Praktikum Adaptive Systeme

Aufgabenblatt 01 - Aufgaben 3a+b

1 Spezifikation

Ziel ist die Implementierung eines Perzeptrons und des Perzeptron-Lernalgorithmus. Das Perzeptron soll dabei mit Mustern von Irispflanzen (bestehend aus der Höhe und Breite des Kelchblattes dieser Pflanzen) trainiert werden, so daß es in der Lage ist, auch zu Mustern mit unbekannter Klassenzugehörigkeit erfolgreich den Iris-Typ zu ermitteln. Siehe dazu Abbildung 1.

Kelchblatt Breite [cm]	Kelchblatt Hhe [cm]	Iris-Art
4.9	3.0	iris setosa
4.7	3.2	iris setosa
5.1	3.5	iris setosa
4.6	3.1	iris setosa
5.0	3.6	iris setosa
5.4	3.9	iris setosa
4.6	3.4	iris setosa
6.3	3.3	iris virginica
5.8	2.7	iris virginica
7.1	3.0	iris virginica
6.3	2.9	iris virginica
6.5	3.0	iris virginica
7.6	3.0	iris virginica
4.9	2.5	iris virginica
5.1	3.3	unbekannt
5.9	3.0	unbekannt
5.7	2.5	unbekannt
5.4	3.7	unbekannt

Abbildung 1: Vorgegebene Trainings- und Testdaten.

Die Muster der ersten 14 Tabelleneinträge stellen dabei die Trainingsdaten dar. Nach Abschluß des Trainings soll zu den letzten vier Tabelleneinträgen mit Hilfe des Perzeptrons dann der Iris-Typ bestimmt werden.

Das zu entwickelnde Softwaresystem soll außerdem ein separat zur Verfügung gestelltes Plotterpaket verwenden, um die Ergebnisse zur visualisieren.

2 Entwicklungsdokumentation

Der wesentliche Punkt bei der Entwicklung des geforderten Softwaresystems ist eine geeignete Modellierung des Perzeptrons, wie es in der Einleitung des ersten Aufgabenblattes dargestellt ist. Wir haben uns dazu entschieden, unser Perzeptron mit den folgenden Eigenschaften auszustatten:

1. Es kann ein Perzeptron mit einer beliebigen Anzahl von Eingaben erzeugt werden. Dabei ist sowohl eine feste, als auch eine zufällige Initialisierung der Verbindungsgewichte zu diesen Eingaben möglich.

2. Möglichkeit, die Verbindungsgewichte des Perzeptrons mit Hilfe des Perzeptron-Lernalgorithmus lernen zu lassen, wobei das Training automatisch abgebrochen wird, sobald zuvor spezifizierte Abbruchbedingungen erfüllt sind.

3. Möglichkeit, während/nach dem Training eine Statistik der Netzperformance auszugeben, um die Qualität des Lernfortschrittes beurteilen zu können.

4. Die konkrete Implementierung soll nicht nur mit fehlerhaften Eingaben zurecht kommen, sondern diese Eingaben auch sinnvoll weiterverarbeiten.

2.1 Initialisierung der Gewichte

Sowohl eine feste, als auch eine zufällige Initialisierung der Gewichte erschien uns sinnvoll, da wir vor Beginn des Trainings des Perzeptrons eine flexible Wahl der anfänglichen Hyperebene (die durch den Gewichtsvektor des Perzeptrons definiert wird) ermöglichen wollten. Auf diese Weise ist es möglich, sowohl mit zufällig gewählten, als auch mit anfänglich fest vorgegebenen Hyperebenen den Lernvorgang des Perzeptrons zu starten.

Unsere Modellierung des Perzeptrons ist außerdem unabhängig von der konkreten Anzahl der Eingaben. Dadurch ist es möglich, für jede beliebige Anzahl von Eingaben ein Perzeptron erzeugen zu können. Das Perzeptron wird auf diese Weise allgemein verwendbar und ist nicht auf die hier geforderte Klassifikation von Mustern mit zwei Komponenten beschränkt.

