Diplomarbeit

Parallele Implementierung von Conditional Random Fields unter Verwendung von General-Purpose computation on Graphics Processing Units

Nico Piatkowski

14. März 2011

Betreuer:
Prof. Dr. Katharina Morik
Dipl.-Inf. Felix Jungermann

Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS8)
Technische Universität Dortmund

Kurzzusammenfassung

In dieser Diplomarbeit wird eine parallele Implementierung von Conditional Random Fields (CRF) unter Verwendung von General-Purpose computation on Graphics Processing Units (GPGPU) entwickelt. CRFs sind ein strukturelles Verfahren zur Klassifikation strukturierter Daten. Das Lernen der CRF-Klassifikationsfunktion erfolgt durch die Anpassung der Parameter einer Exponentialfamilie mit der Maximum-Likelihood Methode. Diese Modelle zeichnen sich durch eine hohe Vorhersagegenauigkeit bei vergleichsweise langen Trainings- und Vorhersagezeiten aus. Mit dem Ziel die Laufzeiten zu verkürzen werden parallelisierbare Berechnungen auf Grafikprozessoren (Graphics Processing Units, GPU) ausgelagert. Dies führt zur bisher ersten Implementierung von CRFs für GPUs.

Im Grundlagenteil werden die für diese Diplomarbeit zentralen Konzepte wie das Klassifikationsproblem mit strukturiertem Ausgaberaum, die Maximum-Likelihood Methode sowie Graphische Modelle eingeführt. Im darauf folgenden Kapitel wird die Theorie der Conditional Random Fields erläutert und verschiedene Inferenz-, Klassifikations- und Optimierungsalgorithmen vorgestellt und dabei stets auf parallelisierungsrelevante Eigenschaften geachtet. Zur Implementierung wird die CUDA-Architektur von NVidia verwendet. Deren Spezifikation, Unterschiede zur Programmierung konventioneller CPUs sowie anderen GPU-Architekturen werden erläutert und die Programmiersprache CUDA C vorgestellt.

Ob die entwickelte Implementierung eine ähnliche Vorhersagegenauigkeit bei zugleich kürzeren Trainings- und Vorhersagezeiten wie bestehendende CPU-Implementierungen aufweist, zeigt eine abschließende Evaluation. Dazu wurden zwei Referenzimplementierungen ausgewählt und gegen die eigene Implementierung auf drei verschiedenen Datensätzen getestet.

Es zeigt sich, dass die Klassifikationsgüte der hier entwickelten Implementierung mit denen der Referenzimplementierungen vergleichbar ist. Bei steigender Anzahl parallel verarbeiteter Trainingsbeispiele ließ sich zudem eine Beschleunigung der Laufzeiten feststellen. Ein Laufzeitgewinn ergibt sich durch die Verwendung von GPGPU insbesondere dann, wenn die zu klassifiziernden Daten eine hohe Anzahl an Merkmalen besitzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Abkürzungsverzeichnis

Abb. Abbildung

Abs. Abschnitt

Alg. Algorithmus

ALU Arithmetic Logic Unit

AutoDiff Automatische Differentiation

BP Belief Propagation

bzw. beziehungsweise

CPU Central Processing Unit

CRCW Concurrent Read, Concurrent Write

CREW Concurrent Read Exclusive Write

CRF Conditional Random Field

CUDA Compute Unified Device Architecture

d.h. das heißt

DSP Digitaler Signalprozessor

EREW Exclusive Read Exclusive Write

et al. und andere

FB Forward-Backward

GD Gradientenabstieg

GHz Gigahertz

GiB Gibibyte

GPGPU General-Purpose computation on Graphics Processing Units

GPU Graphics Processing Unit

HMM Hidden-Markov-Model

Id Identifikationsnummer

IEEE Institute of Electrical and Electronics Engineers

IIS Improved Iterative Scaling

KiB Kibibyte

L-BFGS Limited Memory Broyden-Fletcher-Goldfarb-Shanno-Algorithmus

LBP Loopy-Belief-Propagation

MAP Maximum-a-posteriori

MBK Maximum-Belief Klassifikation

MCMC Markov Chain Monte Carlo

MiB Mebibyte

MIMD Multiple Instruction Multiple Data

MIRA Margin Infused Relaxed Algorihm

ML Maximum-Likelihood

MP Multiprozessor

MPI Message Passing Interface

MRF Markov Random Field

o.B.d.A. ohne Beschränkung der Allgemeinheit

o.g. oben genannt

o.ä. oder ähnlich

OpenCL Open Compute Language

PGM Probabilistisches Graphisches Modell

POSIX Portable Operating System Interface for Unix

PRAM Parallel Random Access Machine

Pthreads POSIX Threads

s. siehe

s.o. siehe oben

s.u. siehe unten

SDK Software Development Kit

SGD Stochastic Gradient Descent

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SMD Stochastic Meta Descent

SMem Gemeinsamer Speicher

sog. so genannt

Tab. Tabelle

u.a. unter anderem

URL Uniform Resource Locator

WT Work-Time

z.B. zum Beispiel

1 Einleitung

Conditional Random Fields (CRF) sind ein strukturelles Verfahren zur Klassifikation strukturierter Daten. Das Lernen der CRF-Klassifikationsfunktion erfolgt durch die Anpassung der Parameter einer Exponentialfamilie mit der *Maximum-Likelihood* Methode, wobei das resultierende Modell einer bedingten Wahrscheinlichkeitsdichte der Label entspricht. Zur Klassifikation von Beobachtungen wird das wahrscheinlichste Label gemäß der gelernten Dichtefunktion gewählt. Die Abhängigkeitsstruktur der Label wird durch ein Probabilistisches Graphisches Modell erfasst. Da die Abhängigkeitsstruktur der Beobachtungen bei CRFs nicht modelliert wird, ist die Verwendung einer Vielzahl komplexer, überlappender Attribute möglich. Als Konsequenz ergibt sich sowohl eine hohe Vorhersagegenauigkeit, als auch vergleichsweise lange Trainings- und Vorhersagezeiten [23]. Daher ist die Anwendung solcher Modelle sowohl auf großen Datenmengen als auch in Echtzeitszenarien nicht praktikabel oder nur auf große Rechencluster beschränkt. Um die hohe Laufzeit auf Workstations oder eingebetteten Systemen zu reduzieren, könnten parallelisierbare Berechnungen auf *Graphics Processing Units* (GPU) ausgelagert werden. Da die Entwicklung dieser programmierbaren Grafikprozessoren seit Mitte der neunziger Jahre stark voran getrieben wurde, zeichnen sich aktuelle GPUs durch eine hohe Anzahl paralleler Arithmetisch-Logischer-Einheiten (ALU) aus. Obwohl GPUs mit gängigen Sprachen wie C oder Fortran programmierbar sind, unterscheidet sich ihre Programmierung von der von CPUs durch ein spezielles Speichermanagement sowie einen sehr hohen Grad an Datenparallelität, wobei allein die Ausnutzung dieser Unterschiede zu einem Geschwindigkeitsvorteil gegenüber CPU-Programmen führt. Die Verwendung des Grafikprozessors für Berechnungen außerhalb der Grafikberechnungen wird als *General-Purpose computation on Graphics Processing Units* (GPGPU) bezeichnet. GPUs finden heute in einer Vielzahl von Bereichen, wie der Bioinformatik, Finanzmathematik, Molekulardynamik sowie in Strömungssimulationen Verwendung. Im Bereich Data-Mining wurden sie u.a. bereits erfolgreich zur Beschleunigung von Frequent-Itemset-Mining [19] und k-Means Clustering [31] angewandt.

