Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

Handschrifterkennung mittels Multilayer Perceptron und Bagging auf dem Android Betriebssystem

Kai Groetenhardt

Studiengang: Master Informatik

14. Juli 2012

Abstract — *Inhalt dieser Arbeit ist die Erläuterung und Implementierung einer Handschrifterkennung auf dem Android Betriebssystem. Zur Erkennung der Handschrift werden zwei verschiedene Machine Learning Verfahren getestet - das Multilayer Perceptron und der Bagging Algorithmus. Zudem wird die Theorie hinter den Algorithmen erläutert. Ebenfalls behandelt wird das Erfassen und Normalisieren der Eingabedaten, welche nativ als Bildschirmkoordinaten eingespeist werden. Abschließend wird die Effizienz der benutzen Algorithmen evaluiert und ein Fazit gezogen, wobei herausgekommen ist, dass das Multilayer Perceptron gegenüber des Baggings wesentlich bessere Ergebnisse erzielt.*

1 Einleitung

Mobiltelefone mit Touchscreen erlangen in der heutigen Zeit immer größere Beliebtheit. Insbesondere Smartphones mit dem Android Betriebssystem. Auf diesen Geräten werden Eingaben hauptsächlich mit der Software Tastatur getätigt. Allerdings bietet sich der Touchscreen auch an Eingaben über Gesten zu machen. Gesten wie nach links oder rechts über das Display wischen sind allgemein bekannt, um in diversen Apps vor oder zurück zu gelangen, oder um in andere Tabs zu wechseln (wie zum Beispiel in Internet Browsern). Die Intuition dieser Gesten legt die Überlegung nahe auch komplexere Gesten zu verwenden, wie das Zeichnen von Ziffern und Buchstaben, um so eine Alternative zur Software Tastatur bereitzustellen.

Das Problem dabei ist, dass das Smartphone die Gesten erkennen können muss. Bei einfachen Gesten wie das Wischen von links nach rechts sind sehr schnell erkannt, hier muss prinzipiell nur geschaut werden, an welcher Stelle der Finger das Display berührt, an welcher Stelle er das Display wieder los lässt, wie lange die Strecke zwischen den beiden Punkten ist und welche Strecke tatsächlich zurückgelegt wurde. Sind die beiden Strecken in etwa gleich lang kann man von einer Wisch-Geste ausgehen.

Mit komplexeren Gesten, wie das Zeichnen von Ziffern, bietet es sich an auf Techniken des Machine Learnings zurückzugreifen. Diese Techniken sind dazu ausgelegt “ungenaue“ Eingaben zu klassifizieren. In dieser Arbeit werden das Multilayer Perzeptron, sowie das Bagging-Verfahren mit dem einfachen Perzeptron als Grundlage zur Lösung des Problems verwendet.

Das Ziel dieser Arbeit ist das Beschreiben und Implementieren der eben erwähnten Algorithmen. Zuerst wird allerdings auf ähnlich gelagerte Projekte eingegangen. Anschließend werden die Grundlagen erläutert, die zur Fertigstellung

dieser Arbeit benötigt werden. Darunter befindet sich die Beschreibung des Verfahrens zur Normalisierung der Eingabedaten, sowie der verwendeten Machine Leraning Algorithmen Multilayer Perceptron und Bagging. Nach den Grundlagen wird auf die Android Anwendung und ihren Aufbau eingegangen. Abschließend wird die Effizienz der Algorithmen evaluiert und ein Fazit vorgestellt.

2 Verwandte Arbeiten

Die folgende Arbeit beschäftigt sich mit der offline Handschrifterkennung mit Hilfe von multidimensionalen rekurrenten neuronalen Netzen. Als Eingabe werden dort Bilddaten als Input verwendet. Hierbei werden auch die Größe und Rotation der Buchstaben zur Auswertung berücksichtigt. (AG09)

Diese Arbeit der Universität Bern beschäftigt sich mit der Kombination von zwei Arten der Hidden Markov Models und rekurrenten neuronalen Netzen, um die Erkennungs-Performance zu erhöhen. (VF09)

3 Grundlagen

In diesem Abschnitt werden die Grundlagen erläutert, die für diese Arbeit benötigt werden.

3.1 Machine Learning

In Machine Learning geht es darum Computer dazu zu bringen ihre Aktionen (Aktionen im Sinne von Vorhersagen oder Kontrolle) so anzupassen, dass diese genauer werden. Die Genauigkeit bezieht sich hierbei darauf wie sehr die Aktionen der Computer von den richtigen Aktionen abweichen. Zum Beispiel könnte man sich vorstellen ein Spiel wie Scrabble gegen einen Computer zu spielen. Am Anfang würde man immer gegen den Computer gewinnen, bis er nach einigen Spielen anfängt zu gewinnen. Entweder wird der Spieler schlechter, oder der Computer lernt, wie er ihn besiegen kann. Diesen Lernfortschritt könnte der Computer nicht nur gegen diesen Spieler einsetzen, sondern gegen jeden beliebigen Spieler, das ist eine Form der Generalisierung.

Machine Learning kombiniert die Ideen der Neurowissenschaften, Biologie, Statistik, Mathematik und Physik um Computer lernen zu lassen. Der beweis, dass es überhaupt möglich ist zu lernen ist wohl offensichtlich das Gehirn. Daher ist es naheliegend Methoden zu nutzen, die das Gehirn auch verwendet, um Machine Learning Algorithmen zu entwickeln.

