
기술문서 Technology Document

CNN 소스코드 : https://github.com/zkdlu/CS_Convolution-Neural-Network

DNN 소스코드 : https://github.com/zkdlu/CS_Deeplearning_without_library

● Convolution Neural Network

1. 심층 신경망(Deep Neural Network) 이란?

: 심층 신경망과 일반적인 신경망의 핵심적인 차이점은 층의 깊이 입니다. 전통적인 기계학습 알고리즘은 하나의 입력층과 하나의 출력층으로 이루어져 있으며 많아야 중간에 하나의 은닉층을 가지고 있습니다. 이러한 구조의 신경망을 얇은 신경망이라 하고, 입력층과 출력층을 포함해 3계층 이상인 즉, 2개 이상의 은닉층을 가진 경우에 깊은 신경망이라고 합니다.

2. 구현

: 이전의 인공 신경망과 같은 원리의 경사 하강법을 사용하여 식을 구하는 과정 및 일부 코드는 생략하겠습니다.

```
NNetwork *NewNetwork(int inputNodes, int hiddenNodes, int hiddenNodes2, int hiddenNodes3,
int hiddenNodes4, int hiddenNodes5, int hiddenNodes6, int hiddenNodes7,
int outputNodes, double learningRate, int bias)
{
    NNetwork *network = (NNetwork *)malloc(sizeof(NNetwork));
    network->inputNodes = inputNodes;
    network->hiddenNodes = hiddenNodes;
    network->hiddenNodes2 = hiddenNodes2;
    network->hiddenNodes3 = hiddenNodes3;
    network->hiddenNodes4 = hiddenNodes4;
    network->hiddenNodes5 = hiddenNodes5;
    network->hiddenNodes6 = hiddenNodes6;
    network->hiddenNodes7 = hiddenNodes7;
    network->outputNodes = outputNodes;
    network->learningRate = learningRate;
    network->bias = bias;

    srand((unsigned int)time(NULL));

    network->input_hidden = InitWeight(network, inputNodes, hiddenNodes);
    network->hidden1_hidden2 = InitWeight(network, hiddenNodes, hiddenNodes2);
    network->hidden2_hidden3 = InitWeight(network, hiddenNodes2, hiddenNodes3);
    network->hidden3_hidden4 = InitWeight(network, hiddenNodes3, hiddenNodes4);
    network->hidden4_hidden5 = InitWeight(network, hiddenNodes4, hiddenNodes5);
```

```

network->hidden5_hidden6 = InitWeight(network, hiddenNodes5, hiddenNodes6);
network->hidden6_hidden7 = InitWeight(network, hiddenNodes6, hiddenNodes7);
network->hidden_output = InitWeight(network, hiddenNodes7, outputNodes);

network->bias_weightIH = InitBias(network, inputNodes);
network->bias_weightHH = InitBias(network, hiddenNodes);
network->bias_weightHH2 = InitBias(network, hiddenNodes2);
network->bias_weightHH3 = InitBias(network, hiddenNodes3);
network->bias_weightHH4 = InitBias(network, hiddenNodes4);
network->bias_weightHH5 = InitBias(network, hiddenNodes5);
network->bias_weightHH6 = InitBias(network, hiddenNodes6);
network->bias_weightH0 = InitBias(network, hiddenNodes7);

network->err_output = (double *)malloc(sizeof(double) * outputNodes);

return network;
}