1.1 Ziele

Im Rahmen dieser Diplomarbeit soll untersucht werden, inwieweit sich parallele, programmierbare Grafikprozessoren dazu eignen, das Training von und die Vorhersage mit Conditional Random Fields zu beschleunigen. Da zur Zeit keine Implementierung von CRF unter Verwendung von GPGPU existiert, kann auf keine explizit zu diesem Thema publizierte Literatur zurückgegriffen werden. Daher sollen aufbauend auf einer ausführlichen Einführung in Conditional Random Fields sowie der hier betrachteten parallelen Berechnungsarchitektur parallele Varianten verschiedener Inferenz- und Optimierungsalgorithmen für CRF unter den Gesichtspunkten Komplexität, Datenparallelität und Datenlokalität entwickelt und mit der Programmiersprache CUDA C implementiert werden. Die entwickelte GPU-Implementierung soll zur Evaluation auf verschiedenen Datensätzen getestet und mit bestehendenden CPU-Implementierungen bezüglich Vorhersagegenauigkeit sowie Trainings- und Vorhersagezeiten verglichen werden.

1.2 Übersicht

In Kapitel 2 erfolgt eine Einführung in die Grundbegriffe des Maschinellen Lernens. Dort werden die für diese Arbeit grundlegenden Konzepte wie das Klassifikationsproblem mit strukturiertem Ausgaberaum, die Maximum-Likelihood Methode sowie Graphische Modelle vorgestellt. Im darauf folgenden Kapitel wird die Theorie der Conditional Random Fields erläutert und verschiedene Inferenz-, Klassifikations- und Optimierungsalgorithmen vorgestellt und dabei stets auf parallelisierungsrelevante Eigenschaften geachtet. Im 4. Kapitel wird die grundsätzliche Idee paralleler Berechnungsmodelle vermittelt, sowie die hier verwendete GPGPU Architektur CUDA eingeführt. Im Anschluss wird in Kapitel 5 die im Rahmen dieser Diplomarbeit entwickelte parallele Implementierung von Conditional Random Fields erläutert. In Kapitel 6 wird diese auf verschiedenen Datensätzen getestet und mit zwei Referenzimplementierungen verglichen. Die Resultate werden diskutiert.

2 Grundbegriffe des Maschinellen Lernens

Dieses Kapitel vermittelt die Grundbegriffe des Maschinellen Lernens. Die Ausführungen der Abschnitte 2.1 und 2.2 dienen der Einordnung der in dieser Diplomarbeit angewandten Methoden in den Kontext des Maschinellen Lernens. Darüber hinaus wird eine formale Definition der hier behandelten Problemstellung sowie verschiedene Gütemaße zur Bewertung von Lernergebnissen vorgestellt. In Abschnitt 2.3 wird das Konzept Probabilistischer Modelle eingeführt und anschließend in Abschnitt 2.4 die Maximum-Likelihood Methode erläutert, die grundlegend für die Anwendung solcher Modelle ist. Anschließend wird die Modellierung probabilistischer Zusammenhänge mit Graphen erklärt. Die hier dargestellten Konzepte sind u.a. in [5] oder [30] zu finden.

Das zentrale formale Werkzeug zur Definition und Erklärung der im Folgenden dargestellten Methoden ist die Wahrscheinlichkeitstheorie nach Kolmogorow [38]. Demnach heißt der Funktionswert einer nicht-negativen Funktion $p : \mathcal{P}(A) \rightarrow \mathbb{R}^+$ *Wahrscheinlichkeit*, falls sich ihre Funktionswerte für alle möglichen Argumente, die *Ereignisse*, zu 1 addieren und sich die Wahrscheinlichkeit des Eintretens zweier disjunkter Ereignisse als Summe ihrer Einzelwahrscheinlichkeiten darstellen lässt. Die Funktion p heißt in diesem Fall *Wahrscheinlichkeitsdichtefunktion*, *Wahrscheinlichkeitsfunktion*, *Dichtefunktion* oder einfach nur *Dichte*. Eine Funktion die mehreren disjunkten Ereignissen A und B ihre Wahrscheinlichkeit $p(A, B)$ zuordnet, wird auch als *gemeinsame Verteilung* bezeichnet. Soll die Wahrscheinlichkeit eines Ereignisses A gemessen werden, unter der Annahme das ein anderes Ereignis B bereits eingetreten ist, so wird die *bedingte Verteilung* als $p(A \mid B)$ ("A gegeben B") notiert. Obgleich der Begriff *Verteilung* eigentlich ein anderes Konzept beschreibt, wird in dieser Diplomarbeit das Wort Verteilung als Synonym für Dichte verwendet, da der Begriff in seiner eigentlichen Bedeutung hier nicht verwendet wird.

Eine Funktion die einem Ereignis seine Realisation zuordnet heißt *Zufallsvariable*. Eine Folge von Realisationen einer Zufallsvariablen wird auch als *stochastischer Prozess* bezeichnet. Zur Vereinfachung der Notation werden Realisationen mit ihren korrespondierenden Zufallsvariablen identifiziert, solange die Interpretation eindeutig bleibt. Das heißt, es wird $p(y)$ für $p(Y = y)$ geschrieben. Selbiges gilt für multivariate Zufallsvariablen $\mathbf{Y}$ und ihre Realisationen $\mathbf{y} \in \mathcal{Y}^n$ (Gleichung (2.1)).

$$p(\mathbf{y}) = p(\mathbf{Y} = \mathbf{y}) = p(Y_1 = y_1, Y_2 = y_2, \ldots, Y_n = y_n) \tag{2.1}$$

Die Wahrscheinlichkeitsdichtefunktion einer multivariaten Zufallsvariablen wird in diesem Kontext auch als *Random Field* bezeichnet. Sofern nicht anders angegeben, werden Variablen und Funktionen mit Kleinbuchstaben und Mengen mit Großbuchstaben bezeichnet. Die Bezeichner von Vektoren werden fettgedruckt gesetzt. Die Semantik griechischer Buchstaben ist stets vom Kontext abhängig.

