{old}, its value after the update should be...

$$i_{new} = \frac{i_{old} - 0.1}{0.7 - 0.1}$$

And the code should be like

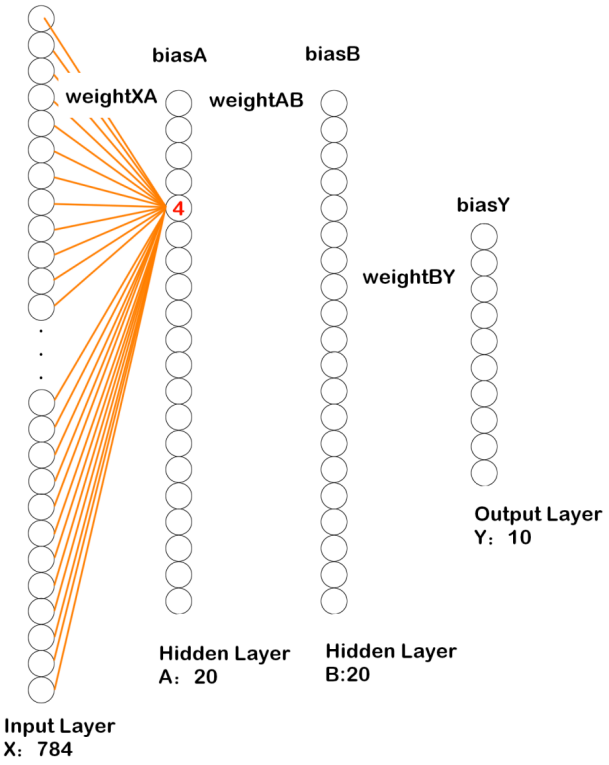
```
for (int x = 0; x < currentImg.length - 1; x++) {
                for (int y = 0; y < currentImg[0].length; y++) {
                    currentImg[x][y] = (currentImg[x][y] - min) / (max - min);
                }
            }
```

Then the initialization of an image is completed.

Forward Propagation

Next is how this neural network performs forward propagation.

The overall structure of the neural network is shown in the figure below.



Firstly, let's take a look at the composition of this neural network:

Input layer **X**: There are 784 neurons, and the subscript is agreed to be l.

Hidden layer 1 **A**: There are 20 neurons, and the subscript is agreed to be r.

Hidden layer 2 **B**: There are 20 neurons, and the subscript is agreed to be i.

Output layer **Y**: There are 10 neurons, and the subscript is agreed to be j. (That is, there are 10 outputs for the numbers from 1 to 9.)

These four layers are named as follows:

```
double[] X = new double[784]; // Input layer
double[] A = new double[20];  // Hidden layer 1
double[] B = new double[20];  // Hidden layer 2
double[] Y = new double[10];  // Output layer
```

Meanwhile, there are also settings for weights:

1. The weight from layer X to layer A is `weightXA[784][20]`, and the mathematical symbol is $W_{l,r}^{(1)}$.
2. The weight from layer A to layer B is `weightAB[20][20]`, and the mathematical symbol is $W_{r,i}^{(2)}$.
3. The weight from layer B to layer Y is `weightBY[20][10]`, and the mathematical symbol is $W_{i,j}^{(3)}$.

Settings for biases:

1. The bias for layer A is `biasA`, and the mathematical symbol is $Bias_r^{(1)}$.
2. The bias for layer B is `biasB`, and the mathematical symbol is $Bias_i^{(2)}$.
3. The bias for layer Y is `biasY`, and the mathematical symbol is $Bias_j^{(3)}$.

```
static double[][]   weightXA    = new double[784][20];  // 下标 weightXA[l][r]
static double[][]   weightAB    = new double[20][20];   // 下标 weightAB[r][i]
static double[][]   weightBY    = new double[20][10];   // 下标 weightBY[i][j]
static double[]     biasA       = new double[20];       // 下标 biasA[r]
static double[]     biasB       = new double[20];       // 下标 biasB[i]
static double[]     biasY       = new double[10];       // 下标 biasY[j]
```

It is worth noting that the values of weights and biases are random at the very beginning (specifically, it is the **He initialization with uniform distribution**, which will be explained at the end).

For the mathematical symbols of **weights**, we make the following conventions:

$$W_{l,r}^{(1)}$$

Here, (1) represents the weights of the first layer, that is, `weightXA`, and l, r represent the weight of the connection from point l on the left to point r on the right.

For the mathematical symbols of **biases**, we make the following conventions:

$$Bias_i^{(1)}$$

This represents the i-th value of the first layer's bias `biasA`.

Examples:

Symbol	Meaning	Symbol	Meaning
$W^{(1)}_{740,3}$	The weight from point 740 of layer X to point 3 of layer A, which is `weightXA[740][3]`	$Bias^{(1)}_{13}$	The bias of point 13 of layer A, which is `biasA[13]`
$W^{(2)}_{16,2}$	The weight from point 16 of layer A to point 2 of layer B, which is `weightAB[16][2]`	$Bias^{(2)}_{7}$	The bias of point 7 of layer B, which is `biasB[7]`
$W^{(3)}_{20,8}$	The weight from point 20 of layer B to point 8 of layer Y, which is `weightBY[20][8]`	$Bias^{(3)}_{4}$	The bias of point 4 of layer Y, which is `biasY[4]`

Then, how to calculate `A[4]` in the figure?

$$a_4 = (x_1 \quad x_2 \quad \cdots \quad x_{784}) \begin{pmatrix} W^{(1)}_{1,4} \\ W^{(1)}_{2,4} \\ \cdot \\ \cdot \\ \cdot \\ W^{(1)}_{783,4} \\ W^{(1)}_{784,4} \end{pmatrix} + bias^{(1)}_4$$

```
for(int left = 0; left < 784; left++)
{
    A[4] += X[left] * weightXA[left][4]; // weightXA[left][right] represents the weight of the connection from the left node 'left' to the right node 'right'.
}
A[4] += biasA[4]; // Add the bias.
```

After the calculation, we need to compress the range of a_4 within $[0, 1]$. So we will use the **sigmoid** function:

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

Therefore,

```
A[4] = sigmoid(A[4]); // Apply the sigmoid function.
```

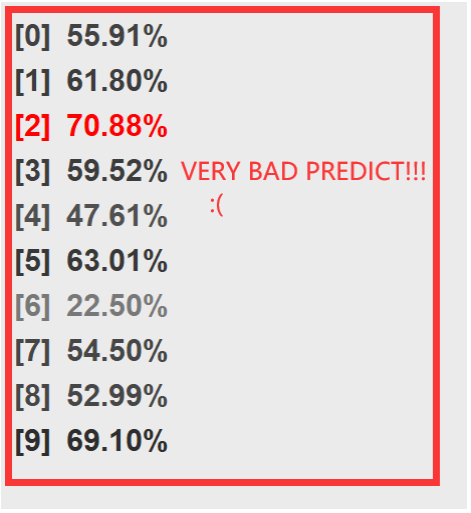
In this way, one neuron is calculated.