```

입력노드, 은닉노드, 출력노드의 수와 학습률과 bias값을 인자로 하여 NNetwork 형식 값을 메모리에 할당하여 생성합니다.

```

double Train(NNetwork *network, double *inputDatas, double *targetDatas)
{
    //Forward Propagation
    double *net_hidden1 = WeightSum(inputDatas, network->input_hidden, network->inputNodes, network->hiddenNodes);
    AddBias(net_hidden1, network->bias, network->bias_weightIH, network->hiddenNodes);
    double *out_hidden1 = ReLu(net_hidden1, network->hiddenNodes);

    double *net_hidden2 = WeightSum(out_hidden1, network->hidden1_hidden2, network->hiddenNodes, network->hiddenNodes2);
    AddBias(net_hidden2, network->bias, network->bias_weightHH, network->hiddenNodes2);
    double *out_hidden2 = ReLu(net_hidden2, network->hiddenNodes2);

    double *net_hidden3 = WeightSum(out_hidden2, network->hidden2_hidden3, network->hiddenNodes2, network->hiddenNodes3);
    AddBias(net_hidden3, network->bias, network->bias_weightHH2, network->hiddenNodes3);
    double *out_hidden3 = ReLu(net_hidden3, network->hiddenNodes3);

    double *net_hidden4 = WeightSum(out_hidden3, network->hidden3_hidden4, network->hiddenNodes3, network->hiddenNodes4);
    AddBias(net_hidden4, network->bias, network->bias_weightHH3, network->hiddenNodes4);
    double *out_hidden4 = ReLu(net_hidden4, network->hiddenNodes4);

    double *net_hidden5 = WeightSum(out_hidden4, network->hidden4_hidden5, network->hiddenNodes4, network->hiddenNodes5);
    AddBias(net_hidden5, network->bias, network->bias_weightHH4, network->hiddenNodes5);
    double *out_hidden5 = ReLu(net_hidden5, network->hiddenNodes5);
}

```

```

        double *net_hidden6 = WeightSum(out_hidden5, network->hidden5_hidden6, network-
>hiddenNodes5, network->hiddenNodes6);
        AddBias(net_hidden6, network->bias, network->bias_weightHH5, network-
>hiddenNodes6);
        double *out_hidden6 = ReLu(net_hidden6, network->hiddenNodes6);

        double *net_hidden7 = WeightSum(out_hidden6, network->hidden6_hidden7, network-
>hiddenNodes6, network->hiddenNodes7);
        AddBias(net_hidden7, network->bias, network->bias_weightHH6, network-
>hiddenNodes7);
        double *out_hidden7 = ReLu(net_hidden7, network->hiddenNodes7);

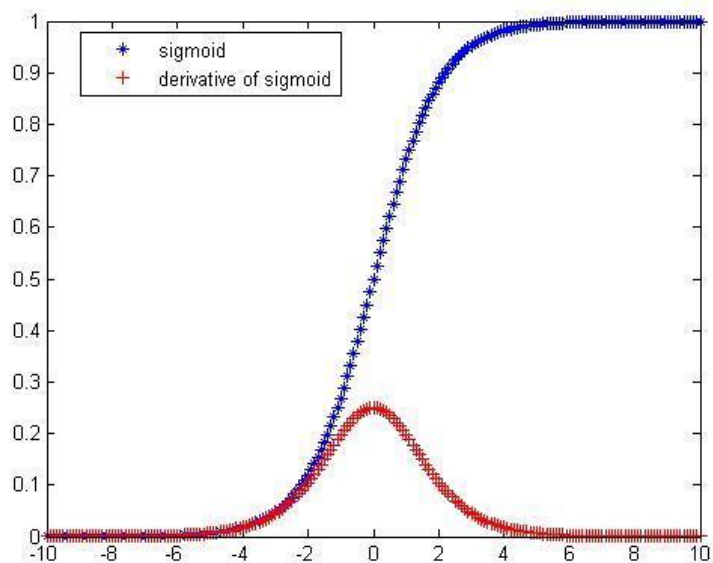
        double *net_output = WeightSum(out_hidden7, network->hidden_output, network-
>hiddenNodes7, network->outputNodes);
        AddBias(net_output, network->bias, network->bias_weightH0, network->outputNodes);
        double *out_output = Sigmoid(net_output, network->outputNodes);

        for (int i = 0; i < network->outputNodes; i++)
        {
            network->err_output[i] = (double)1 / 10 * pow((targetDatas[i] -
out_output[i]), 2);
        }

```

입력 데이터와 목표 데이터를 인자로 하여 순전파 (Forward Propagation)을 수행한다. 가중치와 각 노드의 입력값을 곱하여 나온 값에 활성화 함수로 ReLU를 적용하여 출력한다. ReLU함수란 Rectified Linear Unit의 약어로 $f(x) = \max(0, x)$ 와 같다.