2.1 Lernaufgaben

Das generische Problem des Maschinellen Lernens besteht im Anpassen eines Modells M an eine Menge von Daten $\mathcal{T}$. Diese Anpassung wird als *Lernen* bezeichnet. Einzelne Elemente von $\mathcal{T}$ heißen *Beispiel*. Von $\mathcal{T}$ wird angenommen, dass es eine Teilmenge bzw. eine Stichprobe einer *Grundgesamtheit* $\mathcal{U}$ ist. Die Daten in $\mathcal{U}$ (und damit auch in $\mathcal{T}$) können sowohl quantitativ als auch qualitativ, in ordinaler oder nominaler Form vorliegen und entsprechen realen Messwerten oder sonstigen Beobachtungen, von denen angenommen wird, dass zwischen ihnen ein funktionaler Zusammenhang besteht. Dieser Zusammenhang kann empirisch oder theoretisch belegt, aber auch rein hypothetisch sein. Die Form der Daten sowie der ihnen unterstellte Zusammenhang determinieren die konkrete *Lernaufgabe*. Es werden zwei große Kategorien von Lernaufgaben unterschieden (vgl. [30, S.9]):

1. Das *überwachte Lernen.* Das Lernziel entspricht dem Auffinden einer Abbildung zwischen einzelnen Elementen von $\mathcal{T}$, mit der Hoffnung, dass der gefundene funktionale Zusammenhang auch in $\mathcal{U}$ besteht.

2. *Unüberwachtes Lernen.* Das Lernziel besteht im Auffinden vorher unbekannter Muster in $\mathcal{T}$, mit der Hoffnung, dass diese Muster auch in $\mathcal{U}$ existieren.

Die in dieser Diplomarbeit behandelten Problemstellungen fallen in die Kategorie des überwachten Lernens.

Daneben existieren weitere Kategorien, wie das *Bestärkende Lernen*, dieses entspricht größtenteils dem überwachten Lernen mit dem Unterschied, dass keine Beispiele für die zu lernende Abbildung gegeben sind und stattdessen jeder möglichen Abbildung ein Nutzen bzw. eine Bewertung zugeordnet werden kann, auf dessen Grundlage der Lernprozess stattfindet. Das *Semi-Überwachte Lernen* stellt eine Mischform des überwachten- und unüberwachten Lernens dar, da die zu lernende Abbildung in diesem Fall nur für einige Daten $\mathcal{T}' \subset \mathcal{T}$ verfügbar ist.

Hierbei sei angemerkt, dass die Daten meistens nicht direkt in einer für ein Lernverfahren geigneten Form vorliegen. In diesem Fall werden die Daten vor dem eigentlichen Lernen durch eine *Vorverarbeitung* (engl. *preprocessing*) in eine lernbare Form überführt.

Zur Evaluation eines gelernten Modells können diverse Gütemaße herangezogen werden (s.u.). Da das aus $\mathcal{T}$ gelernte Modell im Anwendungsfall allerdings auf beliebige Daten der Grundgesamtheit angewendet werden soll, ist es sinnvoll, die Güte eines Modells ebenfalls auf Daten zu testen, die nicht in der Trainingsmenge enthalten waren. Dazu werden die verfügbaren Daten in zwei Mengen $\mathcal{T}_{\text{train}}$ und $\mathcal{T}_{\text{test}}$ partitioniert, wobei das Modell aus der *Trainingsmenge* $\mathcal{T}_{\text{train}}$ gelernt und auf der *Testmenge* $\mathcal{T}_{\text{test}}$ evaluiert wird. Diese Partitionierung dient ausschließlich der Evaluation. Im Anwendungsfall wird das Modell stets auf allen verfügbaren Daten trainiert, da durch das Auslassen von Beispielen wichtige, zur korrekten Klassifikation erforderliche Informationen möglicherweise unberücksichtigt blieben.

Da nicht klar ist, welche Partition von $\mathcal{T}$ für eine zuverlässige Bestimmung der Güte geeignet ist, kann dieser Vorgang mehrfach wiederholt werden. Dazu wird $\mathcal{T}$ in k Teilmengen zerlegt, wobei jeweils die k-te Teilmenge die Rolle der Testmenge übernimmt und die Vereinigung der übrigen $k-1$ Teilmengen die Trainingsmenge bildet. Die Güte des Modells entspricht dann dem Mittelwert der k Einzelgüten. Dieses Vorgehen heißt *k-fache Kreuzvalidierung.* Im Extremfall $k = |\mathcal{T}|$, der *Leave-One-Out Kreuzvalidierung,* enthält jede Teilmenge $\mathcal{T}_k$ genau ein Beispiel.

2.2 Klassifikation und Gütemaße

Die *Klassifikation* ist die in dieser Diplomarbeit behandelte Problemstellung des Maschinellen Lernens. Sie fällt in den Bereich des überwachten Lernens. Es gibt unzählige praxisrelevante Instanzen des Klassifikationsproblems, darunter: Sprach-, Handschrift- und Gestenerkennung, Named Entity Recognition, Part-of-Speech Tagging sowie die Klassifikation von Texten, Fotos und bewegten Bildern wie auch von Kundendaten oder sonstigen Messdaten.

Die Trainingsmenge $\mathcal{T} = \left\{(\mathbf{x}, \mathbf{y})^{(i)}\right\}_{1 \leq i \leq N}$ besteht aus Paaren von *Beobachtung* (oder *Merkmalsausprägung*) $\mathbf{x}^{(i)}$ und *Klasse* (engl. *class/label*) $\mathbf{y}^{(i)}$. Eine Beobachtung $\mathbf{x} \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_m$ besteht dabei aus m *Merkmalen*, die alle zur Klassifikation verfügbaren Informationen beinhalten, wobei hier zur Vereinfachung der Notation angenommen wird, dass die Menge $\mathcal{X}$ die Ausprägungen aller Merkmale enthält, also $\mathbf{x} \in \mathcal{X}^m$ gilt. $\mathbf{x}$ wird dabei als bekannt und $\mathbf{y} \in \mathcal{Y}^n$ als unbekannt oder nur im Nachhinein beobachtbar aufgefasst. Die Menge $\mathcal{Y}$ enthält alle Klassen denen ein Beispiel zugehörig sein kann. Sind dies nur zwei Klassen, so spricht man von einem *binären Klassifikationsproblem* und im Fall $|\mathcal{Y}| > 2$ von einem *Mehrklassenproblem.* Ist $n > 1$, d.h.

$\mathbf{y}$ ist ein Vektor von Klassen, so liegt ein *Klassifikationsproblem mit strukturiertem Ausgaberaum* vor. Zur Wahrung der Allgemeingültigkeit wird hier stets die vektorielle Schreibweise für Klassen verwendet.

