fhwedel
UNIVERSITY OF APPLIED SCIENCES

Learning und Softcomputing

Konzeption und Implementierung eines künstlichen neuronalen Netzes

mit Backpropagation in Python

Eingereicht am:

4. Oktober 2015

Jens Begemann
Matr.-Nr.: Inf 101419
Pestalozzistrße 51
25421 Pinneberg
Phone: (0176) 237 346 21
E-Mail: inf101419@fh-wedel.de

Maurice Tollmien
Matr.-Nr.: Inf 101074
Feldstraße 143
22880 Wedel
Phone: (0176) 457 573 62
E-Mail: inf101074@fh-wedel.de

Inhaltsverzeichnis

1 Einleitung

Das Projekt *Learning und Softcomuting* setzt sich mit dem Thema *Lernversuche zur automatischen Zeichenerkennung mit einem Backpropagation-Netz* auseinander. Die Thematik der neuronalen Netze mit Backpropagation-Lern-Algorithmus soll dabei von den Teilnehmern durchdrungen und durch den Einsatz eines neuronalen Netzes umgesetzt werden. Ziel ist es, das Netzwerk so zu realisieren und zu trainieren, dass es die 26 Großbuchstaben des Alphabets erkennt.

Grundsätzliche handelt es sich bei diesem Projekt um ein Gebiet des maschinellen Lernens. Vereinfacht gesagt, ist es die Aufgabe eines Computerprogramms durch Erfahrung zu lernen und somit eine Aufgabe möglichst optimal zu lösen.

Anstelle eine fertigen Frameworks wie *PyBrain* [1] oder *scikitLearn* [2] fokussiert sich dieses Dokument auf die Konzeption und Implementierung eines eigenen neuronalen Netzes in Python.

[1] http://pybrain.org/
[2] http://scikit-learn.org/stable/

2 Thema und Zielsetzung

Die Aufgabenstellung fordert die Entwicklung eines neuronalen Netzes welches in der Lage ist die 26 Großbuchstaben (A..Z) des Alphabets zu erkennen. Ziel der Projektaufgabe ist, die Thematik der Neuronalen Netze mit Backpropagation Lernalgorithmus zu durchdringen, sich in ein Software-Framework zur Realisierung neuronaler Netze einzuarbeiten und dessen Software-Komponenten zur Lösung der gestellten Aufgabe einzusetzen.

Die Anforderungen an das Framework sind wie folgt:

- Modifikation der Parameter: Lernrate, Momentum und Anfangsbelegung der Gewichte
- Unterstützung für reine Feedforward-Netze
- Unterstützung von Backpropagation
- Konstruktionen mehrlagiger Netze mit mindestens 1 und 2 versteckten Schichten (hidden Layers) mit jeweils wählbarer Neuronenanzahlen
- Unterstützung für musterweises und epochenweises Lernen
- Identischer innerer Aufbau der Neuronen etwa mit sigmoider Aktivierungsfunktion

Anstelle ein bestehendes Framework zur Lösung der Aufgabe einzusetzen fiel die Wahl auf die Entwicklung eines eigenen neuronalen Netzes in Python. Die Zielsetzung war dabei, ein einfaches, leistungsfähiges Netzwerk zu entwickeln welches in der Lage ist die Aufgabe möglichst optimal zu bewältigen. Im Gegensatz zum Einsatz von größeren Frameworks wie *scikit-learn* oder vergleichbaren Frameworks entsteht so kein Funktionsoverhead und es wird die Thematik grundlegend selbst erarbeitet.

3 Aufbereitung der Eingabedaten

Zum Trainieren des Netzes stehen die Windows Schriftarten zur Verfügung. Der Test des Netzwerkes erfolgt im Anschluss mit ungesehenen Schriftarten sowie Handschriftproben der Studenten der Fachhochschule Wedel.

