Anhang D: Numerische Tricks

D.1 Allgemeine Bemerkungen zur Optimierung Numerischer Prozesse

Neuronale Netzwerke sind Computerstrukturen höchster Parallelität, welche die Rechenkapazität herkömmlicher, serieller Rechner extrem auslasten. Die Architektur solcher Rechner ist in keinster Weise den Strukturen Neuronaler Netzwerke angepaßt. Dies gilt für Vektorrechner wie eine CRAY Y–MP ebenso wie für Mehrprozessorsysteme wie Transputernetzwerke. Aus diesem Grunde erhebt sich bei der Simulation Neuronaler Netze die Forderung nach Algorithmen, welche die zur Verfügung stehenden Resourcen der Rechner effektiv ausnützen.

Bei der Auswahl von numerisch optimalen Algorithmen spielen normalerweise zwei einander konträre Faktoren eine Rolle: Geschwindigkeit und Platzbedarf. Schnelle Algorithmen erfordern die Abspeicherung von Zwischenergebnissen, also erhöhten Platzbedarf, umgekehrt kann durch die wiederholte Berechnung von Variablen, also durch die Erhöhung der algorithmischen Komplexität, die Speicherung von Variablen vermieden werden.

Bei der Simulation Neuronaler Netze ist meistens der Platzbedarf einer der größten Probleme. Dies hat zwei Gründe: einerseits beansprucht selbst bei moderaten Systemgrößen der Netzwerke die $N \times N$–Matrix der synaptischen Kopplungen erhebliche Bereiche des Zentralspeichers, was die erreichbaren Systemgrößen in der Simulation limitiert, andererseits verlangsamen Prozesse mit großem Platzbedarf auf fast allen Rechnertypen die Ausführungszeit der Programme zum Teil erheblich. Bei Personalcomputern liegt dies hauptsächlich an einer geänderten, langsameren Zugriffsart auf die Variablen, falls Feldgrößen über 64kByte benutzt werden; bei größeren Rechnern, welche immer nur Teile eines Programmes im Arbeitsspeicher halten, erzwingen große Felder das häufige Nachladen von Variablen aus langsameren Massenspeichern. Demgegenüber ist die algorithmische Komplexität Neuronaler Netzwerke eher bescheiden, sodaß Geschwindigkeitsoptimierungen nur eine untergeordnete Rolle spielen.

Von diesen ziemlich allgemeinen Bemerkungen abgesehen ist es schwierig, noch konkretere Richtlinien zur Optimierung Neuronaler Netze anzugeben. Weitergehende Optimierungen zwingen nämlich dazu, rechnerspezifische Gegebenheiten, bis hin zur konkreten Hardware des Rechners, in Betracht zu ziehen und auch auszunutzen. Dies macht die Programme aber rechnerabhängig und sollte, auch wegen dem damit verbundenen Zeitaufwand, nur in wenigen, wichtigen Fällen durchgeführt werden.

D.2 Schnelle Berechnung der Projektor–Matrix

Die Berechnung der Projektor–Matrix über die SVD oder, via Gleichung (B.9), über die Invertierung der Korrelationsmatrix, ist algorithmisch sehr aufwendig und meistens auch speicherplatzintensiv. Der hier vorgestellte, iterative Algorithmus zur Berechnung der Projektor–Matrix vermeidet beide Nachteile. Er ist zudem für Neuronale Netzwerk–Anwendungen besonders attraktiv, da für jedes neue Muster nur ein zusätzlicher Iterationsschritt nötig ist.

Der Algorithmus basiert auf dem Gram-Schmidtschen Orthogonalisierungsverfahren [Ko84, Pe86]. Ausgehend von $\mathcal{P}^0 = \mathbb{1}$, wird iterativ, für jedes Muster, zunächst der orthogonale Projektionsoperator

$$\mathcal{P}_\perp^k = \mathbb{1} - \mathcal{P}^k$$

gebildet. Hierbei ist $\mathcal{P}^k$ die bis dahin errechnete Projektor–Matrix. Mit $\mathcal{P}_\perp^k$ errechnet sich jetzt der Fehlervektor mit dem neuen, $k+1$–ten Muster ξ_i^{k+1} zu

$$|u>= \mathcal{P}_\perp^k \xi^{k+1}$$

Die nächste Projektormatrix ist dann durch

$$\mathcal{P}^{k+1} = \mathcal{P}^k + \frac{|u><u|}{|u|^2}$$

gegeben. Diese Berechnungsschritte müssen für jedes Muster nur einmal durchlaufen werden. Der Algorithmus ist damit sehr schnell, nur unwesentlich langsamer als die Berechnung der Hopfield–Kopplungsmatrix.