Die Aufgabe besteht nun darin, ein Modell $M : \mathcal{X}^m \to \mathcal{Y}^n$ zu finden, so dass unklassifizierten Beispielen $\mathbf{x}$ ihre korrekte Klasse $\mathbf{y}$ zugeordnet werden kann. Um dies zu erreichen wird eine sog. *Verlustfunktion* minimiert, die den Fehler des Modells M auf den Daten $\mathcal{T}$ misst. Die Verlustfunktion kann beispielsweise der Anzahl aller falschen Klassifikationen entsprechen. Sei dazu $1_{\mathcal{T}}\left(\mathbf{x}^{(i)}, \mathbf{y}\right)$ die Funktion, die genau dann den Wert 1 annimmt, falls das Label $\mathbf{y}$ nicht mit dem wahren Label des i-ten Beispiels übereinstimmt, und ansonsten den Wert 0.

$$1_{\mathcal{T}}\left(\mathbf{x}^{(i)}, \mathbf{y}\right) = \begin{cases} 1 & \text{, falls } \left(\mathbf{x}^{(i)}, \mathbf{y}\right) \neq (\mathbf{x}, \mathbf{y})^{(i)} \\ 0 & \text{, sonst} \end{cases}$$

Der sog. 0-1-Verlust L_{01} des Modells M auf den Daten $\mathcal{T}$ entspricht dann Gleichung (2.2).

$$L_{01}\left(\mathcal{T}, M\right) = \sum_{i=1}^{|\mathcal{T}|} 1_{\mathcal{T}}\left(\mathbf{x}^{(i)}, M\left(\mathbf{x}^{(i)}\right)\right) \tag{2.2}$$

Ist der Wert der in Gleichung (2.2) gezeigten Verlustfunktion 0, so entsprechen die Paare $\left(\mathbf{x}^{(i)}, M\left(\mathbf{x}^{(i)}\right)\right)$ genau den Paaren $\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$ der Menge $\mathcal{T}$.

Gütemaße. Bei den im Folgenden vorgestellten Gütemaßen werden nur vollständig korrekte Labelvektoren gezählt. Insbesondere bei Klassifikationsproblemen mit strukturiertem Ausgaberaum kann es sinnvoll sein, die Korrektheit einzelner Komponenten eines Labelvektors $\mathbf{y}$ zu zählen. Dies ist beispielsweise bei den Daten, die in den Abschnitten 6.3.1 und 6.3.2 vorgestellt werden, der Fall. Dort wird das sog. *IOB-Tagging* verwendet, wobei die Korrektheit einzelner Teilstrukturen gezählt wird. Dazu werden Label verwendet, die den Anfang (Begin, B), die Zugehörigkeit (Inner, I) sowie die Nichtzugehörigkeit (Outer, O) zu einer Teilstruktur kennzeichnen. Dies wird im Folgenden jedoch nicht explizit berücksichtigt, da die hier gezeigten Gütemaße allgemeingültig und nicht davon abhängig sind, was genau als korrekte Klassifikation gezählt wird. Wie solche IOB-Daten konkret aussehen ist den o.g. Abschnitten zu entnehmen. Laut [68] basiert das IOB-Tagging auf der Arbeit von Ramshaw und Marcus [58].

Die *Rate der korrekt klassifizierten Beispiele* (engl. *accuracy*) eines Modells entspricht der Wahrscheinlichkeit einer korrekten Klassifikation eines Beispiels aus $\mathcal{T}$ und lässt sich leicht mit Hilfe des 0-1-Verlusts formulieren (2.3).

$$\begin{aligned} Acc\left(\mathcal{T}, M\right) &= p\left(M\left(\mathbf{x}^{(i)}\right) = \mathbf{y}^{(i)} \mid \mathcal{T}\right) \\ &= \frac{\text{Anzahl korrekter Klassifikationen}}{\text{Anzahl der Beispiele}} \\ &= \frac{|\mathcal{T}| - L_{01}\left(\mathcal{T}, M\right)}{|\mathcal{T}|} \end{aligned} \tag{2.3}$$

Solange die Häufigkeiten der einzelnen Klassen in $\mathcal{T}$ nicht stark voneinander abweichen, ist die Accuracy ein angemessenes Maß für die Güte eines Klassifikators. Treten dagegen bestimmte Klassen besonders häufig auf, so ist die Accuracy wenig aussagekräftig. Gehören beispielsweise 90% der Beispiele eines binären Klassifikationsproblems zur Klasse $\mathbf{y}_0$, so erreicht ein Klassifikator, der

jedem Beispiel eben diese Klasse zuweist, eine Accuracy von $0,9$ bzw. 90%. Die Wahrscheinlichkeit einer korrekten Vorhersage ist also sehr hoch, obwohl der Klassifikator alle zu $\mathbf{y}_1$ gehörigen Beispiele falsch klassifiziert. Zur Formulierung von Gütemaßen die solche Schwächen aufdecken, ist es naheliegend, die Klassifikationsgüte auf den Beispielen von $\mathcal{T}$ zu betrachten, die nur zu einer bestimmten Klasse gehören. Sei dazu $\mathcal{T}_\mathbf{y}$ die Teilmenge der Beispiele in $\mathcal{T}$ die zur Klasse $\mathbf{y}$ gehören. Die Accuracy eines Modells auf $\mathcal{T}_\mathbf{y}$ entspricht der Wahrscheinlichkeit, ein Beispiel $\mathbf{x}$ der Klasse $\mathbf{y}$ zuzuordnen, unter der Bedingung, dass das wahre Label von $\mathbf{x}$ tatsächlich $\mathbf{y}$ ist. Dieses Gütemaß wird *Sensitivität* (engl. *recall*) genannt.

$$\begin{aligned} Rec\left(\mathcal{T}, M, \mathbf{y}\right) &= p\left(M\left(\mathbf{x}^{(i)}\right) = \mathbf{y} \,|\, \mathbf{y}^{(i)} = \mathbf{y}, \mathcal{T}\right) \\ &= \frac{\text{Anzahl korrekter Klassifikationen in Klasse } \mathbf{y}}{\text{Anzahl aller Beispiele mit Klasse } \mathbf{y}} \\ &= \frac{|\mathcal{T}_\mathbf{y}| - L_{01}\left(\mathcal{T}_\mathbf{y}, M\right)}{|\mathcal{T}_\mathbf{y}|} \end{aligned} \tag{2.4}$$

Die Wahrscheinlichkeit ein Beispiel $\mathbf{x}$ korrekt zu klassifizieren, unter der Bedingung, dass es der Klasse $\mathbf{y}$ zugeordnet wurde, heißt *Genauigkeit* (engl. *precision*).