All in all, forward propagation is to continuously multiply matrices, then add biases and apply the sigmoid function.

Therefore, the code for **forward propagation** is...

```
double[] X = new double[784];
double[] A = new double[20];
double[] B = new double[20];
double[] Y = new double[10];
// Calculate hidden layer A
for (int i = 0; i < 20; i++) {
    for (int left = 0; left < 784; left++) {
        A[i] += X[left] * Processing.weightXA[left][i]; // Processing. is a
variable in this class.
    }
    // After adding the bias, apply the sigmoid function.
    A[i] += Processing.biasA[i];
    A[i] = sigmoid(A[i]);
}
// Calculate hidden layer B
for (int i = 0; i < 20; i++) {
    for (int left = 0; left < 20; left++) {
        B[i] += A[left] * Processing.weightAB[left][i];
    }
    B[i] += Processing.biasB[i];
    B[i] = sigmoid(B[i]);
}
// Calculate Y
for (int i = 0; i < 10; i++) {
    for (int left = 0; left < 20; left++) {
        Y[i] += B[left] * Processing.weightBY[left][i];
    }
    Y[i] += Processing.biasY[i];
    Y[i] = sigmoid(Y[i]);
}
```

After the propagation is completed, suppose we input the digit 3, then the result of the output layer is shown as follows.



Obviously, we input 3, but it predicts 2, and the probabilities of each digit seem to be similar. Apparently, this output is very poor because we haven't trained the neural network yet!

Then, the following backpropagation will talk about how to use the gradient descent method to train this network.

Backpropagation

Previously, we mentioned that forward propagation is roughly like this:

$$X \to \alpha \to A \to \beta \to B \to \gamma \to Y$$

Here, α, β, γ represent the values before applying the sigmoid function respectively. That is:

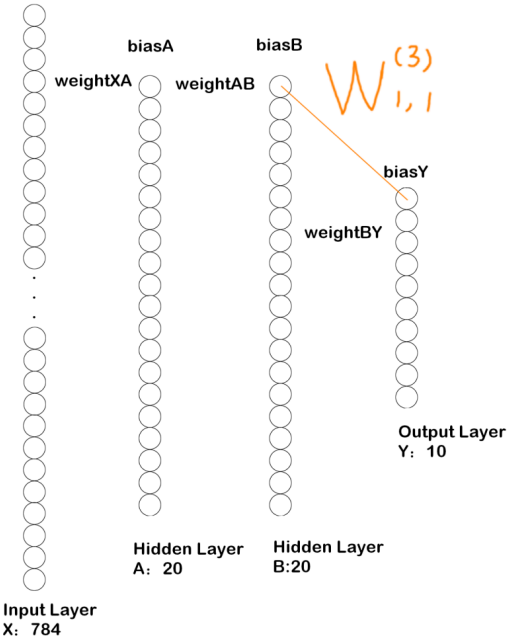
$$\alpha_4 = \begin{pmatrix} x_1 & x_2 & \cdots & x_{784} \end{pmatrix} \begin{pmatrix} W^{(1)}_{1,4} \\ W^{(1)}_{2,4} \\ \cdot \\ \cdot \\ \cdot \\ W^{(1)}_{783,4} \\ W^{(1)}_{784,4} \end{pmatrix} + bias^{(1)}_4$$

$$a_4 = sigmoid(\alpha_4)$$

We define it in this way to make it more convenient to apply the chain rule later.

Updating Weights and Biases

Updating $W^{(3)}_{i,j}$



According to the gradient descent formula,

$$W^{(3)\prime}_{i,j} = W^{(3)}_{i,j} - \epsilon \cdot \nabla G$$

$$\nabla G = \frac{\partial Loss}{\partial W^{(3)}_{i,j}}$$

$$Loss = \frac{1}{2}\sum_{j=1}^{10}(y_j - y_{gt(j)})^2$$

Here, ϵ is the learning rate (or step size), which is a constant. $Loss$ is the loss function, and y_{gt} in it represents the actual value.

For example, if we write a digit 6, then for the output layer, $y_{gt(6)} = Y[6] = 1.0$, while the others are 0.0.

y_j	$y_{gt(j)}$
[0] 63.85%	0.00
[1] 46.90%	0.00
[2] 76.71%	0.00
[3] 65.24%	0.00
[4] 70.42%	0.00
[5] 19.28%	0.00
[6] 40.07%	1.00
[7] 55.20%	0.00
[8] 69.67%	0.00
[9] 55.19%	0.00

Therefore,

$$W_{i,j}^{(3)\prime} = W_{i,j}^{(3)} - \epsilon \cdot \frac{\frac{1}{2}\sum_{j=1}^{10}(y_j - y_{gt(j)})^2}{\partial W_{i,j}^{(3)}}$$

Firstly we calculate the gradient.

$$\nabla G = \frac{\partial Loss}{\partial W_{i,j}^{(3)}} = \frac{\frac{1}{2}\sum_{j=1}^{10}(y_j - y_{gt(j)})^2}{\partial W_{i,j}^{(3)}}$$

However, we can't directly take the partial derivative of this expression. Therefore, we can consider using the chain rule.

Here, we also need to list some relationships so that we can calculate the partial derivative of the loss function with respect to $W_{1,1}^{(3)}$.

$$y_j = sigmoid(\gamma_j)$$

$$\gamma_j = (b_1 \quad b_2 \quad \cdots \quad b_{20}) \begin{pmatrix} W_{1,j}^{(3)} \\ W_{2,j}^{(3)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,j}^{(3)} \end{pmatrix} + Bias_j^{(3)}$$

Therefore,

$$\nabla G = \frac{\partial Loss}{\partial W_{i,j}^{(3)}}$$

$$= \frac{\partial Loss}{\partial y_j} \frac{\partial y_j}{\partial \gamma_j} \frac{\partial \gamma_j}{\partial W_{i,j}^{(3)}}$$

$$= \sum_{j=1}^{10} (y_j - y_{gt(j)}) \cdot sigmoid(\gamma_j)(1 - sigmoid(\gamma_j)) \cdot b_i$$

$$= \sum_{j=1}^{10} (y_j - y_{gt(j)}) \cdot y_i(1 - y_i) \cdot b_i$$

> Note: For the function $sigmoid(x)$, its derivative is $sigmoid(x) \times (1 - sigmoid(x))$.