Das Datenmaterial zum Trainieren und Testen wird in *Patternfiles* zur Verfügung gestellt. In diesen Dateien sind die einzelnen Buchstabenformen in einer *14x14* Binärbild-Matrix abgespeichert. Für die Klassifikation wird ein Ausgabevektor mit 26 Dimensionen bereitgestellt. Die Datensätze sind bereits für ein optimales Lernverhalten normiert worden. Hierfür besteht der Wertebereich für das Eingabemuster aus *[-0.5; 0.5]* (Abbildung 3.1) und der des Ausgabemusters auf *[0.2; 0.8]* (Abbildung 3.2).

```
-0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50  0.50  0.50  0.50 -0.50 -0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50
-0.50 -0.50 -0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50
-0.50 -0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50  0.50 -0.50 -0.50
-0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50 -0.50 -0.50
-0.50 -0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50 -0.50 -0.50
-0.50  0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50 -0.50
-0.50  0.50  0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50 -0.50  0.50  0.50  0.50 -0.50
```

Abbildung 3.1: Eingabemuster mit dem Buchstaben A

```
0.80  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20
0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20  0.20
```

Abbildung 3.2: Ausgabemuster für den Buchstaben A

Die Patternfiles sind nach einem festen Schema (Tabelle 3.1) aufgebaut welches ein Parsen, in die vom neuronalen Netz benötigte Struktur, ermöglicht.

Datensätze	Zeile	Inhalt
14	1	Zeilen des Eingabemusters
14	2	Spalten des Eingabemusters
1	3	Zeilen des Ausgabemusters
26	4	Spalten des Ausgabemusters

Tabelle 3.1: Struktur der Patternfiles

Über eine einfache Funktion wird das Einlesen der Patternfiles in die für das neuronale Netz benötigte Struktur vorgenommen (Listing 3.1).

Listing 3.1: Parsen der Patternfiles

```
1  def parseTrainingData(filename):
2      ...
3      while sampleCount*(inputLines+1)+(inputLines+1) < len(fileArray):
4          singleSample = []
5          index = 0
6          for i in range(4+sampleCount*(inputLines+1),
7                         3+(sampleCount+1)*(inputLines+1)):
8              tmp = filter(bool, re.split(" +", fileArray[i].rstrip()))
9              singleSample.append(string2floatArray(tmp))
10             index = i
11         singleOutput = filter(bool, re.split(" +", fileArray[index+1].rstrip()))
12         samples.append(tuple([numpy.array(singleSample), singleOutput]))
13         sampleCount += 1
14
15     return samples
```

4 Aufbau eines künstlichen neuronalen Netzes

Im Nachfolgenden wird der allgemeine Aufbau eines einfachen Neuronalen Netzes in Python erklärt. Dazu wird ein Netzwerk ohne versteckte Schichten und drei Trainingswerten verwendet. Im Anschluss wird die Erweiterung des Netzwerkes mit mehreren versteckten Schichten erläutert. Weiterhin wird das eingesetzte Gradientenverfahren zur Reduzierung des Fehlers im Netz erläutert.

4.1 Allgemeine Struktur des neuronalen Netzes

Der grundlegende Aufbau des eingesetzten backpropagation Netzes soll im Folgenden an einem vereinfachten Netz ohne versteckte Schichten verdeutlicht werden. Zum Einsatz kommt neben *Python 2.7* die Erweiterung *NumPy* zur Unterstützung von mehrdimensionalen Arrays und Matrizen.

Für die Veranschaulichung soll die Ausgabe anhand der drei Eingabespalten vorhergesagt werden (Tabelle 4.1). Jede Spalte steht für einen Eingabeknoten des Netzwerkes mit jeweils vier Trainingsdaten.

Eingabe			**Ausgabe**
0	0	1	0
1	1	1	1
1	0	1	1
0	1	1	0

Tabelle 4.1: Ein- und Ausgabe für das Netz

Die eingesetzte Aktivierungsfunktion ist eine Sigmoid-Funktion (Listing 4.1) mit der sich ebenfalls die Ableitung bzw. Steigung des jeweiligen Wertes berechnen lässt.