$$\begin{aligned} Prec\left(\mathcal{T}, M, \mathbf{y}\right) &= p\left(M\left(\mathbf{x}^{(i)}\right) = \mathbf{y}_i \,|\, M\left(\mathbf{x}^{(i)}\right) = \mathbf{y}, \mathcal{T}\right) \\ &= \frac{\text{Anzahl korrekter Klassifikationen in Klasse } \mathbf{y}}{\text{Anzahl aller Klassifikationen in Klasse } \mathbf{y}} \\ &= \frac{|\mathcal{T}_\mathbf{y}| - L_{01}\left(\mathcal{T}_\mathbf{y}, M\right)}{|\left\{\mathbf{x} \in X \,|\, M\left(\mathbf{x}\right) = \mathbf{y}\right\}|} \end{aligned} \tag{2.5}$$

Genauigkeit und Sensitivität lassen sich mit Hilfe der Formeln (2.5) und (2.4) berechnen. Die isolierte Betrachtung eines der beiden Maße kann, wie auch bei der Accuracy, zu falschen Schlüssen führen. Ein System das beispielweise immer die Klasse $\mathbf{y}$ vorhersagt, erreicht auf dieser Klasse einen Recall von 100%, da jedes Vorkommen der Klasse sicher richtig erkannt wird. Die entsprechende Precision wäre aufgrund der vielen falschen Vorhersagen allerdings gering. Daher wird in der Regel das gewichtete harmonische Mittel von Precision und Recall (2.6), das sog. *F_β-Maß* (engl. *F_β-score*), zur Bewertung der Klassifikationsgüte herangezogen. Es wurde 1979 von van Rijsbergen [70] im Kontext des *Information Retrieval* eingeführt. Mit der Wahl von β lässt sich die Gewichtung von Precision und Recall festlegen:

- Für $\beta = 1$ werden beide gleich gewichtet.
- Für $\beta = \frac{1}{2}$ wird die Precision doppelt so stark gewichtet wie der Recall.
- Für $\beta = 2$ wird der Recall doppelt so stark gewichtet wie die Precision.

$$F_\beta\left(\mathcal{T}, M, \mathbf{y}\right) = \left(1 + \beta^2\right) \cdot \frac{Prec\left(\mathcal{T}, M, \mathbf{y}\right) \cdot Rec\left(\mathcal{T}, M, \mathbf{y}\right)}{\left(\beta^2 \cdot Prec\left(\mathcal{T}, M, \mathbf{y}\right)\right) + Rec\left(\mathcal{T}, M, \mathbf{y}\right)} \tag{2.6}$$

Zur Lösung des Klassifikationsproblems mit nicht strukturiertem Ausgaberaum ($n = 1$) existieren diverse Ansätze. Darunter die Stützvektormethode (SVM) [71, 15], Entscheidungsbäume [56], Neuronale Netze [4, 30] sowie direkte Modelle der Verteilung der Daten, sog. Probabilistische Modelle, wie z.B. Naive Bayes oder die Logistische Regression [30]. Ist der Ausgaberaum jedoch strukturiert ($n > 1$), so kommen zur Lösung lediglich strukturelle SVMs [69] oder Probabilistische Modelle, wie Hidden Markov Models [57] oder Conditional Random Fields [41] in Frage. Die hier behandelten und in Abschnitt 2.5 vorgestellten Probabilistischen Graphischen Modelle können bei jedem der oben eingeführten Klassifikationsprobleme eingesetzt werden. Sie sind insbesondere für die Klassifikation von Beobachtungen mit strukturiertem Ausgaberaum geeignet, da sie die Modellierung von Abhängigkeiten zwischen Labeln und Beobachtungen sowie zwischen einzelnen Labeln und einzelnen Beobachtungen erlauben.

2.3 Probabilistische Modelle

Betrachtet man das Klassifikationsproblem vor dem Hintergrund der Wahrscheinlichkeitstheorie, so liegt es nahe, die Merkmale und Klassen als Zufallsvariablen aufzufassen. Wäre deren gemeinsame Verteilung bekannt, so könnte die Wahrscheinlichkeit des simultanen Auftretens von Merkmalsausprägungen und Label berechnet und zur Klassifikation eingesetzt werden, indem die wahrscheinlichste Klasse vorhergesagt wird. Ist eine Trainingsmenge gegeben, so ließe sich der stochastische Prozess, der die Trainingsdaten erzeugt hat, modellieren, indem jedem Ereignis eine Eintrittswahrscheinlichkeit zugeordnet wird. Dazu muss eine parametrisierte Wahrscheinlichkeitsfunktion $p_{\boldsymbol{\theta}}$ ausgewählt werden, die möglichst gut zur Verteilung der Daten passt. Solche Modelle stochastischer Prozesse heißen *Probabilistische Modelle*. Die wahre Verteilung der Daten ist dabei unbekannt und die Selektion einer geeigneten parametrisierten Dichtefunktion ist im Allgemeinen ein schweres Problem. Ist die Wahl getroffen, so können die Wahrscheinlichkeiten einzelner Ereignisse mit Hilfe reeller Zahlen $\boldsymbol{\theta} \in \mathbb{R}^d$, den Parametern, gesteuert werden. Mit diesen kann die Dichtefunktion $p_{\boldsymbol{\theta}}$ an die empirische Verteilung bereits beobachteter Ereignisse angepasst werden. Ist dies gelungen, so ist es möglich die Eintrittswahrscheinlichkeiten zukünftiger Ereignisse mit Hilfe von $p_{\boldsymbol{\theta}}$ zu berechnen und damit das Klassifikationsproblem zu lösen.

Ein Probabilistisches Modell ist also eine *Familie* von Wahrscheinlichkeitsdichtefunktionen. Eine Menge A heißt *Familie*, falls die Elemente von A durch eine weitere Menge indiziert sind. Dies ist bei den hier betrachteten Modellen der Fall, da die einzelnen Dichtefunktionen durch die Menge $\mathbb{R}^d$ indiziert werden.

Es folgt ein einführendes Beispiel für die Klassifikation mit Probabilistischen Modellen. Gegeben sei die Trainingsmenge $\mathcal{T} = \left\{\left(y^{(i)}, x^{(i)}\right)\right\}_{1 \leq i \leq N}$. $x^{(i)} \in \mathbb{R}$ seinen reelwertige Beobachtungen (Messwerte) und $y^{(i)} \in \{+, -\}$ die Klassen. Von den Beobachtungen wird angenommen, dass sie durch eine von zwei möglichen Normalverteilungen erzeugt wurden. Folglich wird die Dichtefunktion der Normalverteilung (2.7) als Dichte des Modells gewählt. Die Parameter dieser Dichtefunktion sind der Erwartungswert μ sowie die Varianz σ^2. Da allerdings zwei Funktionen – eine für die Klasse "+" und eine für "−" – geschätzt werden sollen, ist der Parametervektor des Modells $\boldsymbol{\theta} = \left(\mu_+, \sigma_+^2, \mu_-, \sigma_-^2\right) \in \mathbb{R}^4$.

$$p_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.7}$$

Die Parameter können nun durch schlichtes Ausprobieren, heuristisch oder aber analytisch bestimmt werden. Die geläufigen Schätzer der Parameter einer Normalverteilung werden entsprechend Gleichung (2.8) als empirischer Mittelwert sowie entprechend Gleichung (2.9) als mittlere quadratische Abweichung von eben diesem bestimmt.