Therefore, the updated weight W should be:

$$W_{i,j}^{(3)\prime} = W_{i,j}^{(3)} - \epsilon \cdot \sum_{j=1}^{10} (y_j - y_{gt(j)}) \cdot y_i(1 - y_i) \cdot b_i$$

Updating $Bias_j^{(3)}$

In fact, for **biasY**, its update is quite similar to the above.

$$y_j = sigmoid(\gamma_j)$$

$$\gamma_j = \begin{pmatrix} b_1 & b_2 & \cdots & b_{20} \end{pmatrix} \begin{pmatrix} W_{1,j}^{(3)} \\ W_{2,j}^{(3)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,j}^{(3)} \end{pmatrix} + Bias_j^{(3)}$$

The last term of the partial derivative here is 1.

$$\nabla G = \frac{\partial Loss}{\partial Bias_j^{(3)}}$$

$$= \frac{\partial Loss}{\partial y_j} \frac{\partial y_j}{\partial \gamma_j} \frac{\partial \gamma_j}{\partial Bias_j^{(3)}}$$

$$= \sum_{j=1}^{10} (y_j - y_{gt(j)}) \cdot sigmoid(\gamma_j)(1 - sigmoid(\gamma_j)) \cdot 1$$

$$= \sum_{j=1}^{10} (y_j - y_{gt(j)}) \cdot y_j(1 - y_j)$$

Therefore,

$$Bias_j^{(3)\prime} = Bias_j^{(3)} - \epsilon \cdot \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_i(1-y_i)$$

Updating $W_{r,i}^{(2)}$

To update $W_{r,i}^{(2)}$, we need to continue writing from γ_j.

$$y_j = sigmoid(\gamma_j)$$

$$\gamma_j = \begin{pmatrix} b_1 & \cdots & bi & \cdots & b_{20} \end{pmatrix} \begin{pmatrix} W_{1,j}^{(3)} \\ W_{2,j}^{(3)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,j}^{(3)} \end{pmatrix} + Bias_j^{(3)}$$

$$b_i = sigmoid(\beta_i)$$

$$\beta_i = \begin{pmatrix} a_1 & a_2 & \cdots & a_{20} \end{pmatrix} \begin{pmatrix} W_{1,i}^{(2)} \\ W_{2,i}^{(2)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,i}^{(2)} \end{pmatrix} + Bias_i^{(2)}$$

Then, the gradient

$$\nabla G = \frac{\partial Loss}{\partial W_{r,i}^{(2)}}$$

$$= \frac{\partial Loss}{\partial y_j} \frac{\partial y_j}{\partial \gamma_j} \frac{\partial \gamma_j}{\partial b_i} \frac{\partial b_i}{\partial \beta_i} \frac{\partial \beta_i}{\partial W_{r,i}^{(2)}}$$

$$= \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r$$

Therefore, after the update, $W_{r,i}^{(2)\prime}$

$$W_{r,i}^{(2)\prime} = W_{r,i}^{(2)} - \epsilon \cdot \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r$$

Updating $Bias_i^{(2)}$

Refer to the steps above.

$$\nabla G = \frac{\partial Loss}{\partial Bias_i^{(2)}}$$

$$= \frac{\partial Loss}{\partial y_j}\frac{\partial y_j}{\partial \gamma_j}\frac{\partial \gamma_j}{\partial b_i}\frac{\partial b_i}{\partial \beta_i}\frac{\partial \beta_i}{\partial Bias_i^{(2)}}$$

$$= \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot 1$$

Therefore, the updated $Bias_i^{(2)\prime}$ should be...

$$Bias_i^{(2)\prime} = Bias_i^{(2)} - \epsilon \cdot \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r$$

Updating $W_{l,r}^{(1)}$

Continue to list the expressions.

$$y_j = sigmoid(\gamma_j)$$

$$\gamma_j = \begin{pmatrix} b_1 & \cdots & bi & \cdots & b_{20} \end{pmatrix} \begin{pmatrix} W_{1,j}^{(3)} \\ W_{2,j}^{(3)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,j}^{(3)} \end{pmatrix} + Bias_j^{(3)}$$

$$b_i = sigmoid(\beta_i)$$

$$\beta_i = \begin{pmatrix} a_1 & \cdots & a_r & \cdots & a_{20} \end{pmatrix} \begin{pmatrix} W_{1,i}^{(2)} \\ W_{2,i}^{(2)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,i}^{(2)} \end{pmatrix} + Bias_i^{(2)}$$

$$a_r = sigmoid(\alpha_r)$$

$$\alpha_r = \begin{pmatrix} x_1 & \cdots & x_{20} \end{pmatrix} \begin{pmatrix} W_{1,r}^{(1)} \\ W_{2,r}^{(1)} \\ \cdot \\ \cdot \\ \cdot \\ W_{784,r}^{(1)} \end{pmatrix} + Bias_r^{(1)}$$

The gradient

$$\nabla G = \frac{\partial Loss}{\partial W_{l,r}^{(1)}}$$

$$= \frac{\partial Loss}{\partial y_j}\frac{\partial y_j}{\partial \gamma_j}\frac{\partial \gamma_j}{\partial b_i}\frac{\partial b_i}{\partial \beta_i}\frac{\partial \beta_i}{\partial a_r}\frac{\partial a_r}{\partial \alpha_r}\frac{\partial \alpha_r}{\partial W_{l,r}^{(1)}}$$

$$= \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot x_l$$

Therefore,

$$W_{l,r}^{(1)\,\prime} = W_{l,r}^{(1)} - \epsilon \cdot \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot x_l$$

Updating $Bias_r^{(1)}$

Similarly, list the expressions.