3.1.1 Typen des Machine Learnings

Das Lernen kann auch als “durch Übung besser werden“ bezeichnet werden. Dies führt zu der Frage wie ein Computer herausfindet, ob er besser oder schlechter wird und wie er herausfindet sich zu verbessern. Dazu gibt es mehrere Antworten, welche zu verschiedenen Typen des Machine Learnings führen. Es ist möglich dem Algorithmus einige richtige Antworten mitzuteilen, sodass er herausfindet wie das Problem zu lösen ist. Diese Lösung soll auch für noch unbekannte Daten verwendet werden können, also das Problem generalisieren. Alternativ könnte dem Algorithmus auch einfach mitgeteilt werden ob seine Lösung richtig oder falsch ist, aber nicht wie er die richtige Antwort findet, sodass er diese selbst suchen muss. Eine Variante dieser beiden Methoden wäre es dem Algorithmus Punkte auf seine Antwort zu geben, welche aussagen wie gut seine Antwort war. Auch möglich wäre es dem Algorithmus gar keine Informationen darüber zu geben welches richtige oder falsche Antworten sind, sondern ihn einfach nur Gemeinsamkeiten der Daten finden lassen.

Daraus ergeben sich folgende Klassen von Algorithmen:

- Überwachtes Lernen (Supervised learning): Hierbei wird dem Algorithmus zum Lernen ein Training-Set mit Beispieldaten und des dazugehörigen Antworten übergeben. Basierend auf diesen Trainingsdaten wird eine Generalisierung durchgeführt. Es wird auch als Lernen durch Beispiele bezeichnet.
- Unüberwachtes Lernen (Unsupervised learning): In diesem Fall werden keine

Richtigen Antworten zum Training bereitgestellt, sondern es wird versucht Ähnlichkeiten zwischen den Daten herauszufinden und diese in Kategorien einzuteilen.

- Vertärkendes Lernen (Reinforcement learning): Dies ist eine Mischung der vorherigen beiden Algorithmen. Dem Algorithmus wird mitgeteilt ob eine Antwort falsch ist, nicht aber was die richtige Antwort ist.
- Evolutionäres Lernen (Evolutionary learning): Biologische Evolution kann als Lernprozess gesehen werden: Biologische Organismen passen sich an um ihre Überlebenschancen zu verbessern. Dazu wird versucht ein Modell auf dem Computer zu modelieren, welches das Prinzip der Fitness nutzt, was dazu führt der aktuellen Lösung zu bewerten und dementsprechend anzupassen.

Die Methode die meistens verwendet wird ist das Überwachte Lernen, welches auch in dieser Arbeit zum Einsatz kommt.

3.2 Normalisierung via $1.5 Recognizer

Während des Zeichnens wird der Pfad des Fingers in Form von einzelnen Positionen auf dem Display abgebildet. Hierbei kann sich die Anzahl der verschiedenen Positionen von Zeichen zu Zeichen unterscheiden. Allerdings ist die Grundbedingung bei den verwendeten Machine Learning Algorithmen, dass die Eingabedaten immer die gleiche Länge haben müssen. Das heißt, dass jedes Zeichen aus immer gleich vielen Punkten bestehen muss. Zudem sollte dafür gesorgt werden, dass alle Zeichen immer gleich groß sind, um die Abweichungen zwischen den Zeichen zu minimieren.

Um dies zu gewährleisten, muss der Eingabepfad normalisiert werden. Hierfür bietet sich der $1.5 Recognizer Algorithmus an (AS11). Dieser Algorithmus ist eine Verbesserung des $1 Recognizer (WWL07), welcher die Anzahl der Punkte in einem Pfad auf einen einstellbaren Wert bringt (auch Resampling genannt), sowie die Punkte rotiert und skaliert. Das Skalieren der Punkte ist insbesondere deshalb wichtig, da die Eingabedaten zwischen -1 und +1 liegen müssen. Das Rotieren wird in dieser Arbeit nicht verwendet, da die Zeichen in der vom Benutzer gegebenen Rotation bleiben sollen, ansonsten würde es Probleme mit den Ziffern 6 und 9 geben.

Um das Prinzip des Resamplings zu verdeutlichen, sind hier in Abbildung 1 und 2 die Eingaben des Benutzers und das Ergebnis des $1.5 Recognizers mit einer Zielpfadlänge von 16 zu sehen.

Abbildung 1: Eingabedaten des Benutzers

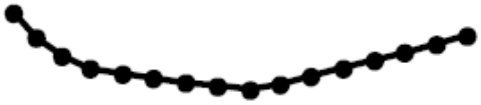

Abbildung 2: Ausgabe des $1.5 Recognizers

Der Resampling Algorithmus des $1.5 Recognizers ist in Algorithmus 1 zu sehen und funktioniert wie folgt: Es wird eine Liste von Punkten übergeben, welche den Zeichenpfad wiedergeben, sowie die Anzahl der Punkte auf den der Pfad reduziert/erhöht werden soll. Zum Start wird bestimmt in welchem Abstand die Punkte gesetzt werden müssen, sodass die alte Pfadlänge sowie die neue Punktmenge erreicht werden. Dazu wird die Pfadlänge berechnet und durch die Anzahl der gewünschten Punkte geteilt (zu sehen in Zeile 1). Solange die Zielpunktliste nicht die übergebene Länge erreicht hat, wird der ursprüngliche Pfad “abgefahren“. Immer wenn die am Anfang berechnete Zieldistanz zwischen den Punkten erreicht wurde, wird ein neuer Punkt in die Zielliste eingefügt. Ist die Zielmenge erreicht, wird die Liste zurückgegeben.

Die Algorithmen zum Skalieren und Verschieben der Punkte sind trivial und werden daher hier nicht aufgeführt, sie können allerdings in der Literatur (WWL07) nachgelesen werden.