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x^{(i)} \tag{2.8}$$

$$\hat{\sigma}^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(x^{(i)} - \hat{\mu} \right)^2 \tag{2.9}$$

Damit können Schätzer für die Parameter μ_+, σ_+^2 aller Beobachtungen der Klasse "+" sowie für die der Klasse "−" berechnet werden. Ist nun ein neuer Messwert zu klassifizieren, so werden die Wahrscheinlichkeiten beider Klassen berechnet und diejenige mit der größeren Wahrscheinlichkeit vorhergesagt.

Obwohl dieses Beispiel einfach ist, vermittelt es doch die grundsätzliche Vorgehensweise bei der Anwendung Probabilistischer Modelle. Im folgenden Abschnitt wird gezeigt, dass die Formeln für die Schätzer (2.8) und (2.9) keinesfalls "vom Himmel" fallen, sondern mit methodischem Vorgehen hergeleitet werden können.

2.4 Maximum-Likelihood Schätzung

Eine Technik zur Schätzung der unbekannten Parameter eines Probabilistischen Modells ist die *Maximum-Likelihood Methode*. Dabei wird angenommen, dass die einzelnen Beobachtungen einer Menge $\mathcal{T} = \left\{ x^{(i)} \right\}_{1 \leq i \leq N}$ von N Werten von ein und demselben stochastischen Prozess unabhängig erzeugt wurden. Basierend auf dieser Annahme können die Parameter $\boldsymbol{\theta}^* \in \mathbb{R}^d$ als diejenigen mit maximaler Wahrscheinlichkeit bei gegebenen Daten $\mathcal{T}$ (2.10) gewählt werden.

$$\boldsymbol{\theta}^* \quad = \quad \arg\max_{\boldsymbol{\theta} \in \mathbb{R}^d} p\left(\boldsymbol{\theta} \,|\, \mathcal{T}\right) \tag{2.10}$$

Diese Wahrscheinlichkeitsfunktion ist in der Regel unbekannt. Allerdings kann sie mit Hilfe von Bayes' Theorem zu Gleichung (2.11) umgeformt werden.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta} \in \mathbb{R}^d} \frac{p\left(\mathcal{T} \,|\, \boldsymbol{\theta}\right) \cdot p\left(\boldsymbol{\theta}\right)}{p\left(\mathcal{T}\right)} \tag{2.11}$$

Da die vorhandenen Daten $\mathcal{T}$ fest sind, ist der Wert von $p\left(\mathcal{T}\right)$ konstant. Daher ändert sich die Maximalstelle in Gleichung (2.11) nicht, wenn die Division durch $p\left(\mathcal{T}\right)$ einfach ausgelassen wird und es ergibt sich Gleichung (2.12). Die Wahrscheinlichkeit $p\left(\mathcal{T} \,|\, \boldsymbol{\theta}\right)$ entspricht dem Funktionswert der parametrisierten Dichtefunktion $p_{\boldsymbol{\theta}}$. Diese kann für einen gegebenen Parameterverktor $\boldsymbol{\theta}$ berechnet werden.

$$\boldsymbol{\theta}^{**} = \arg\max_{\boldsymbol{\theta} \in \mathbb{R}^d} p\left(\mathcal{T} \,|\, \boldsymbol{\theta}\right) \cdot p\left(\boldsymbol{\theta}\right) \tag{2.12}$$

Der Parametervektor $\boldsymbol{\theta}^{**}$ aus Gleichung (2.12) heißt *Maximum-a-Posteriori-Schätzer* (MAP-Schätzer) für $\boldsymbol{\theta}$. Für eine solche Schätzung muss die a-priori Verteilung von $\boldsymbol{\theta}$ allerdings bekannt sein. Dafür wird häufig die Annahme von normalverteilten Parametern gemacht, da die Normalverteilung diejenige mit der größten Entropie [64] unter allen Verteilungen ist. In diesem Fall wird für $p\left(\boldsymbol{\theta}\right)$ eine multivariate Version von Gleichung (2.7) verwendet. Dieses Vorgehen ist allerdings mit weiteren, nicht trivialen Annahmen verbunden, da eine solche Wahrscheinlichkeitsfunktion wiederum unbekannte Parameter besitzt.

Sind keine a-priori Informationen über die Verteilung von $\boldsymbol{\theta}$ bekannt, kann ebenso angenommen werden, dass die Parameter gleichverteilt sind. In diesem Fall sind alle möglichen Parameter gleich wahrscheinlich und $p(\boldsymbol{\theta})$ kann in Gleichung (2.12) entfallen, was zu Gleichung (2.13) führt.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}\in\mathbb{R}^d} p(\mathcal{T} \mid \boldsymbol{\theta}) \tag{2.13}$$

Die Maximalstelle $\boldsymbol{\theta}^*$ dieser Gleichung wird *Maximum-Likelihood Schätzer* (ML-Schätzer) genannt. Die Wahrscheinlichkeit $p(\mathcal{T} \mid \boldsymbol{\theta})$ heißt in diesem Fall *Likelihood* und man schreibt $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) := p(\mathcal{T} \mid \boldsymbol{\theta})$. Da die Elemente aus $\mathcal{T}$ unabhängig sind, entspricht ihre gemeinsame Verteilung dem Produkt der jeweiligen Einzelwahrscheinlichkeiten. Es ergibt sich schließlich Gleichung (2.14) als endgültige *Likelihood-Funktion*.

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \prod_{i=1}^{N} p_{\boldsymbol{\theta}}\left(x^{(i)}\right) \tag{2.14}$$

Aufgrund numerischer Instabilitäten bei der Multiplikation kleiner Wahrscheinlichkeiten, wird anstatt $\mathcal{L}$ ihr Logarithmus $l(\boldsymbol{\theta}; \mathcal{T}) := \ln \mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ maximiert. Eine Optimalstelle dieser sog. *Log-Likelihood* (2.15) ist aufgrund der Monotonie des Logarithmus ebenfalls eine Optimalstelle von $\mathcal{L}$. Dank dieser Transformation müssen zur Berechnung von l lediglich die Logarithmen der Wahrscheinlichkeiten aufsummiert werden.

$$l(\boldsymbol{\theta}; \mathcal{T}) = \sum_{i=1}^{N} \ln p_{\boldsymbol{\theta}}\left(x^{(i)}\right) \tag{2.15}$$

Das folgende Beispiel soll die grundlegende Vorgehensweise bei der Anwendung der Maximum-Likelihood Methode demonstrieren. Gesucht sei der Maximum-Likelihood Schätzer μ^* für den Erwartungswert der Normalverteilung. Dazu setzt man die Dichtefunktion der Normalverteilung (2.7) in $l(\boldsymbol{\theta}; \mathcal{T})$ ein und erhält die Log-Likelihood Funktion der Normalverteilung (2.16). Hierbei bezeichnet $x^{(i)}$ die i-te beobachtete Realisation einer Zufallsvariablen.