Durch geschickte Speicherung der Variablen kann der benötigte Speicherplatz auf die Hälfte reduziert werden, was eine weitere Geschwindigkeitserhöhung zur Folge hat. Erst diese Speicherplatzreduzierung macht auf Personal Computer die Simulation von Neuronalen Netzen mit Systemgrößen bis zu ca. 250 Neuronen möglich. Wir machen uns hierbei einfach zunutze, daß Projektor–Matrizen symmetrische Matrizen sind, also $J_{ij} = J_{ji}$ gilt. Es genügt deshalb, die obere oder untere Dreiecksmatrix von $\mathcal{P}^k$ oder $\mathcal{P}_\perp^k$ abzuspeichern; sie werden beide in der Kopplungsmatrix `J[i][j]` (wir verwenden hier die **C**–Programmiersprache als Code) kombiniert:

$$\texttt{J[i][j]} = \left(\begin{array}{|cc} \mathcal{P}^k & \\ & \mathcal{P}_\perp^k \end{array} \right)$$

Für die noch fehlende Diagonale von $\mathcal{P}^k$ muß ein seperater Vektor `D[i]` eingeführt werden:

$$\texttt{D[i]} = \left(\begin{array}{|c} \\ \end{array} \right)$$

Damit gilt

$$\mathcal{P}_{ij}^k = \begin{cases} \texttt{J[i][j]}, & j > i \\ \texttt{D[i]}, & j = i \\ \texttt{J[j][i]}, & j < i \end{cases} \quad \text{und} \quad \mathcal{P}_{\perp,ij}^k = \begin{cases} \texttt{J[i][j]}, & j \leq i \\ \texttt{J[j][i]}, & j \geq i \end{cases} .$$

Die einzelnen Arbeitschritte können nun in **C**–Code umgesetzt werden. Für die Berechnung des Projektionsoperators in den orthogonalen Raum ergibt sich:

$$\boxed{\mathcal{P}_\perp^k = \mathbb{1} - \mathcal{P}^k} \quad \rightsquigarrow$$

```
for( i=0 ; i<N ; i++)
{
    J[i][i] =  1 - D[i];

    for( j=0 ; j<i ; j++)
        J[i][j] = - J[j][i];
}
```

Stellt `Xi[i]` das gerade zu lernende Muster dar, erhalten wir den Fehlervektor $|u>$ zu

$$\boxed{|u>=\mathcal{P}_{\perp}^{k}\xi^{k+1}} \quad \rightsquigarrow$$

```
for( i=0 ; i<N ; i++ )
{
    u[i] = 0;
    for( j=0 ; j<i ; j++ )
        u[i] += J[i][j] * Xi[j];
    for( j=i ; j<N ; j++ )
        u[i] += J[j][i] * Xi[j];
}
```

Die nächste Projektormatrix erhält man dann durch

$$\boxed{\mathcal{P}^{k+1}=\mathcal{P}^{k}+\frac{|u><u|}{|u|^2}} \quad \rightsquigarrow$$

```
for( i=0 ; i<N ; i++ )
{
    D[i] += u[i] * u[i] / usq;

    for( j=0 ; j<i ; j++ )
        J[j][i] += u[i] * u[j] / usq;
}
```

Hierbei ist `usq` die quadrierte Norm $|u|^2$ des Fehlervektors. Die obigen Iterationen werden für jedes Muster einmal durchgeführt. Zum Schluß muß noch umgespeichert werden, dies erledigt das Programmstück

Umspeichern der Matrix $\rightsquigarrow$

```
for( i=0 ; i<N ; i++ )
{
    J[i][i] = D[i];

    for( j=0 ; j<i ; j++ )
        J[i][j] = J[j][i];
}
```

Damit enthält das Feld `J[i][j]` die Projektor–Matrix in den Raum der Muster. Um die pseudoiverse Kopplungsmatrix zu erhalten, muß lediglich noch die Diagonale genullt werden.

