인공지능

과제 4 Open source software: Weka

백은옥 교수님  
전민철, 조현진, 최승혁 조교님

2013011424  
성예닮  
컴퓨터공학부

1. 환경

운영체제: Windows 10  
언어: Java 8  
개발 IDE: IntelliJ IDEA

2. Sigmoid to ReLU

Sigmoid:   
ReLU:

Sigmoid는 전달된 값을 0과 1 사이로 변형을 한다. 일정 비율로 줄어들기 때문에 왜곡은 아니지만, 값이 현저하게 작아지는 현상이 벌어진다. 3개의 layer를 거치면서 계속해서 1/10로 줄어들었다면, 현재 사용 중인 값은 처음 값의 1/1000이 된다. 이렇게 작아진 값을 갖고 원래의 영향력을 그대로 복원한다는 것은 불가능하다.

Backpropagation에서 전달하는 값이 레이어가 깊어질수록, 처음 레이어에 미치는 영향이 너무 작아지고, 이 문제를 Gradient Vanish라 한다.

Gradient Vanish를 해결하기 위해 나온 방법이 ReLU이다.

처음 레이어 방향으로 가더라도, 큰 값은 그대로 보존하고, 0이하의 값은 0으로 처리하여 영향력을 유지할 수 있다.

또한 히든 레이어의 깊이가 매우 깊고, 유닛이 매우 많은 경우 Sigmoid보다 ReLU의 미분 계산이 더욱 빠르기 때문에 최근에는 ReLU를 가장 많이 사용하고 있다.

**3. 코드 추가 및 수정사항**

**추가 사항**

**RectifiedLinearUnit.java**

@Override  
public double outputValue(NeuralNode node) {  
 // 해당 노드의 가중치 벡터  
 double[] weights = node.getWeights();  
 // 해당 노드의 입력 벡터  
 NeuralConnection[] inputs = node.getInputs();  
 // Bias  
 double value = weights[0];  
  
 // 입력 벡터와 가중치 벡터를 곱해 출력값을 계산한다.  
 for (int noa = 0; noa < node.getNumInputs(); noa++) {  
 value += inputs[noa].outputValue(true)  
 \* weights[noa+1];  
 }  
  
 // 출력부에서의 활성 함수를 ReLU로 한다.  
 value = Math.*max*(0, value);  
 return value;  
}

**출력 값을 반환하는 함수로, 입력, 가중치 벡터를 곱해 모두 더한 후, ReLU 유닛을 통과시켜 출력값을 계산한다.**

public double errorValue(NeuralNode node) {  
 // 해당 노드의 출력 벡터  
 NeuralConnection[] outputs = node.getOutputs();  
 int[] oNums = node.getOutputNums();  
 double error = 0;  
  
 // 에러를 계산한다.  
 for (int noa = 0; noa < node.getNumOutputs(); noa++) {  
 error += outputs[noa].errorValue(true)  
 \* outputs[noa].weightValue(oNums[noa]);  
 }  
 return error;  
}

**에러 값을 반환하는 함수로, 출력 벡터로부터 에러 값을 모두 더해 반환한다.**

public void updateWeights(NeuralNode node, double learn, double momentum) {  
 // 해당 노드의 입력 벡터  
 NeuralConnection[] inputs = node.getInputs();  
 // 변화된 가중치 벡터  
 double[] cWeights = node.getChangeInWeights();  
 // 해당 노드의 가중치 벡터  
 double[] weights = node.getWeights();  
 double learnTimesError = 0;  
 learnTimesError = learn \* node.errorValue(false);  
 double c = learnTimesError + momentum \* cWeights[0];  
 // Bias  
 weights[0] += c;  
 cWeights[0] = c;  
  
 int stopValue = node.getNumInputs() + 1;  
 for (int noa = 1; noa < stopValue; noa++) {  
 c = learnTimesError \* inputs[noa-1].outputValue(false);  
 // 가속도를 곱해주어 더 빠르게 업데이트 할 수 있도록 한다.  
 c += momentum \* cWeights[noa];  
  
 // 가중치를 업데이트한다.  
 weights[noa] += c;  
 cWeights[noa] = c;  
 }  
}

**가중치를 업데이트하는 함수로, 입력 벡터, 가중치 벡터와 학습률을 이용하여 가중치 벡터를 업데이트 한다. Momentum이라는 가속도 변수를 이용하여 더욱 빠르게 업데이트 할 수 있도록 한다.**

**수정 사항**

**MultilayerPerceptron.java**

**(추가된 부분은 찾기 편하도록 // 추가 라는 주석으로 처리)**

*/\*\*  
 \* ReLU Unit입니다.  
 \*/*private final RectifiedLinearUnit m\_rectifiedLinearUnit;