2.2 Abbruchkriterien des Trainings

Eine wichtige Eigenschaft beim Training unseres Perzeptrons soll die Angabe von Abbruchbedingungen sein. Da das Training solange fortgesetzt werden soll, bis möglichst viele Muster korrekt klassifiziert werden, der Trainingsalgorithmus aber nicht unendlich lange laufen soll (falls die Muster beispielsweise nicht durch eine Hyperebene trennbar sind, wie das beim XOR-Problem der Fall ist), ist es erforderlich, hier klare Vorgaben zu machen. Es ist also eine Abwägung sinnvoll, zwischen dem Aufwand, den wir bereit sind beim Training des Perzeptrons zu investieren, und der Qualität des Trainings in Form von der Anzahl der korrekt klassifizierten Muster. Daher haben wir die Möglichkeit integriert, beim Training eine maximale Anzahl auszuführender Trainingsschritte

und eine maximale Anzahl von tolerierbaren falsch klassifizierten Mustern vorzugeben.

Das Training wird beendet, sobald die maximale Anzahl vorgegebener Iterationen erreicht wurde oder die Anzahl fehlerhaft klassifizierter Muster den vorgegebenen Schwellenwert unterschritten hat.

2.3 Trainingsstatistik

Die dritte Eigenschaft, das Perzeptron mit der Fähigkeit auszustatten, selbständig eine Statistik über den Lernvorgang mitzuführen, erschien uns ebenfalls wichtig. Die Statistik ist außerdem erforderlich, um die Abbruchbedingungen des Trainingsvorganges zu überpüfen. Der Hauptgrund ist aber, daß wir eine Möglichkeit zur Verfügung stellen wollten, mit der sich der Lernerfolg und der zum Lernen investierte Aufwand des Perzeptrons beurteilen läßt. Aus diesem Grund haben wir die Möglichkeit integriert, sowohl während des Trainings, als auch nach Trainingsabschluss, die Anzahl der fehlklassifizierten Muster aus der Trainingsmenge und die Anzahl der insgesamt durchgeführten Lernschritte abzurufen.

In der nachfolgenden Abbildung kann die Struktur unserer Modellierung nachvollzogen werden:

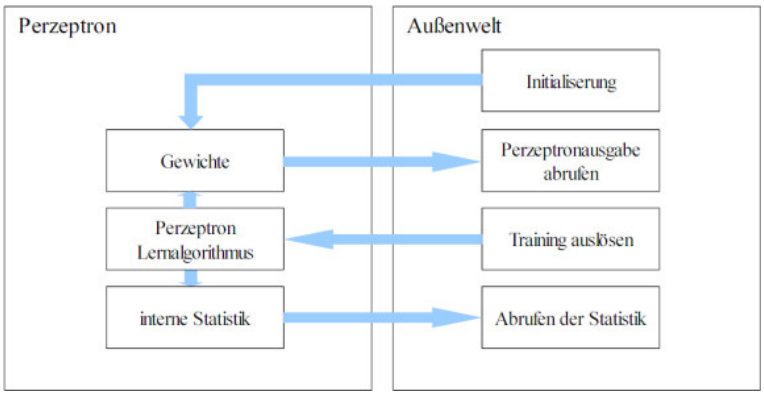

Abbildung 2: Modellierung des Perzeptrons. Das Perzeptron besteht aus einer Anzahl von Gewichten, deren Anzahl und anfängliche Werte von außen bei einer Initialisierung festgelegt werden können. Der Perzeptron-Lernalgorithmus ist fest in die Struktur des Perzeptrons eingebaut und kann mit einer beliebigen Trainingsmenge und der Angabe der zuvor diskutierten Abbruchbedingungen aufgerufen werden. Dieser modifiziert dann die Gewichte des Perzeptrons und baut intern die zuvor erwähnte Statistik über den Trainingsfortschritt auf, welche dann von außen abgerufen werden kann. Unter Angabe eines Eingabevektors kann außerdem die Ausgabe des Perzeptrons anhand der aktuellen Gewichte abgerufen werden.

2.4 Fehlerrobustheit

Die letzte Eigenschaft, die wir in das Perzeptron integrieren wollten, ist die Toleranz der konkreten Java-Implementierung gegenüber unvorschriftsmäßigen Eingaben. Wir haben

es als sinnvoll erachtet, alle Methoden so zu implementieren, daß das Programm bei unvorschriftsmäßigen Eingaben nicht gleich mit einer Fehlermeldung die weitere Ausführung verweigert, sondern weiterhin sinnvoll mit seinen Eingaben umgeht.