D.3 Effektive Speicherung einer symmetrischen Kopplungsmatrix

Auch bei der Simulation Neuronaler Netze genügt statt der Speicherung der vollen Kopplungsmatrix eines Neuronalen Netzwerkes die Speicherung der unteren oder oberen Hälfte der symmetrischen Kopplungsmatrix. Damit kann fast die Hälfte des vom Netzwerk beanspruchten Speicherplatzes eingespart werden.

Diese Optimierung ist ein Beispiel für eine von der jeweils verwendeten Programmiersprache abhängigen Optimierung. In einem **C**–Programm mit seiner dynamischen Speicherverwaltung läßt sich die erforderliche Dreiecksmatrix `D[i][j]` sehr effizient aufbauen. Dazu wird zunächst ein N–dimensionales Array von Pointern auf eindimensionale Felder erzeugt. Diese eindimensionalen Felder werden dann nacheinander mit den unterschiedlich langen Zeilen der unteren Dreiecksmatrix der Kopplungsmatrix `J[i][j]` besetzt. Der Programmteil

```
double **D;

D = (double **) calloc( N, sizeof(double*) );
for( i=0 ; i<N ; i++ )
        D[i] = (double*) calloc( i+1, sizeof(double) );

for( i=0 ; i<N ; i++ )
for( j=0 ; j<=i ; j++ )
        D[i][j] = J[i][j];
```

erledigt dies*. Beim Zugriff auf ein einzelnes Element `J[i][j]` der Kopplungsmatrix muß nun unterschieden werden, ob der Index `i` größer oder kleiner dem Index `j` ist; es gilt

$$\texttt{J[i][j]} = \begin{cases} \texttt{D[i][j]}, & j \leq i \\ \texttt{D[j][i]}, & j \geq i \end{cases} .$$

Es kann bereits beim Programmieren darauf geachtet werden, daß der zweite Index immer kleiner dem ersten Index bleibt, ein Beispiel für diese Vorgehensweise findet sich im vorherigen Abschnitt. Leider wird dadurch das Programm schnell unübersichtlich. Ist dies unerwünscht, kann durch die Präprozessor–Direktive

```
#define J(i,j)  ((j)<(i)?D[i][j]:D[j][i])
```

beim Zugriff auf Elemente der Kopplungsmatrix, `J[i][j]`, der dann stattdessen im Programmcode durch '`J(i,j)`' erfolgt, die Kopplungsmatrix so benutzt werden, als wäre sie die vollständige, quadratische $N \times N$–Matrix. Allerdings vergrößert sich, durch die in der Präprozessor–Direktive enthaltene Abfrage, die Programmausführungszeit.

In **Fortran** existieren weder Zeiger, noch eine dynamische Speicherverwaltung oder Präprozessordirektiven. Trotzdem ist es möglich, und zwar durch Einführung eines Offsetvektors `c(i)`, ähnlich vorzugehen. Dazu werden die Zeilen der Dreiecksmatrix (wir verwenden hier die obere Dreiecksmatrix von `J(i,j)`) zu einem eindimensionalen Feld `D(i)` der Länge $\frac{N(N-1)}{2}$ zusammengefaßt. Wir haben damit für die Indices beider Matrizen folgenden Zusammenhang vorliegen:

		j →							
		1	2	3	4	5	6	7	Offsetvektor
i									
↓	1	.	1	2	3	4	5	6	-1
	2	.	.	7	8	9	10	11	4
	3	.	.	.	12	13	14	15	8
	4	.	.	.	.	16	17	18	11
	5	.	.	.	.	.	19	20	13
	6	.	.	.	.	.	.	21	14
	7	.	.	.	.	.	.	.	.

Hierbei wurde zudem noch von einer genullten Diagonale der Kopplungsmatrix ausgegangen. Der Offsetvektor `c(i)` gibt für eine feste Zeile `i` den zum Index `j` hinzuzuaddierenden Offset an. Das Matrixelement `J(i,j)` bekommt man dann mittels

$$\texttt{J(i,j)} = \begin{cases} \texttt{D(j+c(i))}, & j > i \\ \texttt{D(i+c(j))}, & j < i \end{cases} .$$

Der Offsetvektor kann iterativ zu Beginn des Programms durch

```
      c(1) = -1
      help = N - 2
      do 1 i=2,N-1
         c(i) = c(i-1) + help
         help = help - 1
1     continue
```

berechnet werden. Auf manchen Prozessoren wird durch diesen Programmierkniff nicht nur Speicherplatz, sondern auch Rechenzeit eingespart. Normalerweise erfordert nämlich der Zugriff auf ein zweidimensionales Feld in Fortran