Listing 4.1: Sigmoid-Funktion

```
1 def sigmoid(x, deriv = False):
2     if(deriv == True):
3         return x*(1-x)
4     return 1/(1+np.exp(-x))
```

Die Ein- und Ausgabedaten werden in NumPy Arrays X und Y hinterlegt in denen jede Reihe ein Trainingsdatensatz darstellt. Die Werte für die Gewichte *syn0* werden zufällig über einen Seed initialisiert (Listing 4.2). So lässt sich bei mehrfacher Ausführung ein deterministisches Verhalten produzieren um eine Vergleichbarkeit zu gewährleisten. Da bei diesem Beispiel nur zwei Schichten (Ein- und Ausgabe) zum Einsatz kommen wird nur eine Gewichtsmatrix benötigt um diese beiden Schichten miteinander zu verbinden. Die Dimension dieser Matrix wird von dem Verhältnis der Ein- und Ausgaben, also 3:1, festgelegt.

Listing 4.2: Initialisierung Gewichte

```
1 syn0 = 2*np.random.random((3, 1)) - 1
```

Der Lernprozess wird beispielhaft über zehntausend Durchläufe vorgenommen (Listing 4.3). Das Netz besteht aus den zwei Schichten *l0* und *l1*. Die Eingabedaten werden direkt an die erste Schicht *l0* übergeben. Dabei werden alle vier Trainingsdaten auf einmal verarbeitet (Full-Batch). In der zweiten Schicht *l1* wird versucht die Ausgabe in der ersten Iteration direkt vorherzusagen (Zeile 3). Dabei wird die Eingabematrix *l0* mit der Gewichtsmatrix *syn0* multipliziert. Das Ergebnis wird anschließend an die Sigmoid Funktion übergeben. Bei vier Trainingsdaten erhält man so vier Vorhersagen für die gesuchte Ausgangsbelegung. Jeden Ausgabe stellt dabei den Versuch des Netzes dar anhand der Eingabedaten eine korrekte Vorhersage zu treffen.

Listing 4.3: Lernprozess

```
1 for iter in xrange(10000):
2     l0 = X
3     l1 = sigmoid(np.dot(l0, syn0))
4     l1_error = y - l1
5     l1_delta = l1_error * sigmoid(l1, True)
6     syn0 += np.dot(l0.T, l1_delta)
```

In Zeile 4 lässt sich die Vorhersage mit dem tatsächlichen Ergebnis vergleichen und der Fehler berechnen. Im der nächsten Zeile wird unter Zuhilfenahme der Ableitung

der Sigmoid-Funktion (Abbildung 4.1) bestimmt, wie zuversichtlich das Netzwerk für diesen Wert ist. Das Ziel ist es dabei den Fehler für Werte zu reduzieren bei denen sich das Netz sehr sicher ist. Je geringer die Steigung in einem Punkt (grüne und violette Steigung), umso zuversichtlicher ist das Netz entweder eine richtige, oder falsche Aussage getroffen zu haben. Bei unsicheren Vorhersagen ist der Wert der Steigung am größten und dadurch werden diese Fehler für die nächste Iteration am stärksten berücksichtigt.