Code 1 Anzahl der Pfadpunkte anpassen

```
Funktion: Resample(points, sampleRate)
I ← PathLength(points)/(sampleRate − 1)
rest ← 0
oldI ← 1
nPoints ← Length(points)
Insert(newPoints, 0, points[0])
while Length(newPoints) < sampleRate do
    d ← Distance(points[oldI − 1], points[oldI])
    if d + rest ≥ I then
        if rest == 0 then
            m ← I
        else
            m ← abs(I − rest)
        end if
        r ← (points[oldI] − points[oldI − 1])/d × m
        q ← points[oldI − 1] + r
        Append(newPoints, q)
        Replace(points, oldI − 1, q)
        rest ← 0
    else
        if oldI < nPoints − 1 then
            rest ← rest + d
            oldI ← oldI + 1
        else
            Append(newPoints, points[nPoints − 1])
        end if
    end if
end while
return newPoints
```

3.3 Normalisierung via Bitmap

Eine alternative Weise normalisierte Daten zu erhalten, wäre das Zeichen auf ein (Schwarzweiß-) Bild mit einer bestimmten Größe abzubilden. Da die Bilder immer gleich groß wären, wären die Eingabedaten folglich auch immer von der selben Länge. Der Nachteil hierbei ist, dass die Datenmenge unnötig groß ist. Schon bei einem Bild von 16x16 Pixeln besteht ein Zeichen schon aus 256 Eingabedaten und ist dennoch relativ ungenau hinterlegt. Der Vorteil wäre allerdings im Gegensatz zur Methode vom $1.5 Rcognizer, dass dort auch das Absetzen des Finger beim Zeichnen besser ermöglicht werden könnte.

3.4 Perceptron

Da das Perceptron als Grundlage für den Bagging Algorithmus verwendet wird und es das Verstehen des Multilayer Perceptrons erleichtert, wird hier vorerst das Perceptron erläutert.

Ein Perceptron ist an die Neuronen des Menschlichen Gehirns angelegt, ganz grob gesagt bekommen sie Daten als Input, verarbeiten diese und geben Werte zurück. Ein Perceptron kann beliebig viele Inputs haben, gibt aber immer nur ein Ausgabewert zurück. Ein Netzwerk aus Perceptrons ist in Abbildung 3 zu sehen. Die grauen Punkte stellen die Eingabewerte dar, die schwarzen Punkte sind die Perceptrons mit den Aktivierungsfunktionen rechts daneben. Zudem ist ganz oben ein weiterer Punkt abgebildet welcher eine weitere Eingabe darstellt, das sogenannte Bias. Alle Eingaben sind mit allen Perceptrons mittels einer Linie verbunden. Auf den Linien kann man sich die sogenannten Gewichte vorstellen, welches Werte sind, die mit den Eingabewerten multipliziert werden. Die Gewichte sind die Daten, die verändert werden, um das Perceptron zu trainieren. Das Ändern der Gewichte könnte man auch als das Lernen bezeichnen. Das Bias wird in der Regel mit -1 belegt, um zu ermöglichen, dass ein anderer Wert als 0 ausgegeben werden kann, auch wenn alle Eingabewerte 0 betragen.

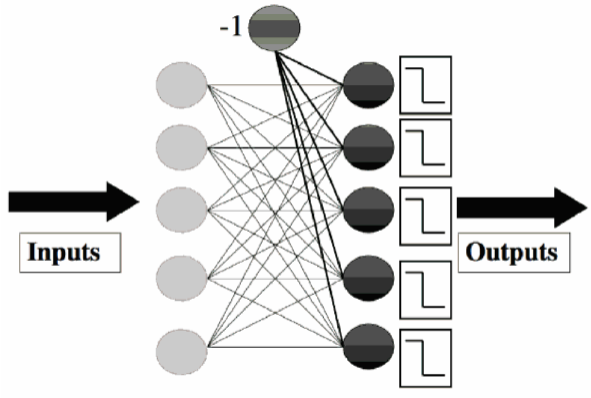

Abbildung 3: Ein Perceptron Netzwerk mit Bias Input

Nun wird grob den Algorithmus des Perceptrons erläutert. Die Gewichte des Perceptrons werden zum Start mit zufälligen Werten zwischen -1 und +1 belegt. Nun wird das Perceptron mit den Trainingsdaten trainiert, das heißt jeder Trainingsdatensatz wird mit den Gewichten multipliziert. Danach wird der Ausgabewert mit dem Zielwert verglichen. Gibt es zwischen Ausgabe-

und Zielwert eine Abweichung werden die Gewichte angepasst, indem sie erhöht oder verringert werden.

Das Perceptron kann allerdings nur linear separierbare Probleme lösen, d.h. nur Klassen von einander trennen, die durch je nach Dimension der Eingabedaten eine Gerade, eine Ebene oder eine Hyperebene separiert werden können. Dazu zählt zum Beispiel das logische AND nicht aber das XOR.

Weitere Details über das Perceptron kann der Quelle Mar09 entnommen werden.

3.5 Multilayer Perceptron

Ein Multilayer Perceptron (kurz MLP) ist ein Machine Learning Verfahren mit dem es möglich ist Daten zu klassifizieren. Das MLP gehört zu den überwachten Lernen, was bedeutet, dass es vorerst mit Daten trainiert wird, deren Klassen bekannt sind.

Vorerst allerdings zur Idee hinter dem MLP: Da die meisten interessanten Probleme nicht linear separierbar sind und sich daher nicht mit einfachen Perceptrons lösen lassen, sollten komplexere Netzwerke verwendet werden. Es wurde gezeigt, dass das Lernen in den Gewichten geschieht, daher ist es naheliegend eine weitere Schicht von Gewichten in das Perceptron einzubauen und somit die Komplexität zu erhöhen. Wie in Abbildung 4 zu sehen ist, wird beim MLP mindestens eine Schicht (die sogenannte versteckte Schicht oder Hidden Layer) zwischen der Input- und Output-Schicht eingefügt. Auch im MLP wird an jeder Schicht ein Bias mit dem Wert -1 als zusätzlicher Eingabewert übergeben.