$$y_j = sigmoid(\gamma_j)$$

$$\gamma_j = \begin{pmatrix} b_1 & \cdots & bi & \cdots & b_{20} \end{pmatrix} \begin{pmatrix} W_{1,j}^{(3)} \\ W_{2,j}^{(3)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,j}^{(3)} \end{pmatrix} + Bias_j^{(3)}$$

$$b_i = sigmoid(\beta_i)$$

$$\beta_i = \begin{pmatrix} a_1 & \cdots & a_r & \cdots & a_{20} \end{pmatrix} \begin{pmatrix} W_{1,i}^{(2)} \\ W_{2,i}^{(2)} \\ \cdot \\ \cdot \\ \cdot \\ W_{20,i}^{(2)} \end{pmatrix} + Bias_i^{(2)}$$

$$a_r = sigmoid(\alpha_r)$$

$$\alpha_r = \begin{pmatrix} x_1 & \cdots & x_{20} \end{pmatrix} \begin{pmatrix} W_{1,r}^{(1)} \\ W_{2,r}^{(1)} \\ \cdot \\ \cdot \\ \cdot \\ W_{784,r}^{(1)} \end{pmatrix} + Bias_r^{(1)}$$

Then, the gradient

$$\nabla G = \frac{\partial Loss}{\partial Bias_r^{(1)}}$$

$$= \frac{\partial Loss}{\partial y_j}\frac{\partial y_j}{\partial \gamma_j}\frac{\partial \gamma_j}{\partial b_i}\frac{\partial b_i}{\partial \beta_i}\frac{\partial \beta_i}{\partial a_r}\frac{\partial a_r}{\partial \alpha_r}\frac{\partial \alpha_r}{\partial Bias_r^{(1)}}$$

$$= \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot 1$$

After the update, $Bias_r^{(1)\prime}$ should be...

$$Bias_r^{(1)\prime} = Bias_r^{(1)} - \epsilon \cdot \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r)$$

Therefore, the overall result should be:

weight/bias	gradient
$W_{i,j}^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_i(1-y_i) \cdot b_i$
$Bias_j^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot 1$
$W_{r,i}^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r$
$Bias_i^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot 1$
$W_{l,r}^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot x_l$
$Bias_r^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot 1$

> We don't need to consider too much about the coefficients, because it's just a matter of multiples. The Adam optimizer will adjust the magnitudes by itself later.

Adding Regularization L1 and L2 to the Weights

To prevent the problem of overfitting, we will add L1 and L2 regularization terms on the basis of the loss function. Among them, we define

$$\lambda_1 = 0.001$$
$$\lambda_2 = 0.001$$

Then the loss function becomes

$$Loss = \frac{1}{2}\sum_{j=1}^{10}(y_j - y_{gt(j)})^2 + \lambda_1 \cdot \sum|W| + \lambda_2 \cdot \sum W^2$$

Meanwhile, the loss function is divided into these three parts: mean squared error loss + L1 loss + L2 loss

$$Loss = L_{MSE} + L_{L1} + L_{L2}$$
$$L_{MSE} = \frac{1}{2}\sum_{j=1}^{10}(y_j - y_{gt(j)})^2$$
$$L_{L1} = \lambda_1 \cdot \sum|W|$$
$$L_{L2} = \lambda_2 \cdot \sum W^2$$

However, it is quite obvious that L1 and L2 regularization only have an impact on the gradients of the weights and have no impact on the gradients of the biases.

Therefore, let's take $W_{i,j}^{(3)}$ as an example to see what the gradients will be like after adding the regularization.

$$\begin{aligned}\nabla G &= \frac{\partial Loss}{\partial W_{i,j}^{(3)}} \\ &= \frac{\partial L_{MSE}}{\partial W_{i,j}^{(3)}} + \frac{\partial L_{L1}}{\partial W_{i,j}^{(3)}} + \frac{\partial L_{L2}}{\partial W_{i,j}^{(3)}} \\ &= \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_i(1-y_i) \cdot b_i + \lambda_1 \cdot sign(W_{i,j}^{(3)}) + 2\lambda_2 \cdot W_{i,j}^{(3)}\end{aligned}$$

Here,

$$sign(x) = \begin{cases} -1 & , \text{if } x < 0 \\ 0 & , \text{if } x = 0 \\ 1 & , \text{if } x > 0 \end{cases}$$

Therefore, the gradients after adding L1 and L2 regularization should be...

weight/bias	gradient
$W_{i,j}^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_i(1-y_i) \cdot b_i + \lambda_1 \cdot sign(W_{i,j}^{(3)}) + 2\lambda_2 \cdot W_{i,j}^{(3)}$
$Bias_j^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j)$
$W_{r,i}^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r + \lambda_1 \cdot sign(W_{r,i}^{(2)}) + 2\lambda_2 \cdot W_{r,i}^{(2)}$
$Bias_i^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i)$
$W_{l,r}^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot x_l + \lambda_1 \cdot sign(W_{l,r}^{(1)}) + 2\lambda_2 \cdot W_{l,r}^{(1)}$
$Bias_r^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r)$

Adam Optimizer

In order to solve the problems of oscillation and missing the minimum value due to an overly large learning rate, the Adam optimizer provides an adjustment scheme.

In the past, the gradient usually used a fixed learning rate, for example, $\epsilon = 0.001$. However, when the gradient is reduced, the learning rate should also be reduced accordingly. Meanwhile, the oscillation problem needs to be solved.

$$W_{i,j}^{(3)\prime} = W_{i,j}^{(3)} - \epsilon \cdot \nabla G$$

Therefore, next I will introduce the steps of the Adam optimizer:

1. Calculate the gradient ∇G.
2. $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla G_t$
3. $v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot \nabla G_t^2$
4. $\hat{m}_t = \dfrac{m_t}{1-\beta_1^t}$
5. $\hat{v}_t = \dfrac{v_t}{1-\beta_2^t}$
6. Update $W_{i,j}^{(3)\prime} = W_{i,j}^{(3)} - \dfrac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

Explanation: Here, there are four constants:

Constant term	$\beta_1 = 0.9$
Constant term	$\beta_2 = 0.999$
A very small value added to prevent the situation where $\sqrt{\hat{v}_t}$ is 0	$\epsilon = 10^{-8}$
Learning rate	$\alpha = 0.001$

In addition:

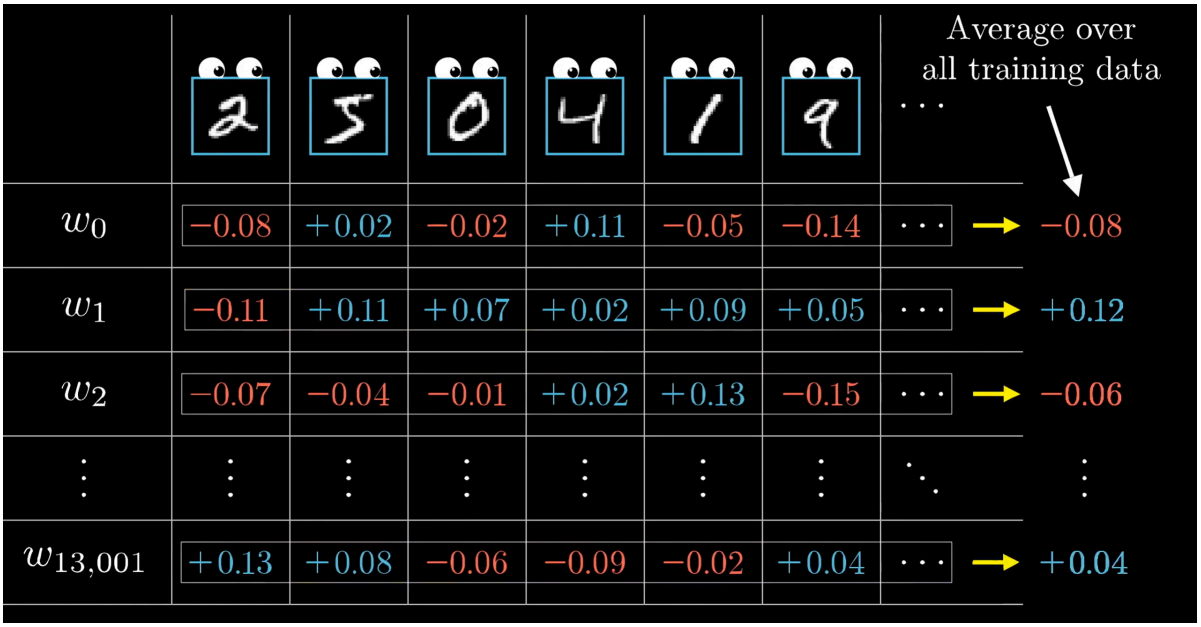
1. The subscript t represents the t-th iteration.
2. The initial values of m_0 and v_0 are both 0.

Stochastic Gradient Descent

Previously, we got to know the gradient.

weight/bias	gradient
$W_{i,j}^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_i(1-y_i) \cdot b_i + \lambda_1 \cdot sign(W_{i,j}^{(3)}) + 2\lambda_2 \cdot W_{i,j}^{(3)}$
$Bias_j^{(3)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j)$
$W_{r,i}^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot a_r + \lambda_1 \cdot sign(W_{r,i}^{(2)}) + 2\lambda_2 \cdot W_{r,i}^{(2)}$
$Bias_i^{(2)}$	$\nabla G = \sum_{j=1}^{10}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i)$
$W_{l,r}^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r) \cdot x_l + \lambda_1 \cdot sign(W_{l,r}^{(1)}) + 2\lambda_2 \cdot W_{l,r}^{(1)}$
$Bias_r^{(1)}$	$\nabla G = \sum_{j=1}^{10}\sum_{i=1}^{20}(y_j - y_{gt(j)}) \cdot y_j(1-y_j) \cdot W_{i,j}^{(3)} \cdot b_i(1-b_i) \cdot W_{r,i}^{(2)} \cdot a_r(1-a_r)$

However, we have to think about a problem. Suppose we want to train 60,000 images. If we still use the original method and conduct gradient descent, then we need to go through all 60,000 images, add up the values of these weights and take the average to obtain a total gradient. Only at this time does the descent happen once. As shown in the following figure.



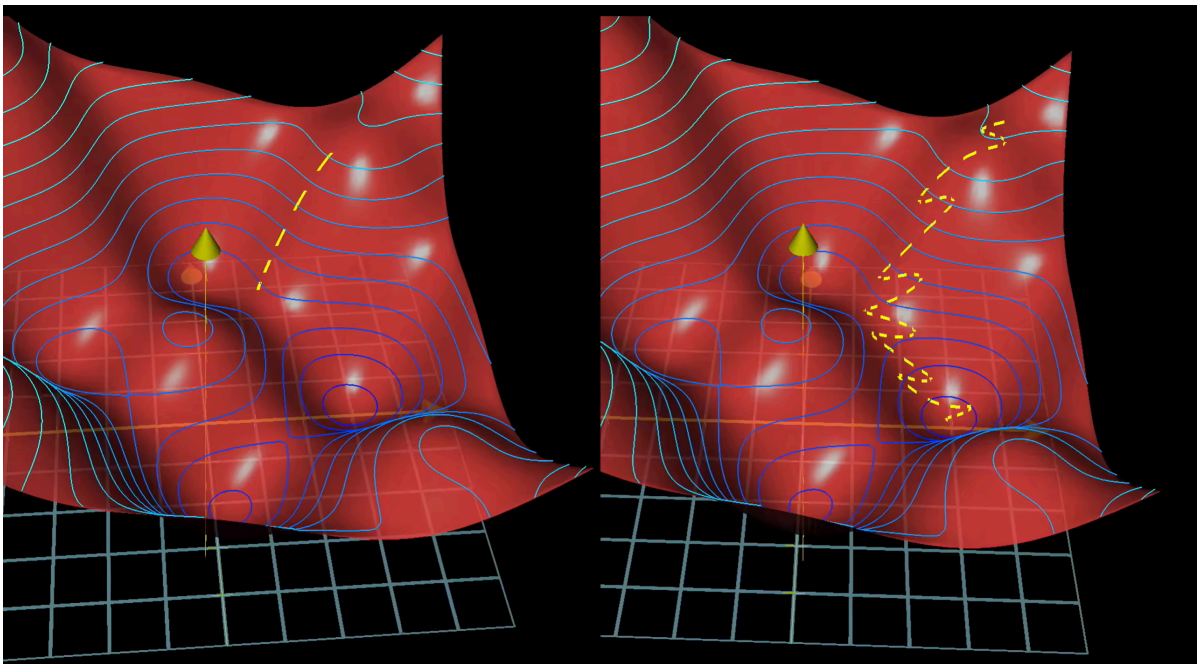
> Image source: [3B1B] Backpropagation, step-by-step | DL3 - YouTube

However, the drawback of this is that it is too slow! We have gone through 60,000 images and only descended once! Although this descent is very accurate, it is very slow, and since all 60,000 images need to be loaded into the memory, it is very unfriendly to the memory.

Therefore, we can introduce the concept of batch. For example, randomly select 64 images from the 60,000 images as a batch. In this way, after we find the mean value of the weight changes of these 64 images, we can perform a descent on the network! And the memory will also be reduced accordingly.