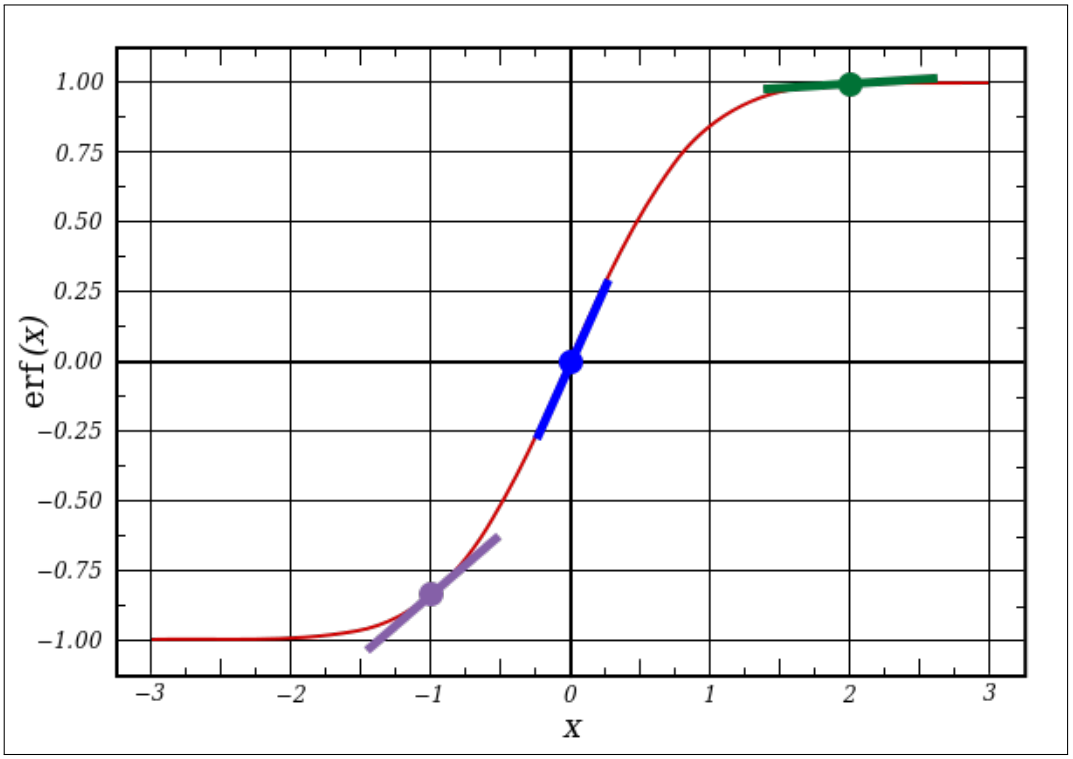

Abbildung 4.1: Ableitung Sigmoid-Funktion

Im letzten Schritt (Zeile 6) werden die Gewichte mit dem entsprechend berechneten Fehlern aktualisiert und der Prozess beginnt von neuem.

4.2 Gradientenverfahren

Zur Optimierung des neuronalen Netzes wird das Gradientenverfahren eingesetzt (Abbildung 4.2). Ziel dieses Verfahrens ist es, ein lokales Minimum zu finden. Dabei wird die Steigung an der aktuellen Position berechnet und anhand des Vorzeichens entschieden ob sich die nächste Position rechts oder links von der aktuellen Stelle befinden soll. Bei einem negativen Vorzeichen ist die Richtung rechts, bei einem positiven links. Dieser Vorgang wird solange wiederholt, bis die Steigung in einem Punkt den Wert *0*, oder ein bestimmten vorgegebenen Wert, erreicht.