3.5.1 Going Forwards

Wie im Perceptron auch, besteht das Trainieren aus zwei Teilen. Das Berechnen der Ausgaben zu den entsprechenden Inputs und das Anpassen der Gewichte in Abhängigkeit des Fehlers, welcher sich aus der Differenz zwischen den Ausgabewerten und den Zielwerten berechnen lässt. Diese beiden Teile sind bekannt unter vorwärts (Going Forwards) und zurück (Going Backwards) durch das Netzwerk gehen. In Abbildung 4 ist zu sehen, dass links mit den Inputs gestartet wird, die dem Netz übergeben werden. Diese werden mit den ersten Gewichten multipliziert und den Aktivierungsfunktionen übergeben. Die Ausgaben der Aktivierungsfunktion werden wiederum als Eingabewerte für die nächste Schicht (dem Output Layer) verwendet, welche mit deren Gewichten multipliziert werden und nach der Aktivierungsfunktion als schlussendliche Ausgabewerte zurückgegeben werden. Da nun die Ausgabewerte berechnet wurden, können sie mit den Zielwerten verglichen werden, um so den Fehler zu berechnen.

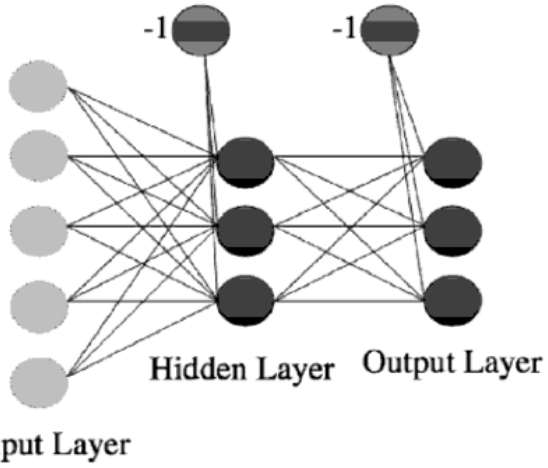

Abbildung 4: Ein Multilayer Perceptron Netzwerk

3.5.2 Going Backwards

Im Backwards-Teil des Algorithmus wird es etwas schwieriger. Die Fehler am Ausgang zu berechnen, ist nicht schwieriger als beim Perceptron, allerdings ist es schwieriger herauszufinden, was man damit machen soll. Dazu wird die sogenannte Methode des “back-propagation of error“ im späteren Verlauf erläutert. Bei dem einfachen Perceptron wurden die Gewichte so verändert, dass die Aktivierungsfunktionen entsprechend der Zielwerte feuerten. Die Veränderungen der Gewichte wurden anhand des Fehlers vollzogen, welcher durch die Anpassungen versucht wurde so klein wie möglich zu machen. Da

das Perceptron nur eine Schicht hat, war dies ausreichend, um das Perceptron zu trainieren.

Prinzipiell soll beim MLP noch immer das gleiche gemacht werden - den Fehler minimieren. Allerdings ist dies durch die weiteren Schichten schwieriger zu bewerkstelligen. Das Problem ist es herauszufinden welches Gewicht welcher Schicht den Fehler verursacht hat, der beim Ausgang berechnet wurde.

Die Fehlerfunktion, die beim Perceptron verwendet wurde, ist folgende $E = t - y$, wobei t die Targets und y die bisherigen Outputs sind. Angenommen es werden zwei Fehler gemacht, beim ersten ist das Target größer als der Output und beim Zweiten ist der Output größer als das Target. Wenn diese beiden Fehler die gleiche Größe hätten und sie zusammengerechnet werden, könnte als Gesamtfehler 0 raus kommen, was bedeutet, dass kein Fehler gemacht wurde. Um dies zu verhindern, sollten alle Fehler auf das selbe Vorzeichen gebracht werden. In diesem Fall wird die sum-of-squares Fehlerfunktion verwendet, welche die Differenz zwischen Targets und Outputs berechnet, dies quadriert und addiert:

$$E(t, y) = \frac{1}{2} \sum_{k=1}^{n} (t_k - y_k)^2 \tag{1}$$

Die $\frac{1}{2}$ am Anfang der Funktion dient dazu diese besser ableiten zu können. Wenn eine Funktion abgeleitet wird, teilt sie den Gradienten mit, welcher die Richtung angibt in welche Richtung sie am stärksten steigt oder sinkt. Das heißt, dass, wenn eine Fehlerfunktion abgeleitet wird, der Gradienten des Fehlers erhalten wird. Da es beim Lernen darum geht den Fehler zu minimieren, ist es nötig dem Fehler "bergab" also in Richtung des negativen Gradienten zu folgen. Da die Gewichte während des laufenden Algorithmus verändert werden, sollten sie bei der Ableitung ebenfalls berücksichtigt werden.