흔히 활성화 함수로 Sigmoid함수를 사용하게 되는데, 이는 뉴런의 동작방식과 유사하기 때문이다. 하지만 신경망을 학습하기 위해서는 미분 결과를 첫 번째 layer까지 전달 해야 하는데, Sigmoid함수의 미분 값은 아래 그림과 같다.



Sigmoid는 출력 값을 0-1로 제한을 하고, 미분 값 또한 0-0.25 사이로 나오게 된다. 이는 미분 값이 앞의 layer로 전달 될 때 마다, 미분 결과가 최소 약 1/4로 줄어들어 결국에는 기울기값 이 소실되는 (Gradient Vanishing) 상황이 생길 수 있다. 이러한 문제를 해결하기 위해 ReLU를 새로운 활성화 함수로 사용한다.

```
//Back Propagation
//Back Propagation
double prevErr[1024] = { 0 };
for (int i = 0; i < network->hiddenNodes7; i++)
{
    for (int j = 0; j < network->outputNodes; j++)
    {
        double gradient = (out_output[j] - targetDatas[j]) / 5.0 * (out_output[j] *
(1 - out_output[j])) * out_hidden7[i];
        prevErr[i] += (out_output[j] - targetDatas[j]) / 5.0 * (out_output[j] * (1 -
out_output[j])) * network->hidden_output[i][j];

        network->hidden_output[i][j] -= network->learningRate * gradient;
        if (i == 0)
        {
            network->bias_weightH0[j] -= network->learningRate * ((out_output[j]
- targetDatas[j]) / 5.0 * (out_output[j] * (1 - out_output[j])));
        }
    }
}

double prev2Err[1024] = { 0 };
for (int i = 0; i < network->hiddenNodes6; i++)
{
    for (int j = 0; j < network->hiddenNodes7; j++)
    {
        double gradient = prevErr[j] * (out_hidden7[j] > 0 ? 1 : 0) * out_hidden6[i];
        prev2Err[i] += prevErr[j] * (out_hidden7[j] > 0 ? 1 : 0) * network-
>hidden6_hidden7[i][j];

        network->hidden6_hidden7[i][j] -= network->learningRate * gradient;
        if (i == 0)
        {
            network->bias_weightHH6[j] -= network->learningRate * prevErr[j] *
(out_hidden7[j] > 0 ? 1 : 0);
        }
    }
}
..
.
.
중략
```