Es genügt dabei, Fehlertoleranz nur für die nach außen sichtbaren Schnittstellen zu gewährleisten (mehr kann ein Benutzer nicht beeinflussen), vgl. Abbildung 2. Die einzige Fehlerquelle besteht hier im Auslösen des Trainingsprozesses und der Berechnung der Perzeptron-Ausgabe. Der Trainingsalgorithmus muß nämlich Zugriff auf eine Trainingsmenge besitzen, die diesem in unserer Implementierung von „außen“ zu übergeben ist und die Berechnung der Perzeptron-Ausgabe erfordert die Angabe eines von außen vorgegebenen Eingabevektors. Zu Problemen kann es kommen, wenn die genannten Parameter, die zu übergeben sind, nicht die Form haben, die von den implementierten Java-Funktionen für eine einwandfreie Bearbeitung vorausgesetzt werden. Konkrete Fehlerquellen, die wir diesbezüglich in unserer Implementierung berücksichtigt haben, sind dabei:

1. Die Trainingsmenge enthält eine unterschiedliche Anzahl von Eingabe- und Ausgabemustern. In diesem Fall werden nur so viele Trainingsmuster berücksichtigt, wie Paare (Eingabemuster, Ausgabemuster) existieren.

2. Es gibt Trainingsvektoren innerhalb der Trainingsmenge, die aus *mehr* Eingabewerten bestehen, als das Perzeptron Gewichte besitzt. In diesem Fall werden nur so viele Eingaben berücksichtigt, wie Gewichte vorhanden sind.

3. Es gibt Trainingsvektoren innerhalb der Trainingsmenge, die aus *weniger* Eingabewerten bestehen, als das Perzeptron Gewichte besitzt. In diesem Fall werden die Eingaben mit Nullen aufgefüllt.

4. Es soll eine Perzeptron-Ausgabe zu einem Eingabevektor berechnet werden, der *mehr* Eingabewerte besitzt, als Gewichte vorhanden sind. Die Vorgehensweise soll dann die gleiche sein, wie unter Punkt 2.

5. Es soll eine Perzeptron-Ausgabe zu einem Eingabevektor berechnet werden, der *weniger* Eingabewerte besitzt, als Gewichte vorhanden sind. Die Vorgehensweise ist hier wieder die gleiche, wie unter Punkt 3.

3 Testdokumentation

3.1 Erfüllung der gestellten Anforderungen

Wir haben das Perzeptron zu Testzwecken anfänglich mit den Gewichten $[0.5, 0.5, -4.0]$ initialisiert, wobei das letzte Gewicht die Verbindungsstärke zu einer konstanten 1-Eingabe bezeichnet (sog. Bias). Wir haben uns für diese Wahl entschieden, da die anfängliche Hyperebene in diesem Fall quer durch die Trainingsmenge verläuft und dadurch eine gute Visualisierung (mit gleichen Bildausschnitten) möglich war.

Mit den genannten Gewichten folgt, daß alle Punkte (x, y), durch die die anfängliche Hyperebene des Perzeptrons geht, die Beziehung $x/2 + y/2 - 4 = 0 \iff x + y = 8$ erfüllen. Insbesondere müssen die Punkte $(5, 3)$ und $(5.5, 2.5)$ auf dieser Hyperebene liegen. Anhand der Graphik aus Abbildung 3, links, ist zu erkennen, daß dies gegeben ist und die Hyperebene demnach korrekt eingezeichnet wurde. Die korrekte Position der eingezeichneten Datenpunkte kann dabei anhand der Iristabelle am Anfang dieser Dokumentation nachvollzogen werden.

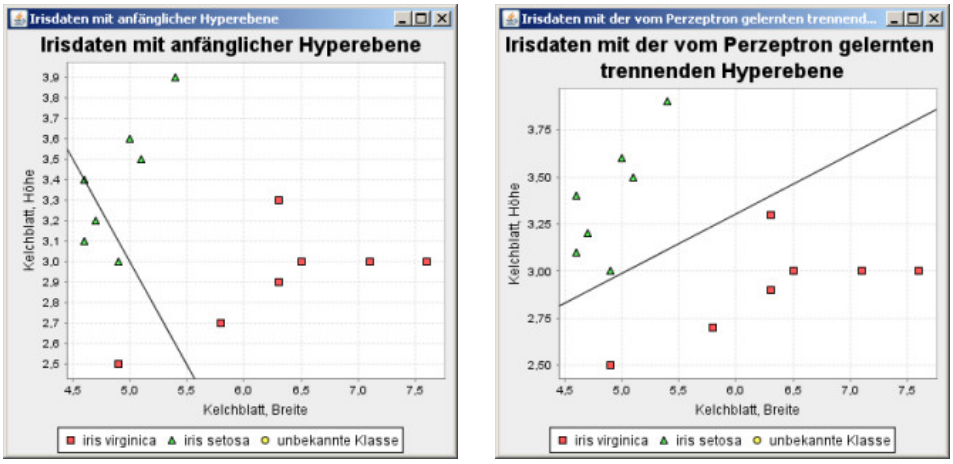

Abbildung 3: *Links:* Irisdaten und Hyperebene vor Beginn des Trainings. *Rechts:* Die gelernte Hyperebene, nachdem das Training abgeschlossen war. Die Abbildungen wurden dabei mit dem Plotterpaket erstellt.