Bei dem Thema Ableiten sollte noch ein Problem bei der Aktivierungsfunktion erwähnt werden. Bisher gibt sie entweder 0 oder 1 zurück, was als Graph vorgestellt einen plötzlichen Sprung von 0 auf 1 bedeutet. Das Problem dabei ist, dass solch eine Funktion nicht abgeleitet werden kann. Um das Problem zu lösen, sollte eine Funktion gefunden werden, welche der bisherigen Aktivierungsfunktion recht ähnlich und ableitbar ist. Die Funktion, die dafür gut geeignet ist, ist die sogenannte sigmoid Funktion:

$$g(h) = \frac{1}{1 + exp(-\beta h)} \tag{2}$$

Nun gibt es eine neue Fehlerberechnung und eine neue Aktivierungsfunktion. Diese kann differenziert werden, sodass die Gewichte, wenn sie geändert werden, in die Richtung geändert werden, sodass der Fehler kleiner wird. Was nun getan werden muss, ist den Gradienten der Fehler zu berechnen und zu entscheiden um wie viel die Gewichte verändert werden müssen. Das wird zuerst für die Knoten getan, die mit der Output-Schicht verbunden sind. Danach die davor liegenden, bis die Inputs wieder erreicht wurden (Going Backwards). Allerdings gibt es dabei folgende Schwierigkeiten: Die Inputs für die Output-Schicht sind unbekannt und die Zielwerte der versteckten Schicht sind unbekannt.

Es ist also möglich den Fehler am Output zu berechnen, da allerdings unbekannt ist welche Inputs diesen Fehler verursacht haben, können die Gewichte nicht einfach wie beim Perceptron angepasst werden. Mit der Kettenregel ist es allerdings möglich das Problem zu umgehen. Details hierzu können der Quelle (Mar09) entnommen werden. Schlussendlich führt das zu zwei verschiedenen Update-Funktionen, eine für jedes Gewichts-Set, welche rückwärts (von den Outputs zu den Inputs) durch das Netzwerk durchgeführt werden müssen.

3.5.3 Klassifizierung

Die Methode, die in dieser Arbeit verwendet wird, um die Klassifikation zu ermöglichen, ist das sogenannte 1-of-N encoding. Dabei wird für jede Klasse ein Knoten bereitgestellt, um diese zu repräsentieren. Das bedeutet, dass ein Vektor erzeugt wird, welcher die Größe der Anzahl der Klassen hat und mit Nullen gefüllt ist, abgesehen von dem Knoten, der die aktuelle Klas-

se widerspiegelt, dieser ist mit einer Eins belegt. Der Vektor [0,0,0,1,0,0] bedeutet, dass dies die vierte der sechs Klassen ist. Bevor also das Training begonnen wird, werden alle Zielklassen in diese Form gebracht und zum trainieren übergeben.

Soll nun ein Datensatz klassifiziert werden, wird die Softmax Methode verwendet, welche das Outputarray mit der Größe der Anzahl der Klassen erstellt, dessen Werte die Summe von 1 bildet. Das Element in dem Array das näher an der 1 ist, als die anderen, gibt die Zielklasse an. Würde folgendes Array als Ausgabe zurückgegeben werden [0.25, 0.05, 0.7], wäre die dritte Klasse die Ausgabeklasse.

3.5.4 MLP Algorithmus

In diesem Unterkapitel wird der Algorithmus des MLPs näher erläutert. Zuerst eine grobe Beschreibung des Algorithmus, danach noch mal detailliert als Pseudocode.

1. Ein Input-Vektor wird den Eingabeknoten übergeben
2. Die Inputs werden vorwärts durch das Netzwerk geführt
   - Die Inputs und die Gewichte der ersten Schicht werden benutzt, um zu bestimmen, was die Outputs der ersten Schicht sind.
   - Die Outputs der ersten Schicht und die Gewichte der zweiten Schicht bestimmen die Outputs der zweiten Schicht.
3. Der Fehler wird über die “sum-of-squares“ Differenz zwischen den Outputs und Targets bestimmt
4. Dieser Fehler wird rückwärts durch das Netzwerk geführt, um die Gewichte der zweiten und ersten Schicht anzupassen.

Im Pseudocode 2 ist der Konstruktor des MLPs zu sehen, dort werden einige Variablen initialisiert. Zum einen die Gewichte welche mit zufälligen Werten zwischen -1 und +1 gefüllt werden. Zum anderen das Beta und Momentum. Das Momentum (übersetzt Triebkraft) wird verwendet, um bei der Fehlerfunktion besser lokale Minima zu überwinden, um die Wahrscheinlichkeit zu erhöhen das globale Minimum zu finden. Üblicherweise wird als Momentum 0.9 gewählt. Das Beta gehört zur Sigmoid Aktivierungsfunktion und ist ein positiver Wert.

Code 2 Konstruktor des MLPs

Funktion: MLP(inputs, targets, beta, momentum)

1: $this.weights1[\#input.columns][nHidden] \leftarrow Random(-1, 1)$
2: $this.weights2[nHidden][\#targets.columns] \leftarrow Random(-1, 1)$
3: $this.beta \leftarrow beta$
4: $this.momentum \leftarrow momentum$

Der Trainingsfunktion (zu sehen in Codeabschnitt 3) werden die Input-Daten, die Zielklassen, das Eta, die Anzahl der Trainingsiterationen und die Anzahl der Knoten in der versteckten Schicht übergeben.

Zum Start der Funktion wird den Inputdaten das Bias angefügt, zu sehen in Zeile 1. In Zeilen 2 und 3 werden zwei Matrizen mit Nullen initialisiert, die später die Werte speichern, mit denen die Gewichte angepasst werden sollen.

Nun beginnt das Training. In Zeile 5 wird das Forwarding durchgeführt, in der ersten Iteration dürfte es noch zu keinen sinnvollen Ergebnissen führen. In den darauf folgenden Zeilen wird der Fehler berechnet, welcher sich aus der Differenz zwischen den Outputs und den Targets berechnen lässt. Anzumerken ist noch, dass in Zeile 6 die Softmax Methode zum Einsatz kommt. Die Fehlerberechnung wurde in dem Kapitel “Going Backwards“ bereits erklärt. Abschließend werden in den Zeilen 10-12 die Gewichte angepasst und die Zeilen der Inputs und Targets gemischt, um das sogenannte Overfitting (das zu genaue Anpassen an die Trainingsdaten) zu verhindern.