경사 하강법을 통해 기울기 값을 첫 번째 layer로 전파하는 역전파 (Back - Propagation) 과정을 수행한다

```
double* Query(NNetwork *network, double *inputDatas)
```

```

{
    double *net_hidden1 = WeightSum(inputDatas, network->input_hidden, network-
>inputNodes, network->hiddenNodes);
    AddBias(net_hidden1, network->bias, network->bias_weightIH, network->hiddenNodes);
    double *out_hidden1 = ReLu(net_hidden1, network->hiddenNodes);

    double *net_hidden2 = WeightSum(out_hidden1, network->hidden1_hidden2, network-
>hiddenNodes, network->hiddenNodes2);
    AddBias(net_hidden2, network->bias, network->bias_weightHH, network->hiddenNodes2);
    double *out_hidden2 = ReLu(net_hidden2, network->hiddenNodes2);

    double *net_hidden3 = WeightSum(out_hidden2, network->hidden2_hidden3, network-
>hiddenNodes2, network->hiddenNodes3);
    AddBias(net_hidden3, network->bias, network->bias_weightHH2, network->hiddenNodes3);
    double *out_hidden3 = ReLu(net_hidden3, network->hiddenNodes3);

    double *net_hidden4 = WeightSum(out_hidden3, network->hidden3_hidden4, network-
>hiddenNodes3, network->hiddenNodes4);
    AddBias(net_hidden4, network->bias, network->bias_weightHH3, network->hiddenNodes4);
    double *out_hidden4 = ReLu(net_hidden4, network->hiddenNodes4);

    double *net_hidden5 = WeightSum(out_hidden4, network->hidden4_hidden5, network-
>hiddenNodes4, network->hiddenNodes5);
    AddBias(net_hidden5, network->bias, network->bias_weightHH4, network->hiddenNodes5);
    double *out_hidden5 = ReLu(net_hidden5, network->hiddenNodes5);

    double *net_hidden6 = WeightSum(out_hidden5, network->hidden5_hidden6, network-
>hiddenNodes5, network->hiddenNodes6);
    AddBias(net_hidden6, network->bias, network->bias_weightHH5, network->hiddenNodes6);
    double *out_hidden6 = ReLu(net_hidden6, network->hiddenNodes6);

    double *net_hidden7 = WeightSum(out_hidden6, network->hidden6_hidden7, network-
>hiddenNodes6, network->hiddenNodes7);
    AddBias(net_hidden7, network->bias, network->bias_weightHH6, network->hiddenNodes7);
    double *out_hidden7 = ReLu(net_hidden7, network->hiddenNodes7);

    double *net_output = WeightSum(out_hidden7, network->hidden_output, network-
>hiddenNodes7, network->outputNodes);
    AddBias(net_output, network->bias, network->bias_weightH0, network->outputNodes);
    double *out_output = Sigmoid(net_output, network->outputNodes);

    free(net_hidden1);
    free(net_hidden2);
    free(net_hidden3);
    free(net_hidden4);
    free(net_hidden5);
    free(net_hidden6);
    free(net_hidden7);

    return out_output;
}

```

모든 학습이 끝나면 신경망에 테스트 데이터를 입력하여 순전파(Forward Propagation)을 수행하여 출력 노드로 출력되는 값을 확인합니다.

```
C:\WINDOWS\system32\cmd.exe
6 => 6 : 0.983717      correct
4 => 4 : 0.999316      correct
1 => 1 : 0.999831      correct
7 => 7 : 0.998829      correct
2 => 3 : 0.998398
6 => 6 : 0.986632      correct
5 => 6 : 0.779295
0 => 0 : 1.000000      correct
1 => 1 : 0.998892      correct
2 => 2 : 0.994032      correct
3 => 3 : 0.995334      correct
4 => 4 : 0.999986      correct
5 => 5 : 0.999506      correct
6 => 6 : 0.999451      correct
7 => 7 : 0.999856      correct
8 => 8 : 0.996520      correct
9 => 9 : 0.999590      correct
0 => 0 : 0.999784      correct
1 => 1 : 0.999818      correct
2 => 2 : 0.999933      correct
3 => 3 : 0.997446      correct
4 => 4 : 0.999962      correct
5 => 5 : 0.997942      correct
6 => 6 : 0.999839      correct
0 => 8 : 0.446485
[ 9721 / 10001 = 97.200280%]
계속하려면 아무 키나 누르십시오 . . .
```

[사진]. 테스트 결과

3. 합성곱 신경망 CNN (Convolution Neural Network)

: 기존의 신경망은 이미지를 인식할 때 여러가지의 문제점이 있다. 예를 들어 이미지가 회전되어 있거나, 크기가 커지거나, 이미지에 변형이 조금만 생기면 새로운 학습 데이터를 넣어주지 않으면 좋은 결과를 기대하기 어려운데 CNN을 사용하면, 이런 문제를 해결할 수 있다. CNN은 일반적인 뉴럴 네트워크 앞에 전처리 과정인 여러 계층의 컨볼루션 계층을 붙인 형태이다. 앞의 컨볼루션 계층을 통해 입력 받은 이미지에 대한 특징(Feature)를 추출하게 되고, 이렇게 추출된 특징을 기반으로 기존 뉴럴 네트워크를 이용하여 분류를 하게 된다.