*Die Programmbeispiele sollen nur der Erläuterung, nicht als Vorlage für wirklichen Programmcode dienen. Deshalb fehlen — aus Gründen der Übersichtlichkeit — im vorherigen Programmstück die Überprüfungen der erfolgreichen Allokation der Variablen.

eine rechenintensive Multiplikation eines der Feldindices mit der Dimension des Feldes. Diese Multiplikation wird bei der hier vorgestellten Methode durch einen einfachen Feldzugriff ersetzt, wofür die meisten Prozessoren optimal ausgelegt sind.

D.4 Erzeugung von Startvektoren

Bei der numerischen Abtastung der Oberfläche des Überlapp–Dodekaeders werden binäre Startvektoren mit genau spezifizierten Überlappungen zu jeweils drei Mustern benötigt. Hier soll der entsprechende Algorithmus vorgestellt werden. Er läßt sich leicht auf mehrere Muster und entsprechend komplexe Überlapp–Konfigurationen erweitern (man beachte jedoch dabei die in Kapitel 2 skizzierten Schwierigkeiten).

Auf den Rauten, aus denen die Oberfläche des Überlapp–Dodekaeders besteht, variieren nur zwei der bei drei Mustern vorhandenen vier UG–Magnetisierungen. Die beiden anderen Untergitter sind voll durchmagnetisiert. Da die Rauten Oberflächen des vierdimensionalen Untergitter–Hyperkubus sind, arbeitet man am besten vollständig im Untergitter–Raum. Dazu sind zunächst die vier möglichen Untergitter festzustellen. Ist `xi[i][j]` die $p \times N$–Mustermatrix, so erhält man die Untergitter–Mächtigkeiten `n_ug[i]` und die Klasseneinteilung `ug[i]` bezüglich der ersten drei Muster durch das Programmstück

```
n_ug[0] = n_ug[1] = n_ug[2] = n_ug[3] = 0;
for( i=0 ; i<n ; i++ )
{
        ug[i] =   (xi[1][i]==xi[0][i]) +
                2*(xi[2][i]==xi[0][i]);
        n_ug[ ug[i] ]++;
}
```

womit dem ersten Untergitter der Index `0` und dem vierten Untergitter der Index `3` zugeordnet wurde. Nach diesem Programmstück enthält der Vektor `n_ug[i]` die Untergitter–Mächtigkeiten der vier Untergitter und der Vektor `ug[i]` für jeden Gitterplatz `i` den Index des dazugehörigen Untergitters.

Das Abtasten der Raute wird anschließend durch eine Doppelschleife erledigt, bei der zwei der vier Untergitter–Magnetisierungen variieren. Die anderen beiden Untergitter–Magnetisierungen bleiben fest auf -1. Es sei `m[i]` der Vektor der mit den Untergitter–Mächtigkeiten skalierten Untergitter–Magnetisierungen. Dann erfolgt durch

```
m[0] = n_ug[0];
m[1] = n_ug[1];
for( m[2]=-n_ug[2]; m[2]<=n_ug[2] ; m[2]+=2 )
{
        for( m[3]=-n_ug[3] ; m[3]<=n_ug[3] ; m[3]-=2 )
        {
                ....
        }
}
```

das Abtasten der Raute.

Der Spinvektor mit den geforderten Untergitter–Magnetisierungen ergibt sich bei vorgegebenen, skalierten Untergitter–Magnetisierungen `m[i]` durch das Programmstück

```
for( i=0 ; i<n ; i++ )
        vec[i] = -xi[0][i];
```

```
for( j=0 ; j<4 ; j++ )
{
        count = 0;
        toflip = (m[j] + n_ug[j])/2;
        while( count!=toflip )
        {
                i = random(n);
                if( ug[i]==j && vec[i]!=xi[0][i] )
                {
                        vec[i] = xi[0][i];
                        count++;
                }
        }
}
```

Hier wird zunächst der zu erzeugende Startvektor `vec[i]` auf `-xi[0][i]` gesetzt, wodurch alle vier Untergitter–Magnetisierungen den Wert -1 annehmen. Danach werden alle Untergitter–Magnetisierungen durch Drehen passender Komponenten des Neuronenvektors `vec[i]` nacheinander auf die vorgegebenen Werte `m[i]` erhöht.