Die Forwardfunktion ist im Codeabschnitt 4 zu sehen, welche während des Trainings genutzt wird, um den aktuellen Stand des Trainings zu überprüfen und später für den Anwender genutzt wird die Eingabedaten zu klassifizieren. Am Anfang in Zeile 1 werden wie gewöhnlich

Code 3 Trainings-Funktion des MLPs

Funktion: Train(inputs, targets, eta, nIterations, nHidden)

1: $inputs \leftarrow ConcatenateColum(inputs, -1)$
2: $updateW1[\#weights1.rows][\#weights1.columns] \leftarrow 0)$
3: $updateW2[\#weights2.rows][\#weights2.columns] \leftarrow 0)$
4: **for** $n = 1 \rightarrow nIterations$ **do**
5: $outputs \leftarrow Forwards(inputs)$
6: $deltao \leftarrow (targets - outputs)/\#inputs.rows$
7: $deltah \leftarrow hidden * (1 - hidden) * (Dot(deltao, Transpose(weights2)))$
8: $updatew1 \leftarrow eta * (Dot(Transpose(inputs), deltah[:, : -1])) + this.momentum * updatew1$
9: $updatew2 \leftarrow eta * (Dot(Transpose(this.hidden), deltao)) + this.momentum * updatew2$
10: $this.weights1 \leftarrow this.weights1 + updatew1$
11: $this.weights2 \leftarrow this.weights2 + updatew2$
12: $ShuffleRows(inputs, targets)$
13: **end for**

die Inputs mit den Gewichten multipliziert und in einer Matrix gespeichert. In der zweiten Zeile kommt die Aktivierungsfunktion aus der Formel 2 zum Einsatz. In Zeile 3 wird der Matrix das Bias hinzugefügt, da diese als Input für die zweite Schicht verwendet wird. Darauffolgend werden die neuen Input Werte mit den Gewichten der zweiten Schicht multipliziert. Abschließend wird der Forwarding-Teil der Softmax Methode auf die Outputs angewendet, welche zurückgegeben werden.

Code 4 Forwards-Funktion des MLPs

Funktion: Forwards(inputs)

1: $this.hidden \leftarrow Dot(inputs, this.weights1)$
2: $this.hidden \leftarrow 1/(1 + Exp(-this.beta * this.hidden))$
3: $this.hidden \leftarrow ConcatenateColum(hidden, -1)$
4: $outputs \leftarrow Dot(inputs, this.weights2)$
5: $normalisers = SumColum(Exp(outputs))$
6: **return** $Transpose(Transpose(Exp(outputs))/normalizers)$

3.6 Bagging

Bagging ist eine Methode, um weniger mächtige Klassifizierer zu kombinieren, und so einen mächtigeren Klassifizierer erhalten. In dieser Arbeit wird als Grundlage für den Bagging Algorithmus das Perceptron eingesetzt. Der Algorithmus funktioniert wie folgt: Als erstes wird eine ungerade Anzahl von Perceptrons trainiert. Die Trainingsdaten werden für jedes Perceptron extra zufällig aus den vorhandenen Trainigsdaten gezogen, bis die Datenmenge gleich groß ist. Es kann und soll vorkommen, dass ein Trainingsdatensatz mehrfach in den Trainingsdaten des Perceptrons auftaucht. Soll nun nach dem Trainieren ein Datensatz klassifiziert werden, werden alle Perceptrons befragt. Die Klasse, die von den meisten Perceptrons gewählt wurde, wird als die Ergebnisklasse gewählt. Zu erwähnen ist noch, dass immer eine ungerade Anzahl an schwachen Klassifizierern gewählt werden sollte, damit es bei der Abstimmung der Klasse nicht zu einem Unentschieden kommt.

4 Applikation

Die Applikation, welche die Zeichenerkennung durchführt, ist für das Android Betriebssystem, also in Java, geschrieben. Die Zeichenerkennung begrenzt sich in dieser Arbeit lediglich auf Ziffern.

Zusammengefasst hat die Anwendung folgende Grundfunktionen:

- Erfassen, Normalisieren und Speichern von Trainingsdaten
- Trainieren des Multilayer Perceptrons und des Baggings
- Erkennen von handgeschriebenen Ziffern durch die beiden Machine Learning Algorithmen

In den folgenden Unterkapiteln werden die Grundfunktionen näher erläutert.

4.1 Trainingsdaten

Da es sich bei den hier verwendeten Algorithmen um überwachte Lerner handelt, müssen diese, bevor sie eingesetzt werden können, mit Trainingsdaten trainiert werden. In diesem Fall sind Trainingsdaten vom Menschen gezeichnete Zeichen, deren Zielzeichen mit angegeben wurden. Um diese Trainingsdaten aufzunehmen, hat die Applikation einen Modus, in dem es möglich ist ein Zielzeichen anzugeben und danach beliebig oft das Zeichen auf einer dafür vorgesehenen Fläche zu zeichnen. Das Zeichnen muss

in einer Geste ausgeführt werden, das heißt der Finger darf nicht abgesetzt werden. Der gezeichnete Pfad wird mit Hilfe des $1.5 Recognizers normalisiert und abschließend in das Trainingsdatenset gespeichert. Der Pfad wird auf 64 Zeichen gebracht, sodass 128 Werte einen Datensatz bilden.

4.2 Training der Lerner

Nachdem Trainingsdaten angelegt wurden, können damit die Lerner trainiert werden. Wird das Trainieren durch den Benutzer angestoßen, werden die Trainingsdaten aus einer Datei gelesen und in ein Format gebracht, das von den Lernern interpretiert werden kann. In diesem Fall sind es zwei Matrizen, zum einen die Input Matrix, welche die gezeichneten Ziffern in Form der Pfadkoordinaten enthält und zum anderen die Target Matrix, welche die Zielziffern enthält.