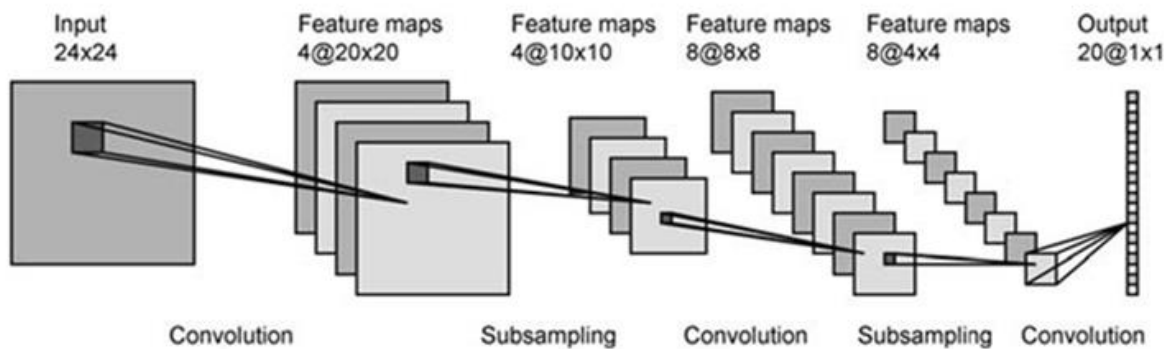
CNN의 과정은 다음과 같은 다음과 같이 크게 3단계로 이루어진다.

1. 특징을 추출하는 단계. – Convolution
2. topology변화에 영향을 받지 않도록 하는 단계. – Sub sampling

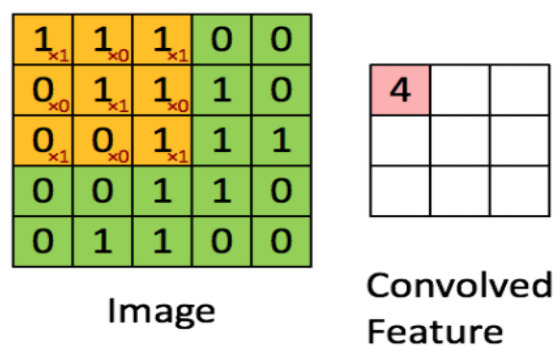
3. 분류 단계 – Classification

대부분의 영상 인식 알고리즘에서는 특징을 추출하기 위해 filter를 사용한다. 보통 짝는 5x5 혹은 3x3과 같은 작은 영역에 대해 적용 하며, 필터에 사용하는 계수들의 값에 따라 각각 다른 특징을 얻을 수 있다. 일반적으로 이 filter의 계수들은 특정 목적에 따라 고정되지만, CNN에서 사용하는 filter 혹은 Convolutional Layer는 학습을 통해 최적의 계수를 결정할 수 있게 하는 점이 다르다. 통상적으로 sub-sampling은 보통 고정된 위치에 있는 픽셀을 고르거나, 혹은 sub-sampling 윈도우 안에 있는 픽셀들의 평균을 취한다.

이동이나 변형 등에 무관한 학습 결과를 보이려면, 강하고 글로벌한 특징을 추출해야 하는데, 이를 위해 통상적으로 Convolution + Sub sampling 과정을 여러 번을 거치게 되면, 좀 더 전체 이미지를 대표할 수 있는 글로벌한 특징을 얻을 수 있게 된다. 이렇게 얻어진 특징을 fully-connected network를 통해 학습을 시키게 되면, 2차원 영상 정보로부터 topology 변화에 강인한 인식 능력을 갖게 된다.