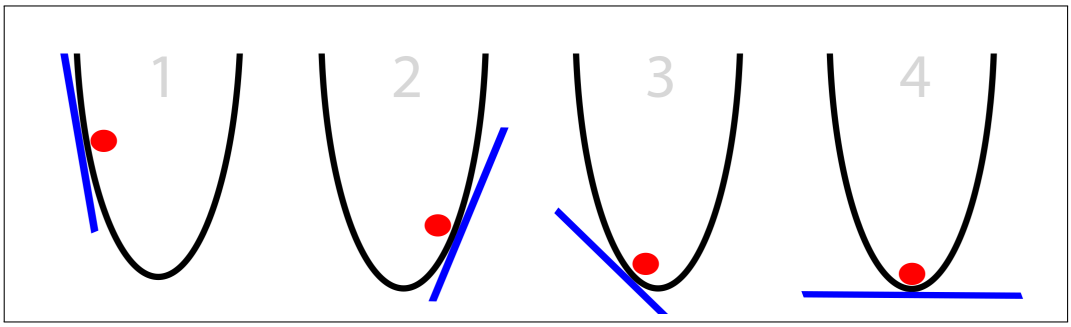

Abbildung 4.2: Gradientenverfahren

Die Schwierigkeit besteht darin, einen geeigneten Faktor zu bestimmen um den die Position geändert wird. Ein einfacher Ansatz ist dabei, die Verschiebung anhand der Steigung zu bestimmen. Je größer der Wert einer positiven Steigung, desto weiter wird die Position nach links verschoben. Je kleiner der Wert umso geringer fällt die Richtungsänderung aus. So werden Änderungen geringer bei einer Annäherung an ein lokales Minimum. Sollte der Nullpunkt erreicht werden, konvergiert das Verfahren. Bei zu großen Richtungsänderungen kann das Verfahren divergieren und man entfernt sich immer weiter von dem gesuchten Minimum.

Um dies zu verhindern wird ein *Alpha* Wert eingeführt. Der Wertebereich von *Alpha* liegt zwischen 0 und 1. Die neue Position wird nun durch den alten Positionswert und dem Produkt aus Steigung und Alphawert ermittelt. Bei der Wahl eines geeigneten Alphawertes muss berücksichtigt werden, dass ein zu kleiner Wert das Gradientenverfahren ausbremst und ein zu großer Wert den Einfluss auf das Ergebniss minimiert oder das Verfahren divergieren lässt.

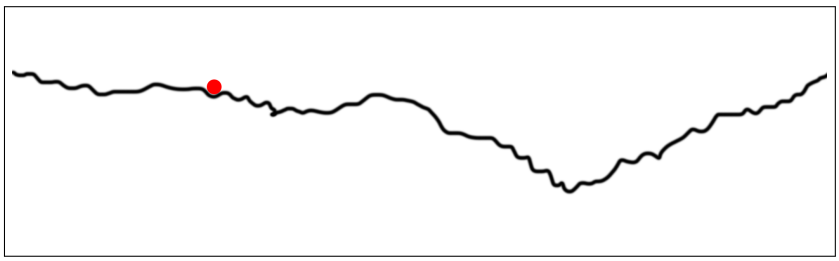

Abbildung 4.3: Alphawert Einfluss (Teil 1)

Die Wahl eines zu geringen Alphawertes kann zu zwei Problemen führen. Zum einen kann das Verfahren sich in lokalen Minima festsetzen (Abbildung 4.3), sodass das eigentliche Minimum des zu untersuchenden Abschnittes nicht gefunden wird. Im zweiten Fall (Abbildung 4.4) fallen die Änderungen bedingt durch ein zu kleines *Alpha* so gering aus, dass sich die Laufzeit zum Auffinden des Minimums enorm steigert.

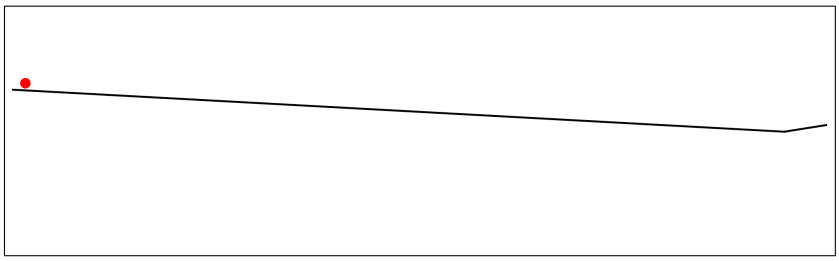

Abbildung 4.4: Alphawert Einfluss (Teil 2)

4.3 Erweiterung des Netzwerkes

Durch eine Erweiterung des Netzwerkes mit mehreren versteckten Schichten und Neuronen steigt die Leistungsfähigkeit, wodurch eine Kapselung von Funktionen notwendig wird um die Übersicht und Einfachheit weiterhin zu gewährleisten. Im Folgenden werden die einzelnen Funktionen zur Inbetriebnahme des Netzes erläutert.