Die Targets müssen allerdings angepasst werden, da diese die Zielzeichen im Klartext enthalten, also die Ziffern 0-9. Da zum Klassifizieren allerdings beim Multilayer Perceptron die Softmax Methode verwendet wird, muss die jeweilige Zielklasse die Form eines Arrays mit 10 (für die Anzahl der Ziffern) Einträgen annehmen, welche entweder mit 0 oder 1 belegt sind. Um dies umzusetzen wird die 0 als folgendes Zielarray umgesetzt [1,0,0,...,0], die 1 als dieses [0,1,0,...,0] usw.

Wurde die Anpassung der Daten vorgenommen, kann der Lerner mit ihnen trainiert werden.

4.3 Zeichenerkennung

Nach dem Training wechselt die Anwendung automatisch in den Erkennungsmodus. Dort ist es dem Benutzer möglich zu zeichnen. Die gezeichneten Ziffern werden ebenfalls wie die Trainingsdaten normiert und an die Lerner gereicht. Diese klassifizieren die Daten und geben die entsprechende Ziffer zurück.

4.4 Testen

Die Anwendung hat eine weitere Funktion, welche es ermöglicht hinterlegte Daten klassifizieren zu lassen. Den Daten ist wie bei den Trainingsdaten die Zielklasse beigefügt, um überprüfen zu können, ob die Daten richtig klassifiziert wurden. Dadurch lässt sich berechnen wie gut klassifiziert wurde, dafür kann der prozentuale Anteil an richtigen Klassifikationen ausgegeben werden.

4.5 Benutzeroberfläche

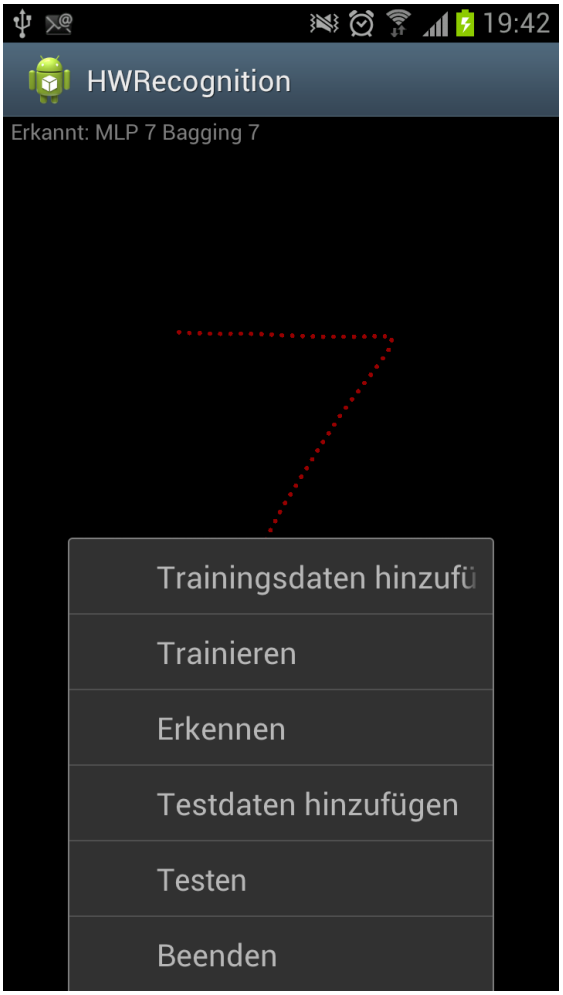

Abbildung 5: Benutzeroberfläche der Android App

In Abbildung 5 ist die vollständige Benutzeroberfläche der Handschrifterkennungs-Applikation zu sehen. Oben links ist die Statusanzeige zu sehen, welche verrät in welchem Modus sich die Anwendung gerade befindet, beziehungsweise welche Ziffer zuletzt erkannt wurde, falls der Erkennungsmodus aktiviert ist. Un-

ten ist das Menü zu sehen, in dem der Benutzer die eben erwähnten Funktionalitäten auswählen kann. Die schwarze Fläche hinter dem Menü kann zum Zeichnen der Ziffern verwendet werden. Dort ist in der Momentaufnahme eine gezeichnete Sieben zu erkennen, welche bereits durch den $1.5 Recognizer normalisiert wurde.

4.6 Grobe Softwarearchitektur

In diesem Abschnitt wird die grobe Softwarearchitektur des Android Prototyps erläutert.

Das Herzstück der Software ist die ControllingUnit, welche alle Eingaben des Users verarbeitet. Hier kann die Aufnahme der Trainings- und Testdaten realisiert werden. Zudem werden dort die Klassifizierer initialisiert, deren Training dort auch angestoßen werden kann. Auch der in den Grundlagen beschrieben $1.5 Recognizer ist dort implementiert, um die Eingabedaten, die von der DrawingView an ihn gesendet werden zu normalisieren. Die DrawingView wird auch dazu verwendet das aktuell gezeichnete Zeichen darzustellen. Zudem gibt es noch die Klassen MLP, Perceptron und Bagging welche die in den Grundlagen erläuterten Machine Learning Algorithmen implementieren.