Of course, at this time, one round of Epoch is equivalent to performing about 938 descents of batches with a size of 64. At the same time, one round of Epoch is about 938 times faster than the original.

The left figure shows the standard gradient descent, which is slow but very accurate; the right figure shows the stochastic gradient descent, that is, the method of introducing batches. Although the descent is not very accurate, it is very fast and can also reach the lowest point.



> Image source: [3B1B] Backpropagation, step-by-step | DL3 - YouTube

However, in fact, there is no necessity to set the batch in this program, because in the actual test, there is no difference in terms of time, memory and results.

If a single data set is not 28×28 but larger, then the concept of introducing batches can be considered.

After introducing the batch, assuming the batch = 64, then the gradient should be changed to the average of the gradients of all images in this batch.

weight/bias	gradient
$W_{i,j}^{(3)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_i^{(n)}(1-y_i^{(n)}) \cdot b_i^{(n)} + \lambda_1 \cdot sign(W_{i,j}^{(3)}) + 2\lambda_2 \cdot W_{i,j}^{(3)}$
$Bias_j^{(3)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)})$
$W_{r,i}^{(2)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot a_r^{(n)} + \lambda_1 \cdot sign(W_{r,i}^{(2)}) + 2\lambda_2 \cdot W_{r,i}^{(2)}$
$Bias_i^{(2)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)})$
$W_{l,r}^{(1)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}\sum_{i=1}^{20}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1-a_r^{(n)}) \cdot x_l^{(n)} + \lambda_1 \cdot sign(W_{l,r}^{(1)}) + 2\lambda_2 \cdot W_{l,r}^{(1)}$
$Bias_r^{(1)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}\sum_{i=1}^{20}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1-a_r^{(n)})$

Here, the superscript (n) of x, a, b, y represents the n-th image in the batch.

For example, $x_{740}^{(52)}$ represents the 740th value of the input layer **X** of the 52nd image.

Code Implementation

First, it should be noted that when updating the last layer of weights, that is, $W_{l,r}^{(1)}$, it will involve five layers of loops, which will greatly reduce the code running efficiency. However, the gradients of the weights and biases are repeated in many places.

For example, when considering this problem, we first remove the influence of regularization on the gradient. It will be found that for all biases with respect to the weights, only the last term is replaced with 1. Therefore, we can calculate the biases first and then the weights.

Before starting, we can consider pre-storing the four layers of **X**, **A**, **B**, **Y** from 1 to 64 into a `double[][]`.

Input an `ArrayList<double[][]> batch` represents that there are `batch.size()` 29 x 28 pixel pictures in this batch.

Among them, `batch.get(n)[28][0]` is the label of the n-th image.

The value of `XList[52][740]` is the value of $x_{740}^{(52)}$.

```
private static void trainBatch(ArrayList<double[][]> batch) {
        double[][] XList = new double[batch.size()][784];
        double[][] AList = new double[batch.size()][20];
        double[][] BList = new double[batch.size()][20];
        double[][] YList = new double[batch.size()][10];

        for (int numberOfBatch = 0; numberOfBatch < batch.size();
numberOfBatch++) {
            double[] X = new double[784];
            double[] A = new double[20];
            double[] B = new double[20];
            double[] Y = new double[10];
            double[][] currentImg = batch.get(numberOfBatch);
            // Convert the two-dimensional array image to a one-dimensional array
            int counter = 0;
            for (int x = 0; x < 28; x++) {
                for (int y = 0; y < 28; y++) {
                    X[counter++] = currentImg[x][y];
                }
            }
            // Calculate the hidden layer A
            for (int i = 0; i < 20; i++) {
                for (int left = 0; left < 784; left++) {
                    A[i] += X[left] * Processing.weightXA[left][i];
                }
                // Add the bias and then push it into the sigmoid function
                A[i] += Processing.biasA[i];
                A[i] = sigmoid(A[i]);
            }
            // Calculate the hidden layer B
            for (int i = 0; i < 20; i++) {
                for (int left = 0; left < 20; left++) {
                    B[i] += A[left] * Processing.weightAB[left][i];
                }
                B[i] += Processing.biasB[i];
                B[i] = sigmoid(B[i]);
            }
            // Calculate Y
            for (int i = 0; i < 10; i++) {
                for (int left = 0; left < 20; left++) {
                    Y[i] += B[left] * Processing.weightBY[left][i];
                }
```

```
                Y[i] += Processing.biasY[i];
                Y[i] = sigmoid(Y[i]);
            }
            XList[numberOfBatch] = arrayCopy(X);
            AList[numberOfBatch] = arrayCopy(A);
            BList[numberOfBatch] = arrayCopy(B);
            YList[numberOfBatch] = arrayCopy(Y);
        }

        // There is still subsequent code here......
}
```

Calculate the Gradient

First, we calculate the gradient. At the beginning of the `trainBatch` method, we also define...

```
double[][] gradient_weightXA = new double[784][20];
double[][] gradient_weightAB = new double[20][20];
double[][] gradient_weightBY = new double[20][10];
double[] gradient_biasA = new double[20];
double[] gradient_biasB = new double[20];
double[] gradient_biasY = new double[10];
```

At this time, observe the table:

weight/bias	gradient
$W_{i,j}^{(3)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_i^{(n)}(1-y_i^{(n)}) \cdot b_i^{(n)} + \lambda_1 \cdot sign(W_{i,j}^{(3)}) + 2\lambda_2 \cdot W_{i,j}^{(3)}$
$Bias_j^{(3)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)})$
$W_{r,i}^{(2)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot a_r^{(n)} + \lambda_1 \cdot sign(W_{r,i}^{(2)}) + 2\lambda_2 \cdot W_{r,i}^{(2)}$
$Bias_i^{(2)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)})$
$W_{l,r}^{(1)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}\sum_{i=1}^{20}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1-a_r^{(n)}) \cdot x_l^{(n)} + \lambda_1 \cdot sign(W_{l,r}^{(1)}) + 2\lambda_2 \cdot W_{l,r}^{(1)}$
$Bias_r^{(1)}$	$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}\sum_{i=1}^{20}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1-b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1-a_r^{(n)})$

It is found that $\sum_{n=1}^{64}$ can be taken out and then written into a loop framework.

```
for (int n = 0; n < batch.size(); n++) {
     // Descend biasY
     // Descend weightBY
     // Descend biasB
     // Descend weightAB
     // Descend biasA
     // Descend weightXA
}
```

First, we extract the correct label of the n-th image.

```
double label = batch.get(n)[28][0];
```

Step 1, Calculate gradient_biasY

$$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1-y_j^{(n)})$$

According to the formula:

```
// Descend biasY
for (int j = 0; j < 10; j++) {
    double ygt; // Actual value
    if (label == j) {
        ygt = 1.0;
    } else {
        ygt = 0.0;
    }
    double value = (YList[n][j] - ygt) * YList[n][j] * (1.0 - YList[n][j]);
    gradient_biasY[j] += value;
    storage1.add(value);
}
```

Here, we store the result of each calculation in `ArrayList<Double> storage1 = new ArrayList<>()`.