Das Netzwerk lässt sich durch Parameterübergabe an die Initialisierungsfunktion (Listing 4.4) mit einer beliebigen Anzahl von versteckten Schichten und Neuronen erstellen. Die Ein- und Ausgabeschicht wird dabei auf die entsprechenden Größen der Trainings- und Ausgabedaten angepasst. Die Gewichte werden mit zufälligen Werten vorbesetzt, wobei immer derselbe Seed zum Einsatz kommt (Zeile 2) um eine Vergleichbarkeit zu erreichen, die zwischen verschiedenen Durchgängen nur von der Anzahl der Schichten und Neuronen abhängig ist.

Listing 4.4: Initialisierung des Netzes

```
1 def initNetwork(inputArraySize, outputArraySize, hiddenLayerCount, hiddenLayerSize):
2     np.random.seed(1)
3     syns = []
4     syns.append(2*np.random.random((inputArraySize, hiddenLayerSize)) - 1)
5     for i in range(hiddenLayerCount):
6         syns.append(2*np.random.random((hiddenLayerSize,hiddenLayerSize)) - 1)
7     syns.append(2*np.random.random((hiddenLayerSize,outputArraySize)) - 1)
8     return syns
```

Nachdem das Netzwerk durch die Init-Funktion erstellt wurde, kann der Trainingsvorgang begonnen (Listing 4.5) werden. Dafür werden neben den Trainingsdaten (`inputData` und `outputData`) Vorgaben für den Fehlerwert und die Rundenzahl benötigt. Durch diese beiden Parameter lassen sich die Abbruchbedingungen des Netzes festlegen. Diese ist entweder erreicht wenn die maximale Anzahl an Trainingsrunden erreicht wurde oder der Fehlerwert das vorgegebene Minimum erreicht hat (Zeile 3). Eine Reduzierung des Fehlers wird über das beschriebene Gradientenverfahren (Kapitel 4.2) zusammen mit dem Alphawert erreicht. Nach der Bestimmung des aktuellen Fehlers über die Sigmoid Funktion (Zeile 11) werden im Anschluss die Gewichte aus dem Produkt des Deltas und dem Alphawert angepasst (Zeile 18).

Listing 4.5: Trainieren des Netzes

```
def train(inputData, expectedData, syns, maxError, maxRounds, alpha, printError):
    ...
    while (roundCount < 1 or np.mean(np.abs(l_errors[dataSize-1])) > maxError)
           and roundCount < maxRounds:
        ls = []
        ls.append(inputData)
        for atLayer in xrange(dataSize):
            ls.append(sigmoid(np.dot(ls[len(ls)-1], syns[atLayer])))

        l_errors[dataSize-1] = ls[dataSize] - expectedData
        l_deltas[dataSize-1] =
            l_errors[dataSize-1] * sigmoid(ls[dataSize], deriv=True)

        for i in xrange(dataSize-2, -1, -1):
            l_errors[i] = l_deltas[i+1].dot(syns[i+1].T)
            l_deltas[i] = l_errors[i] * sigmoid(ls[i+1], deriv=True)

        for i in xrange(dataSize-1, -1, -1):
            syns[i] -= alpha * np.dot(ls[i].T, l_deltas[i])

        roundCount += 1
    ...
    return syns
```

Um das Netzwerk mit Eingaben zu trainieren und anschließend die Handschriftproben zu klassifizieren wurde eine parametrisierte Funktion erstellt (Listing 4.6). Diese Funktion unterstützt neben dem Epochenlernen auch die Verarbeitung von mini-Batches. Dabei lassen sich die Trainingsdaten in kleinere Datensätze aufteilen und das Netz so anstelle der kompletten Trainingsdaten, stückweise trainieren. Bei größeren Trainingssätzen ist so eine schnellere Abschätzung bezüglich der eingestellten Parameter möglich.