Nach der Gewichtsinitialisierung wurde das Perzeptron mit den Iris-Mustern trainiert. Während des Trainings haben wir dabei die im vorigen Abschnitt erwähnte Statistik ausgegeben:

```
Trainiere Perzeptron...
training epoch: 0 learning rate: 0.1 classification errors: 7
training epoch: 1 learning rate: 0.1 classification errors: 7
training epoch: 2 learning rate: 0.1 classification errors: 7
:
training epoch: 33 learning rate: 0.1 classification errors: 7
training epoch: 34 learning rate: 0.1 classification errors: 1
```

```
training epoch: 35 learning rate: 0.1 classification errors: 4
training epoch: 36 learning rate: 0.1 classification errors: 7
training epoch: 37 learning rate: 0.1 classification errors: 7
training epoch: 38 learning rate: 0.1 classification errors: 7
training epoch: 39 learning rate: 0.1 classification errors: 0
fertig

Anzahl der falsch klassifizierten Muster innerhalb der Trainingsmenge : 0
Gesamtzahl der durchgeführten Iterationen (Lernschritte) : 560
```

Eine Trainingsepoche bestand dabei aus einem einzelnen Durchlauf mit allen Trainingsmustern. Wie wir sehen, wurden insgesamt 40 Epochen ausgeführt, was bei der hier verfügbaren Anzahl von 14 Trainingsmustern insgesamt $40 \cdot 14 = 560$ Lernschritte des Perzeptron-Lernalgorithmus bedeutet. Anhand der Statistik ist zu erkennen, daß alle Muster der Trainingsmenge korrekt gelernt wurden. Abbildung 3, rechts, bestätigt dies, da die gelernte Hyperebene hier beide Klassen von Iris-Typen korrekt voneinander zu trennen scheint.

Als nächstes wollen wir uns die Performance des Netzes bezüglich einer separaten Testmenge (einer Menge, mit der das Perzeptron zuvor *nicht* trainiert worden ist) bestehend aus den 4 vorgegebenen Mustern unbekannter Klassenzugehörigkeit ansehen und dabei überprüfen, wie die zugehörige Klassifikationsentscheidung aussieht. Abbildung 4, links, zeigt die Irisdaten und die gelernte Hyperebene, zusammen mit den vier Punkten (gelb) der unbekannten Klassen. In Abbildung 4, rechts, wurden diese Punkte durch Symbole entsprechend der Klassifikationsentscheidung des Perzeptrons ersetzt. Die Symbole der übrigen Punkte wurden mit dem Wissen, zu welcher Klasse sie gehören, eingezeichnet.

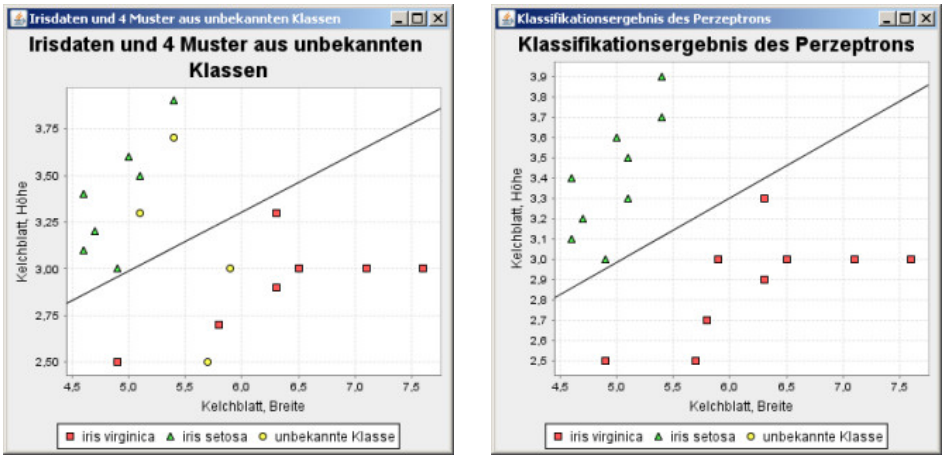

Abbildung 4: *Links:* Abgebildet sind die Irisdaten mit der gelernten Hyperebene. Die gelben Kreise stellen 4 Muster aus einer unbekannten Klasse dar, zu denen die Klassenzugehörigkeit mit Hilfe des Perzeptrons ermittelt werden soll. *Rechts:* Die Symbole der 4 Muster wurden hier entsprechend der Klassifikationsentscheidung des Perzeptrons eingezeichnet.