$$\begin{aligned} l_{\text{normal}}(\mu, \sigma; \mathcal{T}) &= \sum_{i=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{\left(x^{(i)} - \mu^*\right)^2}{2\sigma^2}\right)\right) \\ &= -\frac{N}{2}\ln\left(2\pi\sigma^2\right) - \sum_{i=1}^{N} \frac{\left(x^{(i)} - \mu^*\right)^2}{2\sigma^2} \end{aligned} \tag{2.16}$$

Da derjenige Parameter $\mu^* \in \mathbb{R}$ gesucht ist, für den l_{normal} ihren maximalen Wert annimmt, wird die Funktion partiell nach μ^* abgeleitet (2.17).

$$\frac{\partial l_{\text{normal}}(\mu, \sigma; \mathcal{T})}{\partial \mu^*} = \sum_{i=1}^{N} \frac{x^{(i)} - \mu^*}{\sigma^2} \tag{2.17}$$

Setzt man diese partielle Ableitung $= 0$, so ergibt sich das Arithmetische Mittel (2.18) als

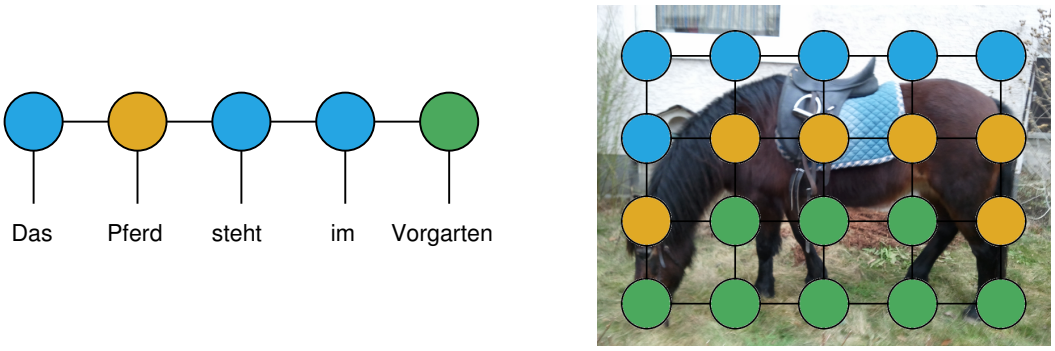


Abbildung 2.1: Beispielhafte Anwendung Graphischer Modelle zur Text- (links) und Bildsegmentierung (rechts). In beiden Fällen wird versucht komplexe, reale Daten mit einer überschaubaren graphischen Struktur zu überdecken. So erhält man ein vereinfachtes Modell der Daten. Auf diese Modelle können probabilistische Methoden angewendet werden, um Rückschlüsse auf den Inhalt der Daten zu ziehen. So könnte man versuchen mit Hilfe von PGMs die Fragen zu beantworten, ob auf dem rechten Bild ein Pferd zu sehen ist, oder ob der linke Satz eine Ortsangabe enthält.

Maximum-Likelihood Schätzer für den Erwartungswert einer Normalverteilung.

$$\begin{aligned} 0 &= \sum_{i=1}^{N} \frac{x^{(i)} - \mu^*}{\sigma^2} \\ \Leftrightarrow 0 &= \sum_{i=1}^{N} x^{(i)} - \sum_{i=1}^{N} \mu^* \\ \Leftrightarrow \mu^* &= \frac{1}{N} \sum_{i=1}^{N} x^{(i)} \end{aligned} \tag{2.18}$$

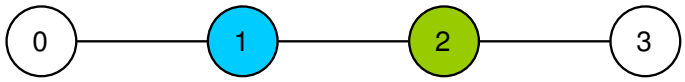
Diese Methode wird in ähnlicher Form in Kapitel 3 angewendet, um die Parameter der in dieser Diplomarbeit behandelten Conditional Random Fields zu schätzen.

2.5 Probabilistische Graphische Modelle

In diesem Abschnitt wird die Klasse der *Probabilistischen Graphischen Modelle* (PGM) eingeführt, zu denen auch die in dieser Diplomarbeit betrachteten Conditional Random Fields zählen. PGMs eignen sich insbesondere für die Klassifikation mit strukturiertem Ausgaberaum, da es mit diesen möglich ist, Interaktionen zwischen einzelnen Labeln und Beobachtungen zu modellieren. In diesem Abschnitt wird die Notation für PGMs sowie der Begriff der Randverteilung eingeführt. In Abschnitt 2.3 wurde erwähnt, dass die Wahl einer passenden Dichtefunktionen eines Probabilistischen Modells im Allgemeinen ein schweres Problem darstellt. Motiviert durch diese Aussage, wird in Abs. 2.6 eine Klasse von PGMs eingeführt, deren Dichtefunktion aus speziellen Modelleigenschaften hergeleitet werden kann. Hier sei einschränkend erwähnt, dass in dieser Diplomarbeit ausschließlich Modelle für Zufallsvariablen mit nominalem Bildraum betrachtet werden. Die Theorie von PGMs umfasst zwar auch reelwertige Label und Beobachtungen, da dieser Abschnitt allerdings als Grundlage für die im folgenden Kapitel eingeführten Conditional Random Fields dienen soll und diese zur Segmentierung nominaler Daten verwendet werden, wird auf die Berücksichtigung reelwertiger Realisationen verzichtet.

Definition 2.1. Ein Probabilistisches Graphisches Modell M ist ein Paar $(G, p_{\boldsymbol{\theta}})$, bestehend aus einer multivariaten Dichtefunktion $p_{\boldsymbol{\theta}}$ sowie einem Graphen G.