Listing 4.6: Trainieren und testen auf Handschriften

```
def trainAndPredictParametrized(epochCount, batchSize, inputCount, X, y,
                                testData, testTargets, syns, maxErrorRate,
                                maxRounds, alpha, debugPrints):

    for epoch in range(epochCount):
        for i in range(inputCount/batchSize):
            syns = train(np.array(X[i:i+batchSize]),  np.array(y[i:i+batchSize]),
                         syns, maxErrorRate, maxRounds, alpha, debugPrints)
        correctPredicted = testAllData(testData, testTargets, syns, False)
        if debugPrints:
            print "correct predicted:", correctPredicted, " --> ",
                  str(100*correctPredicted/len(testData)) + "%"
    return correctPredicted
```

5 Ergebnisse

5.1 Bewertung des Netzes

Nach wenigen Testläufen haben sich sehr schnell die optimalen Einstellungen für das Netzwerk festlegen lassen (Tabelle 5.1).

Verst. Schichten	Neuronen	Alpha	Batchsize	Epochen	Durchläufe
2	25	0.02	260	1	30.000

Tabelle 5.1: Initialwerte des neuronalen Netzes

Die Erfolgsquote des Netzwerkes liegt ungefähr im selben Bereich wie die der eingesetzten Frameworks von allen teilgenommen Studentengruppen aus dem Sommersemester 2015 (Tabelle 5.2).

Windows Schriftart	Handschrift alt	Handschrift SS2015
80%	50%	14%

Tabelle 5.2: Erkennungsrate mit unterschiedlichen Schriftsätzen

Auffällig bei den Ergebnissen, ist das schlechte Abschneiden des neuronalen Netzes bei den Handschriftproben aus dem aktuellen Sommersemester 2015. Eine detaillierte Betrachtung der Ergebnisse zeigt eine häufige nicht-Erkennung von bestimmten Buchstaben (Abbildung 5.1).

Dieses Ergebnis wurde von den teilnehmenden Gruppen in ihren Auswertungen bestätigt und lässt sich auf mangelhafte Datensätze zurückführen. Beim Vergleich von zuverlässig erkannten und gar nicht erkannten Buchstaben wird die Ursache schnell deutlich (Abbildung 5.2). Die obersten beiden Zeilen entsprechen Buchstaben aus dem aktuellen Handschriftprobensatz die nicht erkannt wurden. Diese Datensätze sollten im Vorfeld