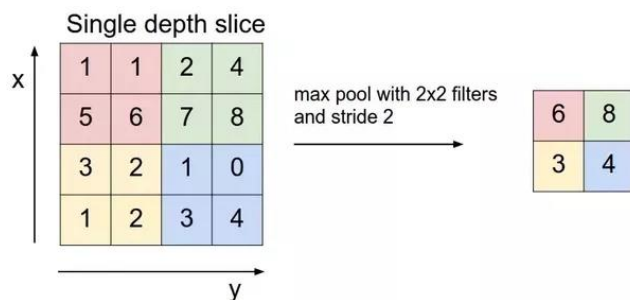


컨볼루션은 신호처리 분야에서 가장 많이 사용되는 연산 중 하나입니다. 기본적으로 입력 신호에 특정 형태의 필터를 씌어 슬라이딩하여 결과를 얻습니다.

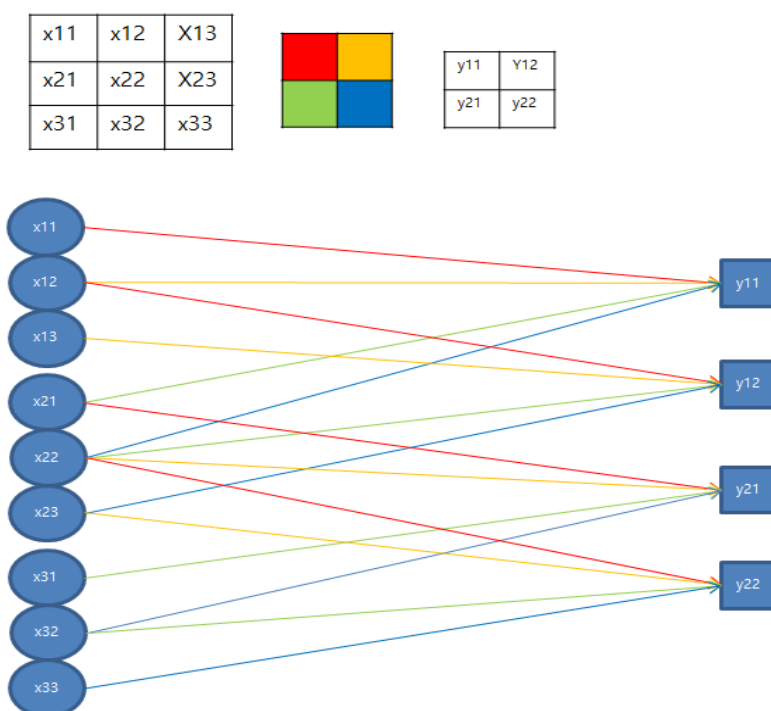


[그림]. 합성곱 연산

서브 샘플링 단계에서는 풀링 작업을 하게 되는데, 인접한 유닛들을 이용하여 가장 큰 값만 내보내거나, 평균값을 내보낸다. 즉 인접한 값들을 표현하기 위한 방법이다. 이 과정을 통해 신경망은 파라미터의 개수나 연산량을 줄일 수 있고, 학습 하지 않은 이미지에 대한 성능이 향상 된다.

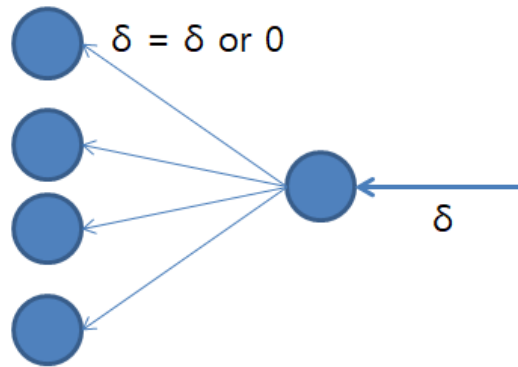


컨볼루션 및 풀링 레이어 또한 역전파를 하게 되는데, 컨볼루션 단계의 역전파를 이해하기 위해 일반 신경망의 그림으로 변환하면 다음과 같다.