This can facilitate the subsequent calculations!

Step 2, Calculate gradient_weightBY

$$\nabla G = \sum_{n=1}^{64} \sum_{j=1}^{10} (y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_i^{(n)} (1 - y_i^{(n)}) \cdot b_i^{(n)}$$

```
// weightBY
for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 10; j++) {
        double value = storage1.get(j) * BList[n][i]; // Because the repeated
part can be directly called!
        gradient_weightBY[i][j] += value;
    }
}
```

Step 3, Calculate gradient_biasB

$$\nabla G = \sum_{n=1}^{64} \sum_{j=1}^{10} (y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)} (1 - y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)} (1 - b_i^{(n)})$$

It can be seen that, compared with **weightBY**, **biasB** only has two more terms. Therefore, we can add a method to store it in `ArrayList<Double> storage2 = new ArrayList<>()` in the code of **weightBY**.

```
// weightBY (Updated code)
for (int i = 0; i < 20; i++) {
    double temp = 0; // Add temp
    for (int j = 0; j < 10; j++) {
        double value = storage1.get(j) * BList[n][i];
        gradient_weightBY[i][j] += value;

        temp += storage1.get(j) * Processing.weightBY[i][j]* BList[n][i]*(1.0 -
 BList[n][i]); // Accumulate
    }
    storage2.add(temp);
}
```

Therefore, **biasB** can be directly written as:

```
// biasB
for (int i = 0; i < 20; i++) {
    gradient_biasB[i] += storage2.get(i);
}
```

Step 4, Calculate gradient_weightAB

$$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1 - y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1 - b_i^{(n)}) \cdot a_r^{(n)}$$

Obviously, **weightAB** only has one more term $a_r^{(n)}$ than **biasB**.

Therefore, we can directly call `storage2`!

```
// weightAB
for (int r = 0; r < 20; r++) {
    for (int i = 0; i < 20; i++) {
        gradient_weightAB[r][i] += storage2.get(i) * AList[n][r];
    }
}
```

Step 5, Calculate gradient_biasA

$$\nabla G = \sum_{n=1}^{64}\sum_{j=1}^{10}\sum_{i=1}^{20}(y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1 - y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1 - b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1 - a_r^{(n)})$$

For **biasA**, there are two more terms. Therefore, we can make some modifications to the code of weightAB and let the accumulated result of weightAB in each iteration be stored in `storage3`.

```
// weightAB
for (int r = 0; r < 20; r++) {
    double temp = 0;
    for (int i = 0; i < 20; i++) {
        gradient_weightAB[r][i] += storage2.get(i) * AList[n][r];
        temp += storage2.get(i) * AList[n][r] * weightAB[r][i] * (1.0 - AList[n]
[r]);
    }
    storage3.add(temp);
}
```

Therefore, for **biasA**, we can write it like this:

```
// biasA
for (int r = 0; r < 20; r++) {
    gradient_biasA[r] += storage3.get(r);
}
```

Step 6, Calculate gradient_weightXA

$$\nabla G = \sum_{n=1}^{64} \sum_{j=1}^{10} \sum_{i=1}^{20} (y_j^{(n)} - y_{gt(j)}^{(n)}) \cdot y_j^{(n)}(1 - y_j^{(n)}) \cdot W_{i,j}^{(3)} \cdot b_i^{(n)}(1 - b_i^{(n)}) \cdot W_{r,i}^{(2)} \cdot a_r^{(n)}(1 - a_r^{(n)}) \cdot x_l^{(n)}$$

Following this logic, we can write:

```
// weightXA
for (int l = 0; l < 784; l++) {
    for (int r = 0; r < 20; r++) {
        gradient_weightXA[l][r] += storage3.get(r) * XList[n][l];
    }
}
```

Adding Regularization Terms

Next, add the regularization terms! It is only necessary to add them for the three weights.

```
// Gradient of weightBY
for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 10; j++) {
        gradient_weightBY[i][j] += 2 * LAMBDA_L2 * Processing.weightBY[i][j] +
LAMBDA_L1 * sign(Processing.weightBY[i][j]);
    }
}
// Gradient of weightAB
for (int right = 0; right < 20; right++) {
    for (int i = 0; i < 20; i++) {
        gradient_weightAB[right][i] += 2 * LAMBDA_L2 * Processing.weightAB[right]
[i] + LAMBDA_L1 * sign(Processing.weightAB[right][i]);
    }
}
// Gradient of weightXA
for (int left = 0; left < 784; left++) {
    for (int right = 0; right < 20; right++) {
        gradient_weightXA[left][right] += 2 * LAMBDA_L2 *
Processing.weightXA[left][right] + LAMBDA_L1 * sign(Processing.weightXA[left]
[right]);
    }
}
```

Performing Gradient Descent with the Adam Optimizer

Next, we will use the Adam optimizer to perform gradient descent.

The steps of the Adam optimizer are as follows:

1. Calculate the gradient ∇G.
2. $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla G_t$
3. $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \nabla G_t^2$
4. $\hat{m}_t = \dfrac{m_t}{1 - \beta_1^t}$
5. $\hat{v}_t = \dfrac{v_t}{1 - \beta_2^t}$

6. Update $x' = x - \dfrac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

First, define the variables in the global scope as follows:

```
static double BETA1 = 0.9;
static double BETA2 = 0.999;
static double ALPHA = 0.001; // Learning rate
static double EPSILON = 1.0E-08; // To prevent the denominator from being zero
static double LAMBDA_L1 = 0.001; // Parameter for L1 regularization
static double LAMBDA_L2 = 0.001; // Parameter for L2 regularization
// Adam optimizer data for weightXA
static double[][] weightXA_Mt = new double[784][20];
static double[][] weightXA_Vt = new double[784][20];
static int weightXA_Iteration = 0;
// Adam optimizer data for weightAB
static double[][] weightAB_Mt = new double[20][20];
static double[][] weightAB_Vt = new double[20][20];
static int weightAB_Iteration = 0;
// Adam optimizer data for weightBY
static double[][] weightBY_Mt = new double[20][10];
static double[][] weightBY_Vt = new double[20][10];
static int weightBY_Iteration = 0;
// Adam optimizer data for biasA
static double[] biasA_Mt = new double[20];
static double[] biasA_Vt = new double[20];
static int biasA_Iteration = 0;
// Adam optimizer data for biasB
static double[] biasB_Mt = new double[20];
static double[] biasB_Vt = new double[20];
static int biasB_Iteration = 0;
// Adam optimizer data for biasY
static double[] biasY_Mt = new double[20];
static double[] biasY_Vt = new double[20];
static int biasY_Iteration = 0;
```