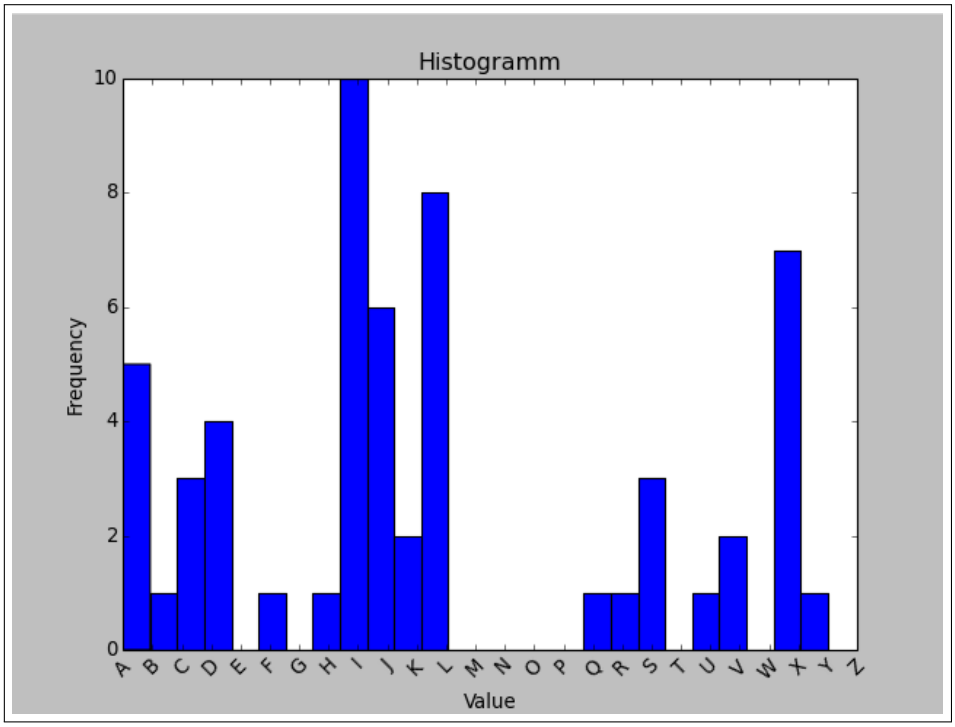

Abbildung 5.1: Erkennungsraten

als fehlerhaft aussortiert werden. Die unteren beiden Zeilen stellen hingegen positiv-Beispiele für gute Datensätze dar, die von den neuronalen Netzen in jedem Durchlauf auch zuverlässig erkannt werden können.

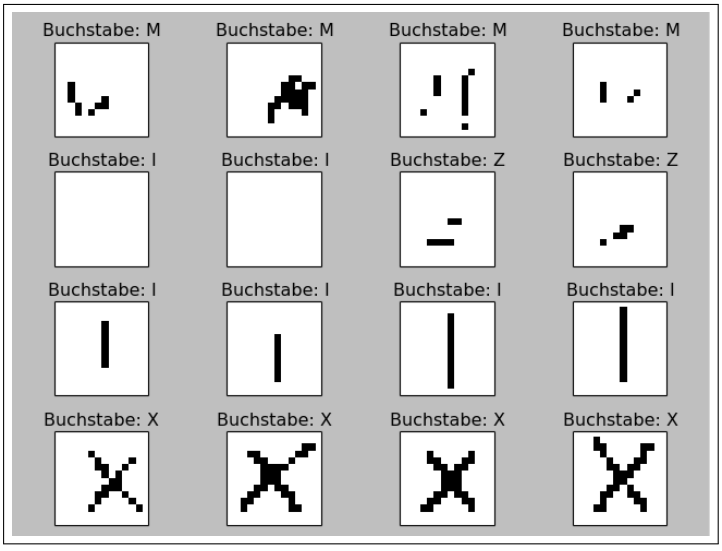

Abbildung 5.2: Gute und schlechte Handschriftproben

5.2 Fazit

Durch die Implementierung eines eigenen neuronalen Netzes konnten die erlernten Grundlagen optimal eingesetzt werden. Das Netzwerk lässt sich durch die gute Übersichtlichkeit, bedingt durch den schlanken Code, schnell anpassen. Änderungen und Erweiterungen lassen sich folglich einfach umsetzen. Im direkten Vergleich zu bestehenden Frameworks ist bei der eigenen Implementierung kein Funktionsoverhead entstanden, wodurch auch die Geschwindigkeit des Netzwerkes profitieren konnte.

Abschließend lässt sich feststellen, dass sich durch die Konzeption und Implementierung eines eigenen neuronalen Netzes die Anforderungen (Kapitel 2) optimal umsetzen ließen. Die Ergebnisse der Handschrifterkennung liegen auf dem selben Niveau der eingesetzten Framworks wie z. B. *PyBrain* [1] von den teilgenommen Studentengruppen.

[1] http://pybrain.org/

6 Mögliche Weiterentwicklungen

Bei der Entwicklung des neuronalen Netzes lag der Fokus auf der eigenständigen Erarbeitung von Grundlagen im Bereich des maschinellen Lernens. Durch den Ausbau des Netzes lässt sich dessen Funktionsumfang noch deutlich erweitern. Mögliche Ansatzpunkte für Erweiterungen können dabei aus den folgenden Bereichen gefunden werden

- Paralellisierung
- GPU-Computing
- Hour-Glass (Unterschiedliche Größen der versteckten Schichten)
- Dropout Verfahren (Zur effizienten Suche von Minima beim Gradientenvefahren)
- Simulated Annealing
- Einsatz verschiedener Aktivierungsfunktionen

Weiterhin ist die Entwicklung eines Statistik-Moduls sinnvoll um die Ergebnisse aus den Lernvorgängen durch automatisch generierte Plots und Datensätzen zu ermöglichen.

Abbildungsverzeichnis

Listingverzeichnis