**MultilayerPerceptron 클래스의 멤버변수로 ReLU 유닛을 추가하였다.**

// ReLU Unit 객체를 생성한다.  
m\_rectifiedLinearUnit = new RectifiedLinearUnit();

**멤버변수에 ReLU 유닛을 생성자로 생성한다.**

// Sigmoid Unit을 ReLU로 대체한다.  
// NeuralNode temp = new NeuralNode(String.valueOf(m\_nextId), m\_random,  
// m\_sigmoidUnit);  
NeuralNode temp = new NeuralNode(String.*valueOf*(m\_nextId), m\_random,  
 m\_rectifiedLinearUnit);

**MultilayerPerceptron 클래스의 setupOutputs() 메서드에서 Sigmoid를 ReLU로 대체한다.  
이번 과제에서는 회귀 문제가 아닌 분류 문제기 때문에 Sigmoid를 그대로 사용해도 상관 없다.  
히든 유닛의 개수가 늘어날수록 ReLU와 Sigmoid의 차이는 줄어들고,  
히든 유닛의 개수가 적어질수록 Sigmoid가 더 나은 성능을 보인다.(실험 결과를 토대로)**

// Hidden layer에서 Sigmoid Unit 대신 ReLU Unit을 사용한다.  
// NeuralNode temp = new NeuralNode(String.valueOf(m\_nextId), m\_random,  
// m\_sigmoidUnit);  
NeuralNode temp = new NeuralNode(String.*valueOf*(m\_nextId), m\_random,  
 m\_rectifiedLinearUnit);

**MultilayerPerceptron 클래스의 setupHiddenLayer() 메서드에서 Sigmoid를 ReLU로 대체한다.  
히든 레이어에서의 활성함수를 ReLU로 변경한 것이다.**

// Sigmoid를 ReLU로 변경  
// ((NeuralNode) m\_neuralNode).setMethod(m\_sigmoidUnit);  
((NeuralNode) m\_neuralNode).setMethod(m\_rectifiedLinearUnit);

**MultilayerPerceptron 클래스의 setEndsToLinear() 메서드에서 Sigmoid를 ReLU로 대체한다.**

**(실제로는 LinearUnit으로 계산되기에 해당 코드는 무의미하다.)**

**4. 실행 파라미터**

**Default: -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a**

**수정된 파라미터: -L 0.01 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H 100**

**-L, -H에 해당하는 파라미터를 수정하였고, 이는 각각 학습률과 히든 레이어의 Unit 깊이 및 개수에 해당한다.**

**ReLU로 활성함수를 변경한 후, 파라미터를 수정하지 않고 실행했을 경우, 정답률이 14%정도에 불과하였다. 이는 여러가지 이유가 있을 수 있으나, Sigmoid 함수의 경우 에러 값이 수렴했으나, ReLU의 경우 에러 값이 수렴하지 않는 경우라고 볼 수 있다. 이를 해결하는 방법으로 학습률과 히든 레이어의 개수를 변경해보았다. 학습률은 MNIST 데이터 등에서 가장 많이 쓰이는 0.01로 변경하였고, 히든 레이어의 수는 100으로 변경하였더니 괜찮은 결과가 나오는 것을 볼 수 있었다. 히든 레이어의 깊이를 변경한 경우(100,100 or 60,60)에는 과적합이 일어나는지 제대로 학습이 되지 않았다.**

**Sigmoid의 Default 파라미터의 경우 정답률이 79%,  
ReLU의 수정된 파라미터의 경우 정답률이 81%임을 볼 수 있었다.**

**5. 결과 파일**

**assignment4/results/result\_final.txt 파일을 첨부하였습니다.**

**6. 소스코드를 컴파일하는 방법**

**1) IntelliJ IDEA 프로젝트이기 때문에, assignmen4 폴더를 다운로드하여 IntelliJ에서 Import 한다.  
File -> New -> Project From Existing sources -> 다운로드 받은 경로 입력 후 OK  
-> Create project from existing sources -> Project name 입력 후 모두 Overwrite**

**7. 기타 사항**

1) 결과 파일의 개행 문자가 CRLF가 아닌 LF기 때문에, Windows 메모장에서는 개행이 되지 않는다. UNIX 개행 문자를 지원하는 에디터(Sublime text, atom 등)으로 출력 파일을 확인해야 한다.