5 Auswertung

In diesem Abschnitt werden die Beobachtungen vorgestellt, die beim Testen der beiden Machine Learning Verfahren gemacht wurden. Zum Testen der Verfahren wurde ein Testdatenset erstellt, welches aus 100 Datensätzen besteht (10 für jede Ziffer). Zum Testen wurden auch die Parameter der Algorithmen testweise verändert, zum Beispiel wurde beim MLP die Anzahl der versteckten Knoten und der Iterationen zur Fehlerbehebung variiert, beim Bagging wurde die Anzahl der Iterationen beim Trainieren der Perceptrons sowie die Anzahl der Perceptrons verändert. Die Ergebnisse mit den Testdaten werden als Confirmation Matrix dargestellt, welche zeigt in welche Klassen die Testdaten eingeordnet wurden und in welche sie eigentlich gemusst hätten. Zudem wird der prozentuale Anteil der richtigen Einordnungen dargestellt. Getestet wurden die Verfahren auf dem Samsung Galaxy S3 mit einem 1.4GHz Quad-Core Prozessor.

Beim MLP wurden sehr gute Ergebnisse mit 15 versteckten Knoten und 50 Iterationen beobachtet. Die Treffergenauigkeit liegt hier zwischen 95% und 100% je nachdem welche anfänglichen Zufallswerte in die Gewichte gefüllt werden. Dabei dauert die Verarbeitung der ca. 400 Trainingsdatensätzen ungefähr 30 Sekunden.

	Targets									
	1	0	2	3	4	5	6	7	8	9
Outputs										
1	10	0	0	0	0	0	0	0	0	0
0	0	8	0	0	0	0	0	0	0	0
2	0	0	10	0	0	0	0	0	0	0
3	0	0	0	10	0	0	0	0	0	0
4	0	0	0	0	10	0	0	0	0	0
5	0	0	0	0	0	10	0	0	0	0
6	0	0	0	0	0	0	10	0	0	0
7	0	0	0	0	0	0	0	10	0	0
8	0	2	0	0	0	0	0	0	10	0
9	0	0	0	0	0	0	0	0	0	10

98 of 100 correct (98%)

Abbildung 6: Confirmation Matrix des MLPs mit 15 versteckten Neuronen und 50 Trainingsiterationen

Das Bagging Verfahren schneidet in dieser Arbeit im Gegensatz zum MLP sehr schlecht ab. Zum einen dauert das Trainieren wesentlich länger als beim MLP, zum anderen sind die Ergebnisse um einiges schlechter. Das Trainieren von 501 Perceptrons mit 10 Iterationen dauert ca. 30 Minuten. Dabei liegt die Treffergenauigkeit nur bei etwa 50% und weniger, dabei werden wie in der Confirmations Matrix zu sehen ist, einige Ziffern immer richtig und andere immer falsch klassifiziert. Es wäre möglich gewesen den Algorithmus zu parallelisieren, dies würde die Rechenzeit zwar wesentlich verkürzen, allerdings liegt die Vermutung nahe, dass es trotzdem noch zu lange dauern würde die Perceptrons zu trainieren.

6 Fazit

Die prototypisch erstellte Anwendung, welche die beiden Machine Learning Verfahren Multi-

	Targets									
	1	0	2	3	4	5	6	7	8	9
Outputs										
1	10	0	0	0	0	0	0	0	0	10
0	0	0	0	0	0	0	0	0	0	0
2	0	0	10	0	0	0	5	0	0	0
3	0	0	0	3	0	7	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	6	3	3	0	0	0
6	0	0	0	0	0	0	2	0	0	0
7	0	10	0	7	0	0	0	10	2	0
8	0	0	0	0	0	0	0	0	8	0
9	0	0	0	0	4	0	0	0	0	0

46 of 100 correct (46%)

Abbildung 7: Confirmation Matrix des Baggings mit 501 Perceptrons und 10 Trainingsiterationen

layer Perceptron und Bagging verwendet, wurde mit etwa 400 Trainingsdaten trainiert und getestet. Das Training beschränkte sich in dieser Arbeit lediglich auf Ziffern, welche aus normalisierten Pfaden bestanden, die als Bildschirmkoordinaten hinterlegt wurden. Bei den Tests kam heraus, dass das MLP sehr gut abschneidet, nahezu alle Eingaben werden schnell und korrekt erkannt. Das Bagging Verfahren hingegen schneidet sehr schlecht ab und erkennt nur die wenigsten Eingaben richtig. Zudem ist es auch je nach Anzahl der Perceptrons sehr langsam.

Als Erweiterung könnten weitere Zeichen mit trainiert werden, um zu überprüfen, ob die Schrifterkennung mit Hilfe des MLPs auch als Ersatz für die Software Tastatur in Frage kommen könnte. Dazu wäre es auch sinnvoll das Absetzen des Fingers während des Zeichnens zu ermöglichen, da sich dadurch einige Zeichen besser zeichnen lassen.

Literatur

[AG09] Alex Graves, Juergen S.: *Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks*. http://www.idsia.ch/~juergen/nips2009.pdf. Version: 2009

[AS11] Andreas Schmidt, Friedrich Volmering Marc K. Ulrich Schwanecke S. Ulrich Schwanecke: *A $1.5 Gesture Recognizer as Input Device for a Multi-Platform Virtual Game Controller*. http://www.mi.hs-rm.de/~schwan/Veroeffentlichungen/docs/gesture2011.pdf. Version: 2011

[Mar09] Marsland, Stephen: *Machine Learning: An Algorithmic Perspective*. 2009

[VF09] Volkmar Frinken, Andreas Fischer Horst Bunke Trinh-Minh-Tri Do Thierry A. Tim Peter P. Tim Peter: *Improved Handwriting Recognition by Combining Two Forms of Hidden Markov Models and a Recurrent Neural Network*. http://www.springerlink.com/content/t523p78qp350g648/. Version: 2009

[WWL07] Wobbrock, Jacob O. ; Wilson, Andrew D. ; Li, Yang: *Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes*. http://faculty.washington.edu/wobbrock/pubs/uist-07.1.pdf. Version: 2007