Es ist zu erkennen, daß die Klasseneinteilung des Perzeptrons mit der Klasseneinteilung der übrigen Punkte konsistent ist. Das Klassifikationsergebnis für diese Punkte haben wir durch unser Programm nocheinmal ausgegeben lassen:

```
Eingabemuster (5.1, 3.3) wurde als Mitglied der folgenden Klasse erkannt: ‘‘iris setosa‘‘
Eingabemuster (5.9, 3.0) wurde als Mitglied der folgenden Klasse erkannt: ‘‘iris virginica‘‘
Eingabemuster (5.7, 2.5) wurde als Mitglied der folgenden Klasse erkannt: ‘‘iris virginica‘‘
Eingabemuster (5.4, 3.7) wurde als Mitglied der folgenden Klasse erkannt: ‘‘iris setosa‘‘
```

Wir sehen, daß die 4 Punkte einerseits korrekt in den Plot eingezeichnet werden und unser Programm andererseits eine konsistente Ausgabe zu produzieren scheint. Abschließend wollen wir noch die vollständige Iristabelle angeben, die wir mit unserem Perzeptron ermittelt haben:

Kelchblatt Breite [cm]	Kelchblatt Höhe [cm]	Iris-Art	
4.9	3.0	iris setosa	
4.7	3.2	iris setosa	
5.1	3.5	iris setosa	
4.6	3.1	iris setosa	
5.0	3.6	iris setosa	
5.4	3.9	iris setosa	
4.6	3.4	iris setosa	
6.3	3.3	iris virginica	
5.8	2.7	iris virginica	
7.1	3.0	iris virginica	
6.3	2.9	iris virginica	
6.5	3.0	iris virginica	
7.6	3.0	iris virginica	
4.9	2.5	iris virginica	
5.1	3.3	iris setosa	(Klassifikation durch Perzeptron)
5.9	3.0	iris virginica	(Klassifikation durch Perzeptron)
5.7	2.5	iris virginica	(Klassifikation durch Perzeptron)
5.4	3.7	iris setosa	(Klassifikation durch Perzeptron)

Damit wurden alle in der Spezifikation genannten Anforderungen erfolgreich getestet und erfüllt.

A Zusatz – Nachweis der Fehlertoleranz

(Hinweis: Dies ist ein optionaler Teil der Dokumentation, den ihr nicht unbedingt erstellen müßt. Bei praktischen Implementierungen sollte man sich jedoch immer mit möglichen Fehlerquellen beschäftigen und versuchen, möglichst „Robuste“ Programme zu erzeugen.)

A.1 Test 1 – Training mit einer unterschiedlichen Anzahl von Eingabe- und Ausgabemustern

In diesem Abschnitt weisen wir die aus Abschnitt 2.4 erwähnte Fehlertoleranz unserer Implementierung nach (eine detailierte Beschreibung der im Folgenden verwendeten Funktionen findet sich in der Implementierungsdokumentation (Javadoc)). Wir erzeugen uns als erstes ein Perzeptron:

```
Perzeptron p = new Perzeptron();
```

Anschließend initialisieren wir es mit 5 zufälligen Gewichten aus dem Intervall $[-0.5, +0.5]$

```
p.initWeights(5,-0.5,+0.5);
```

und trainieren es mit einere Lernrate von 1.0 für maximal 100 Trainingsepochen mit den Eingabevektoren $[1,1,1,1,1]^T$, $[1,1,1,1,0]^T$ und gewünschten Ausgabewerten $1,0,1,0$:

```
p.train(new double[][]{{1,1,1,1,1},{1,1,1,1,0}}, new int[]{1,0,1,0}, 1, 0,
100, true);
```

Man beachte, daß wir mehr Ausgaben angegeben haben, als Eingabevektoren vorhanden sind. Wir erhalten die Ausgabe:

```
training epoch: 1 learning rate: 1.0 classification errors: 1
training epoch: 2 learning rate: 1.0 classification errors: 1
training epoch: 3 learning rate: 1.0 classification errors: 1
training epoch: 4 learning rate: 1.0 classification errors: 1
training epoch: 5 learning rate: 1.0 classification errors: 1
training epoch: 6 learning rate: 1.0 classification errors: 0
```