일반적인 신경망에서는 다음 레이어의 각 노드가 이전 레이어의 모든 노드들과 연결된 가중치가 존재한다. 하지만 합성곱 신경망에서의 Convolution 단계는 위의 그림에서 보듯이, 지역적인 연결이 이루어진 신경망으로 이해가 가능하다. 따라서 각 가중치가 연결된 노드에 영향을 끼친 정도를 구하여, 에러 역전파 및 가중치에 해당하는 필터를 갱신한다.

풀링 레이어의 역전파 또한 아래의 그림으로 보면 이해가 쉽다. 흔히 최대값 풀링으로 2x2 윈도우의 안에서 풀링 작업을 하게 되므로, 해당 윈도우 안에서 풀링이 된 위치에만 역전파를 하고 다른 공간은 0이 된다.



[그림]. 풀링 레이어의 역전파

4. 구현

Mat 클래스를 구현하여 CNN에 사용하였습니다.

```
public double Train(Mat<double> image, double[] targetDatas)
{
    //Forward Propagation
    Mat<double> convolved = Convolution(image, kernels1);
    active_convolved = ApplyReLU(convolved);
    Mat<double> pooled = MaxPooling(active_convolved);

    Mat<double> convolved2 = Convolution(pooled, kernels2);
    active_convolved2 = ApplyReLU(convolved2);
    Mat<double> pooled2 = MaxPooling(active_convolved2);

    Mat<double> convolved3 = Convolution(pooled2, kernels3);
    active_convolved3 = ApplyReLU(convolved3);
}
```

위의 순전파 과정에서는 컨볼루션->활성화->풀링->컨볼루션->활성화->풀링->컨볼루션 단계의 작업 후 분류 작업을 하였다.

컨볼루션 단계의 출력 행렬을 vectorization 하여 Fully-Connected-Layer에 입력 값으로 주게 된다.

Fully-Conncted Layer단계의 신경망 연산은 기존의 신경망의 순전파 단계와 역전파 단계는 동일하다.

