Bericht

Josef Schulz

05. Juni 2013

Inhaltsverzeichnis

1 Einleitung

Die Natur hat mit Neuronalen Netzwerken dem Menschen das Lernen und Denken ermöglicht. Neuronale Netze sind unglaublich Flexible Rechensysteme. Sie sind in der Lage Muster zu klassifizieren und in der Test Phase wieder zu erkennen. Einfache Anwendungsbeispiele dieser Art sind Schrifterkennungen und die Musterrekonstruktion. Im Rahmen dieser Praktischen Arbeit soll ein einfaches *feed-forward-* Netzwerk Vektoren klassifizieren. Ich habe mich dieser Aufgabe mit der Sprache C++ gestellt und versucht das geforderte Netz ohne zusätzliche Bibliotheken zu konstruieren.

2 Programm

Das Programm besteht aus 5 Quellcodedateien: *main.cpp, fileReader.h, fileReader.cpp, feedforward.h* und der *feedorward.cpp*. In der *fileReader-* Klasse wird die Datei des Vorgegeben Formates eingelesen und es werden die Trainingsbeispiele extrahiert. Die *feedforward-* Klasse erstellt das Netz mit einer gewünschten Anzahl an versteckten Schichten und stellt eine Methode bereit die Anhand eines Eingabe/Ausgabe Beispiels die Gewichte lernt auf der Basis des error-backpropagation- Verfahrens. In der Datei *main.cpp* werden dem Netz im Wechsel die Beispiele präsentiert und das Netz lernt die Beispiele zu Klassifizieren. Abgebrochen wird das Lernen wenn eine gewünschte Anzahl an Durchläufen überschritten wird oder alle Beispiele Korrekt klassifiziert werden, was aber in der Regel nicht erreicht wird. Aus diesem Grund ist es sinnvoller den Lernprozess in Abhängigkeit der Lernrate, der Anzahl der Schritte und mit der Anzahl der Verborgenen Schichten zu regulieren. Die Lernrate sollte zwischen 0.1 und 0.01 festgelegt werden. Ist sie zu hoch wird zwar schnell gelernt, es kann aber schnell passieren das Minima übersprungen werden oder das System oszilliert. Bei einer kleinen Lernrate wird jedoch extrem langsam gelernt und man kommt von einem Plateau nicht weg, die Anzahl der Lernschritte sollte stark erhöht werden auf über 10000. Etwas Problematischer wird es bei der Anzahl der Zwischenschichten. Ist sie zu klein kann das Netz von Grund auf nicht genügend Vektoren speichern und ist sie zu groß kommt es zum sogenannten *overfitting*-Effekt. Das Programm lässt sich in zwei verschieden Modi starten, zum Lernen und zum Anwenden. Im Lernmodus werden die Gewichte trainiert und gespeichert und im Anwendungsmodus werden die bereits gelernten Gewichtungen aus einer Datei geladen.

Ich möchte nicht im Detail auf den *error-backpropagation-* Ansatz eingehen sondern werde nur die verwendeten delta Terme und Gewichtsveränderungen darstellen:

$$\Delta w_{ij}(t+1) = \eta\delta_i(t)v_j(t)+\alpha\Delta w_{ij}(t)$$

$$W_{ij}(t+1) = W_{ij}(t) + \Delta w_{ij}(t+1)$$

3 Kompilieren und Ausführen

Ich habe das Programm nur unter Linux kompiliert, aber es sollte ebenso unter Windows funktionieren, da nur Standard Bibliotheken zur Anwendung kommen.

Im Ordner liegt eine *run.sh* bereit die das Programm kompiliert und eine Ausführbare Datei unter dem Namen *NSI* erzeugt. Alternativ kann das Programm mit dem Aufruf:

g++ -o NSI main.cpp fileReader.cpp feedforward.cpp

kompiliert werden.

3.1 Ausführen

./NSI train trainingsset-10.txt [options]

Wie oben erwähnt kann das Programm in dem *train* oder *test* Modus gestartet werden, es folgt eine Datei mit den Test bzw. Anwendungsfällen und zusätzlich können noch optionale Einstellungen vorgenommen werden:

./NSI <train/test> <data> -p 100.0

Wird die Option -p gewählt kann ein Prozentsatz der in der Datei verfügbaren Vektoren ausgewählt werden, dieser wird immer von oben nach unten gewählt.

./NSI <train/test> <data> -a 100.0 10000 0.1 0.1 25

Nach dem *-a* folgt wieder der Prozentsatz der Verfügbaren Vektoren. Als nächstes kommt die Anzahl an Maximalen Schritten des Lernvorgangs gefolgt von der Lernrate. Zu Schluss gewichtet noch ein α den Momentum-term. Zum Schluss werden noch die verborgenen Schichten modelliert. Diese werden hinter einander durch Kommata getrennt aufgeführt: 10,4,5.

$$\Delta w_{ij}(t+1) = \eta\delta_i(t)v_j(t)+\alpha\Delta w_{ij}(t)$$

3.2 Details und Ergebnisse

Ist das Programm durchgelaufen werden keine Vektorpaare direkt ausgegeben. Man bekommt eine Liste mit Details zum Lernvorgang dargeboten welche im Anschluss in der *details.txt* gespeichert werden. Die eigentlichen Klassifikationen werden in den Dateien *resultA.txt* und *resultB.txt* gespeichert.

In der *resultA.txt* wird im selben Format gespeichert wie vorgegeben, es werden die Eingabevektoren gefolgt von den Ausgabevektoren gespeichert. Hier sind die Ausgabevektoren jedoch vom Netz errechnet worden. Die Datei *resultB.txt* speichert die eigentlich zu erzielende Ausgabe gefolgt von der vom Netz erzeugten.

Beispieldateien liegen bei.

4 Ausblick

Was sich schnell bemerkbar gemacht hat und sich am Anfang als schwer zu findender Fehler heraus gestellt hat ist der Effekt des *overfittings*. Zu viele Schichten sorgen für eigenwillige Ergebnisse. Um das Lernen weiter zu verbessern ist eine Temperaturfunktion unerlässlich da mit einer konstanten Lernrate η nur schwer gute Ergebnisse erzielt werden können. Des weiteren sollte noch der Lernfortschritt vergangener Epochen mit einbezogen werden:

$$\Delta w_{ij}(t+1) = \eta\delta_i(t)v_j(t)+\alpha\Delta w_{ij}(t)$$

Die Erweiterungen habe ich aus Zeitgründen nicht mehr implementiert. Um noch effizienter zu Werden sollte Parallel mit verschiedenen Startgewichtungen gelernt werden, da so ein globales Minimum gefunden werden kann.