Nach dieser Ausgabe und unseren Vorgaben aus Abschnitt 2.4 müssen die Muster $[1,1,1,1,1]^T$, $[1,1,1,1,0]^T$ nun den Klassen $1,0$ zugeordnet sein. Überprüfen wir dies durch Berechnung der Perzeptron-Ausgabe bei Eingabe dieser Muster:

```
int klasseMuster1 = p.netOutput(new double[]{1,1,1,1,1});
int klasseMuster2 = p.netOutput(new double[]{1,1,1,1,0});
```

Dies liefert die Werte:

```
klasseMuster1 = 1
klasseMuster2 = 0
```

Die beiden Muster wurden also den korrekten Klassen zugeordnet und wir sehen, daß das Training des Perzeptrons auch trotz unterschiedlicher Anzahl von Eingabe- und Ausgabemustern erfolgreich ausgeführt werden kann. Dabei werden nur die Paare berücksichtigt, die auch tatsächlich vorhanden sind. (Wir haben zwar hier nur einen Test mit einer größeren Anzahl an Ausgabemustern vorgestellt, man erhält jedoch ein entsprechendes Ergebnis, wenn mehr Eingabe- als Ausgabemuster verwendet werden.)

A.2 Test 2 – Berechnung der Ausgabe mit zu langen/kurzen Eingabevektoren

Das Perzeptron des vorigen Tests konnte die beiden Muster $[1,1,1,1,1]^T$ bzw. $[1,1,1,1,0]^T$ den richtigen Klassen, 1 bzw. 0, zuordnen. Nach unseren Vorgaben aus Abschnitt 2.4 müssen die zusätzlichen Komponenten längerer Eingabevektoren ignoriert werden. Überprüfen wir dies mit dem trainierten Perzeptron des vorigen Abschnittes, indem wir die Ausgabe (die Klassifikationsentscheidung) des Perzeptrons bei Eingabe der Mustervektoren $[1,1,1,1,1,1,0,1]^T$ und $[1,1,1,1,0,1,1]^T$ berechnen:

```
int klasseMuster3 = p.netOutput(new double[]{1,1,1,1,1,1,0,1});
int klasseMuster4 = p.netOutput(new double[]{1,1,1,1,0,1,1});
```

Wir erhalten die Werte:

```
klasseMuster3 = 1
klasseMuster4 = 0
```

Offensichtlich ist dies korrekt, da die ersten 5 Komponenten der Eingabemuster, zu denen wir die Ausgabe berechnet haben, gerade die von unserem Perzeptron zuvor gelernten Trainingsmuster sind.

Testen wir nun noch, was bei kürzeren Eingaben passiert:

```
int klasseMuster5 = p.netOutput(new double[]{1,1});
int klasseMuster6 = p.netOutput(new double[]{1,1,1,1});
```

Die entsprechenden Werte lauten:

```
klasseMuster5 = 0
klasseMuster6 = 0
```

Die zweite Ausgabe scheint korrekt, da nach unseren Vorgaben kürzere Eingabevektoren mit Nullen aufgefüllt werden, was gerade das zweite gelernte Muster ergibt. Um die erste Ausgabe überprüfen zu können, lassen wir uns den Gewichtsvektor des Perzeptrons geben:

```
double[] weights = p.getWeights();
```

Eine Ausgabe dieses Wertes liefert:

```
weights = [0.18013586040720375, -0.19581888229578037, -0.29823423102352986,
0.11114593456594468, 0.3767485610094773]^T
```

Bei Eingabe von $[1, 1]^T$ besitzt das Perzeptron nach den Vorgaben aus Abschnitt 2.4 daher eine Aktivierung von $z = 0.18013586040720375 \cdot 1 - 0.19581888229578037 \cdot 1 \approx -0.016$ (die fehlenden Komponenten des Eingabevektors $[1, 1]^T$ werden ja als „0“ angenommen). Mit der Heaviside-Funktion als Ausgabefunktion des Perzeptrons wird bei Eingabe dieses negativen Wertes dann die Ausgabe 0 erzeugt, was konsistent ist mit unserer beobachteten Ausgabe.

Wir folgern, daß unsere Implementierung des Perzeptrons bei unseren praktischen Versuchen die unter Abschnitt 2.4 genannten fehlertoleranten Eigenschaften zu besitzen scheint.