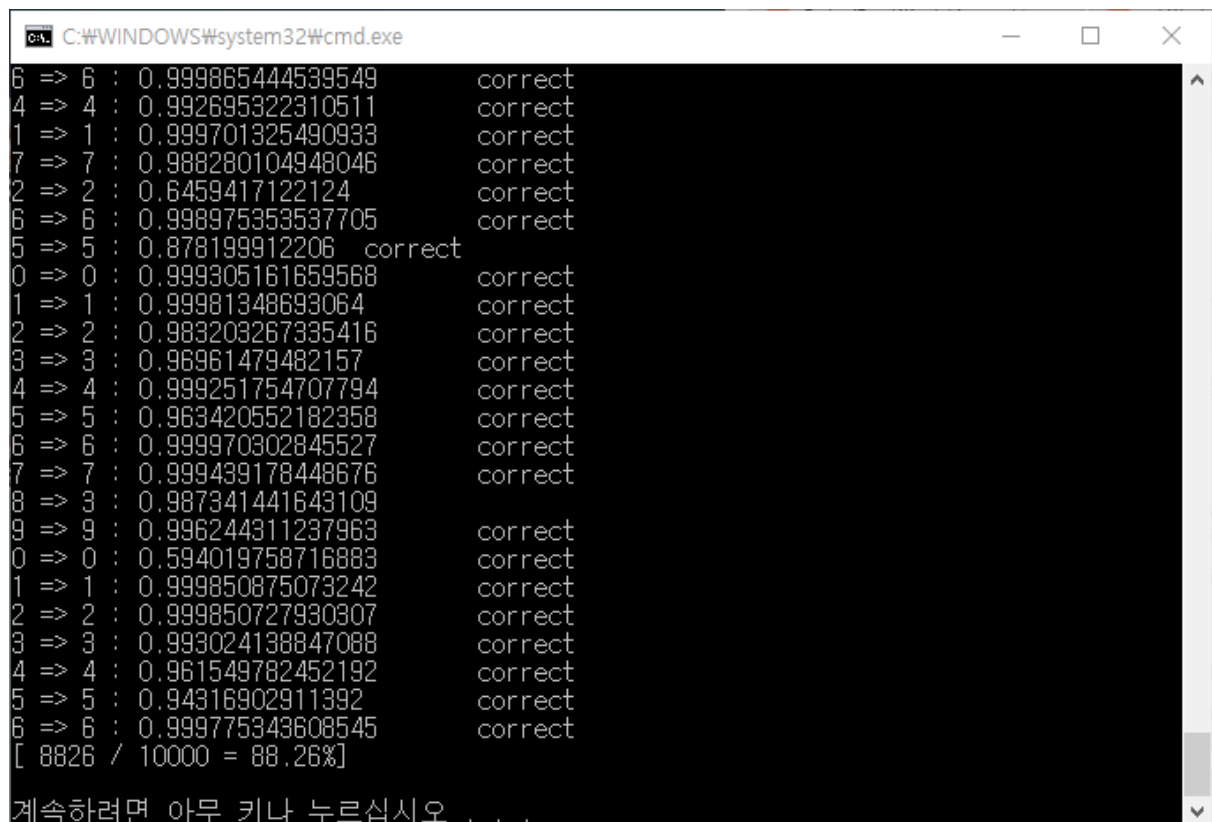
```
Mat<double> errorMat = Mat<double>.ConvertToMat(prevErr);
Mat<double> pool2Gradient = ConvPropagateGradient(errorMat, active_convolved3, kernels3);

Mat<double> conv2Gradient = PoolPropagateGradient(pool2Gradient, pooled2, active_convolved2);
Mat<double> poolGradient = ConvPropagateGradient(conv2Gradient, active_convolved2, kernels2);

Mat<double> convGradient = PoolPropagateGradient(poolGradient, pooled, active_convolved);

UpdateKernel(errorMat, active_convolved3, pooled2, kernels3);
UpdateKernel(conv2Gradient, active_convolved2, pooled, kernels2);
UpdateKernel(convGradient, active_convolved, image, kernels1);
```

앞에 나온 방법을 이용하여 각 단계의 그래디언트 값을 구한 후, 각 컨볼루션 레이어의 커널을 학습시키게 된다.



```
C:\WINDOWS\system32\cmd.exe
6 => 6 : 0.999865444539549      correct
4 => 4 : 0.992695322310511      correct
1 => 1 : 0.999701325490933      correct
7 => 7 : 0.988280104948046      correct
2 => 2 : 0.6459417122124        correct
6 => 6 : 0.998975353537705      correct
5 => 5 : 0.878199912206         correct
0 => 0 : 0.999305161659568      correct
1 => 1 : 0.99981348693064      correct
2 => 2 : 0.983203267335416      correct
3 => 3 : 0.96961479482157      correct
4 => 4 : 0.999251754707794      correct
5 => 5 : 0.963420552182358      correct
6 => 6 : 0.999970302845527      correct
7 => 7 : 0.999439178448676      correct
8 => 3 : 0.987341441643109
9 => 9 : 0.996244311237963      correct
0 => 0 : 0.594019758716883      correct
1 => 1 : 0.999850875073242      correct
2 => 2 : 0.999850727930307      correct
3 => 3 : 0.993024138847088      correct
4 => 4 : 0.961549782452192      correct
5 => 5 : 0.94316902911392      correct
6 => 6 : 0.999775343608545      correct
[ 8826 / 10000 = 88.26%]
계속하려면 아무 키나 누르십시오 . . .
```

[사진]. 합성곱 테스트 결과