In the method, we need to increment the default iteration number by 1.

```
++weightXA_Iteration;
++weightAB_Iteration;
++weightBY_Iteration;
++biasA_Iteration;
++biasB_Iteration;
++biasY_Iteration;
```

Therefore, the code should be as follows:

1. **Update biasY**

```
// biasY
for (int j = 0; j < 10; j++) {
    biasY_Mt[j] = BETA1 * biasY_Mt[j] + (1.0 - BETA1) * gradient_biasY[j];
    biasY_Vt[j] = BETA2 * biasY_Vt[j] + (1.0 - BETA2) * gradient_biasY[j] *
gradient_biasY[j];
    double MtHat = biasY_Mt[j] / (1.0 - Math.pow(BETA1, biasY_Iteration));
    double VtHat = biasY_Vt[j] / (1.0 - Math.pow(BETA2, biasY_Iteration));
    biasY[j] = biasY[j] - (ALPHA * MtHat) / (Math.sqrt(VtHat) + EPSILON);
}
```

2. **Update weight_BY**

```
// weight_BY
for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 10; j++) {
        weightBY_Mt[i][j] = BETA1 * weightBY_Mt[i][j] + (1.0 - BETA1) *
gradient_weightBY[i][j];
        weightBY_Vt[i][j] = BETA2 * weightBY_Vt[i][j] + (1.0 - BETA2) *
gradient_weightBY[i][j] * gradient_weightBY[i][j];
        double MtHat = weightBY_Mt[i][j] / (1.0 - Math.pow(BETA1,
weightBY_Iteration));
        double VtHat = weightBY_Vt[i][j] / (1.0 - Math.pow(BETA2,
weightBY_Iteration));
        weightBY[i][j] = weightBY[i][j] - (ALPHA * MtHat) / (Math.sqrt(VtHat) +
EPSILON);
    }
}
```

3. **Update biasB**

```
// biasB
for (int i = 0; i < 20; i++) {
    biasB_Mt[i] = BETA1 * biasB_Mt[i] + (1.0 - BETA1) * gradient_biasB[i];
    biasB_Vt[i] = BETA2 * biasB_Vt[i] + (1.0 - BETA2) * gradient_biasB[i] *
gradient_biasB[i];
    double MtHat = biasB_Mt[i] / (1.0 - Math.pow(BETA1, biasB_Iteration));
    double VtHat = biasB_Vt[i] / (1.0 - Math.pow(BETA2, biasB_Iteration));
    biasB[i] = biasB[i] - (ALPHA * MtHat) / (Math.sqrt(VtHat) + EPSILON);
}
```

4. **Update weightAB**

```
// weightAB
for (int right = 0; right < 20; right++) {
    for (int i = 0; i < 20; i++) {
        weightAB_Mt[right][i] = BETA1 * weightAB_Mt[right][i] + (1.0 - BETA1) *
gradient_weightAB[right][i];
        weightAB_Vt[right][i] = BETA2 * weightAB_Vt[right][i] + (1.0 - BETA2) *
gradient_weightAB[right][i] * gradient_weightAB[right][i];
        double MtHat = weightAB_Mt[right][i] / (1.0 - Math.pow(BETA1,
weightAB_Iteration));
        double VtHat = weightAB_Vt[right][i] / (1.0 - Math.pow(BETA2,
weightAB_Iteration));
        weightAB[right][i] = weightAB[right][i] - (ALPHA * MtHat) /
(Math.sqrt(VtHat) + EPSILON);
    }
}
```

5. **Update biasA**

```
// biasA
for (int right = 0; right < 20; right++) {
    biasA_Mt[right] = BETA1 * biasA_Mt[right] + (1.0 - BETA1) *
gradient_biasA[right];
    biasA_Vt[right] = BETA2 * biasA_Vt[right] + (1.0 - BETA2) *
gradient_biasA[right] * gradient_biasA[right];
    double MtHat = biasA_Mt[right] / (1.0 - Math.pow(BETA1, biasA_Iteration));
    double VtHat = biasA_Vt[right] / (1.0 - Math.pow(BETA2, biasA_Iteration));
    biasA[right] = biasA[right] - (ALPHA * MtHat) / (Math.sqrt(VtHat) + EPSILON);
}
```

6. **Update weightXA**

```
// weightXA
for (int left = 0; left < 784; left++) {
    for (int right = 0; right < 20; right++) {
        weightXA_Mt[left][right] = BETA1 * weightXA_Mt[left][right] + (1.0 -
BETA1) * gradient_weightXA[left][right];
        weightXA_Vt[left][right] = BETA2 * weightXA_Vt[left][right] + (1.0 -
BETA2) * gradient_weightXA[left][right] * gradient_weightXA[left][right];
        double MtHat = weightXA_Mt[left][right] / (1.0 - Math.pow(BETA1,
weightXA_Iteration));
        double VtHat = weightXA_Vt[left][right] / (1.0 - Math.pow(BETA2,
weightXA_Iteration));
        weightXA[left][right] = weightXA[left][right] - (ALPHA * MtHat) /
(Math.sqrt(VtHat) + EPSILON);
    }
}
```

Up to this point, the backpropagation is completed.

Therefore, suppose we select `GUI.BATCH_SET_SIZE` pieces of data as the total samples and `DEFINE_BATCH_SIZE` as the size of one batch. Then the completion of the following two loops represents the end of one round of Epoch.

```
for (int i = 0; i < GUI.BATCH_SET_SIZE / DEFINE_BATCH_SIZE; i++) {
    for (int j = 0; j < DEFINE_BATCH_SIZE; j++) {
        // Put in
        int randomIndex = rand.nextInt(GUI.BATCH_SET_SIZE);
        batch.add(allImg.get(randomIndex));
    }
    trainBatch(batch);
    batch.clear();
}
```