Wie auch bei den Probabilistischen Modellen ist $p_{\boldsymbol{\theta}}$ eine parametrisierte Dichte, die einer Realisa-



$$p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\{v_1,v_2\}}\right) = p_{\boldsymbol{\theta}}\left(y_{v_1} = \text{blau}, y_{v_2} = \text{grün}\right) = \sum_{y,y' \in \mathcal{Y}} p_{\boldsymbol{\theta}}\left(y_{v_0} = y, y_{v_1} = \text{blau}, y_{v_2} = \text{grün}, y_{v_3} = y'\right)$$

Abbildung 2.2: Dargestellt ist ein Graph mit vier Knoten $V = \{v_0, \ldots, v_3\}$. Die möglichen Realisationen der Zufallsvariablen sind $\mathcal{Y} = \{\text{gelb}, \text{grün}, \text{blau}\}$. Die Menge U enthält die Knoten $\{v_1, v_2\}$. Die Notation $\mathbf{y}_U$ bezeichnet eine Realisation der Knoten v_1 und v_2. Ist zum Beispiel $\mathbf{y}_{\{v_1,v_2\}} = \{y_{v_1} = \text{blau}, y_{v_2} = \text{grün}\}$, so müssen zur Berechnung der Randverteilung $p(\mathbf{y}_U)$ die Wahrscheinlichkeiten der Farbkombinationen aller Knoten in denen der Knoten v_1 blau und der Knoten v_2 grün ist aufsummiert werden. Das heißt, dass die Farben gelb, grün und blau für die Knoten v_0 und v_3 in Betracht gezogen werden müssen, um die Wahrscheinlichkeit zu berechnen, dass die Knoten v_1 und v_2 blau bzw. grün sind.

tion einer Zufallsvariablen ihre Wahrscheinlichkeit zuweist. Sind die Realisationen allerdings keine reelwertigen Zahlen, wie in Abs. 2.3, sondern komplexe Strukturen, so kann es hilfreich sein, diese in vereinfachte Teilstrukturen zu zerlegen, wobei jede einer eigenen Zufallsvariablen entspricht. Der Graph eines PGMs dient der Modellierung dieser Zerlegung. In diesem Fall wird eine Dichtefunktion benötigt, die den Realisationen der einzelnen Teilstrukturen ihre Wahrscheinlichkeit zuordnet. Auch in PGMs stellt die Wahl einer passenden Dichtefunktion im Allgemeinen ein schweres Problem dar. Abbildung 2.1 soll das Konzept Probabilistischer Graphischer Modelle sowie die grundsätzliche Vorgehensweise bei der Modellbildung verdeutlichen.

Im Folgenden wird die formale Notation für Graphen und Realisationen von Zufallsvariablen sowie die Randverteilung von Zufallsvariablen eingeführt. In PGMs ist $G = (V, E)$ ein Graph mit Knotenmenge $V = \{v_1, \ldots, v_n\}$, $U \subseteq V$ und Kantenmenge $E \subseteq V^2$. Zwei Knoten u und v heißen *benachbart* oder *adjazent*, falls (u, v) eine Kante in E ist. Die mit einem Knoten v verbundenen Kanten heißen *inzident*. Ist ein Knoten zu genau einer Kante inzident, wird dieser auch als *Blatt* bezeichnet. Ein Graph dessen Kanten keinen Kreis schließen heißt *Baum*. Jeder Knoten $v \in V$ wird mit einer Zufallsvariablen indentifiziert (vgl. [30, S.625]). Bei PGMs findet im Allgemeinen keine Unterscheidung zwischen Beobachtung und Label statt. Aus diesem Grund wird vorerst angenommen, dass die Realisationen aller Zufallsvariablen aus der Menge $\mathcal{Y}$ stammen.

In diesem Kontext kann auch von der *Belegung-*, dem *Zustand-* oder dem *Label des Knotens* v gesprochen werden[1], wann immer eine Realisation $y \in \mathcal{Y}$ der Zufallsvariablen $v \in V$ gemeint ist. Für die Vereinigung zweier Knotenmengen $U, W \subseteq V$ wird $U + W$ und für ihre Exklusion $U - W$ geschrieben. Ein Querstrich über einer Knotenmenge bezeichnet ihr Komplement $\overline{U} = V - U$.

Für die simultane Realisation einer Menge $U \subseteq V$ von Zufallsvariablen wird $\mathbf{y}_U \in \mathcal{Y}^{|U|}$ geschrieben. Die Wahrscheinlichkeit einer solchen Realisation wird mit Hilfe der Randverteilung $p_{\boldsymbol{\theta}}(\mathbf{y}_U)$ (Gleichung (2.19)) berechnet, wobei $p_{\boldsymbol{\theta}}(\mathbf{y}_U)$ die Wahrscheinlichkeit einer beliebigen Realisation der Knoten in U bezeichnet. Die Berechnung der Randverteilung wird *Inferenz* genannt. Dazu müssen die Wahrscheinlichkeiten aller Realisationen $\mathbf{y}_V \in \mathcal{Y}^n$ aufsummiert werden, in denen die Knoten in U die Belegung $\mathbf{y}_U$ haben. Abbildung 2.2 soll die Notation simultaner Realisationen sowie deren Randverteilung veranschaulichen.

$$p_{\boldsymbol{\theta}}(\mathbf{y}_U) = \sum_{\mathbf{y}_{\overline{U}} \in \mathcal{Y}^{|\overline{U}|}} p_{\boldsymbol{\theta}}\left(\mathbf{y}_U, \mathbf{y}_{\overline{U}}\right) \tag{2.19}$$

[1] In [27] wird der Begriff *Färbung* verwendet, da dort die Realisationen der Zufallsvariablen den zufälligen Farben der Knoten entprechen und die Dichtefunktion die Wahrscheinlichkeit einer gegebenen Graphfärbung bestimmt.

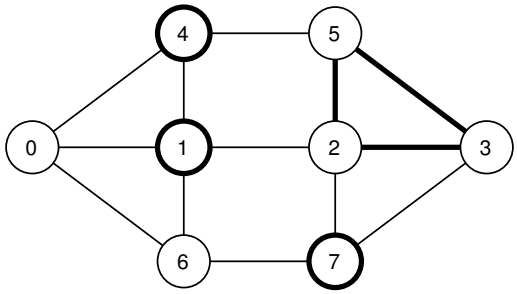


Abbildung 2.3: Ein Graph $G = (V, E)$ mit acht Knoten. Jeder Knoten $v_i \in V$ entspricht einer Zufallsvariablen, deren Realisation der i-ten Komponente der Realisation $\mathbf{y} := \mathbf{y}_V$ entspricht. Die fettgedruckten Kanten markieren die Clique $\{2, 3, 5\}$. Die fettgedruckten Knoten visualisieren ihre Nachbarschaft $\Delta(\{2, 3, 5\}) = \{1, 4, 7\}$.

Des Weiteren bezeichnet v sowohl den Knoten $v \in V$ als auch die Menge $\{v\} \subset V$, die nur diesen Knoten enthält. Für eine Knotenmenge $U \subseteq V$ heißt $\Delta(U)$ *Nachbarschaft* von U und enthält alle Knoten aus $\overline{U}$, die mit mindestens einem Knoten aus U benachbart sind (2.20).

$$\Delta(U) \quad := \quad \left\{ v : (v, u) \in E, v \in \overline{U}, u \in U \right\}. \tag{2.20}$$

Eine Menge von Knoten C heißt genau dann *Clique*, wenn alle Knoten in C benachbart sind. Eine Clique C ist per Definition vollständig in der Nachbarschaft jeder ihrer Knoten enthalten und es gilt die Teilmengenbeziehung (2.21).

$$C \quad \subseteq \quad v + \Delta(v), \forall v \in C \tag{2.21}$$

Abbildung 2.3 zeigt exemplarisch einen Graphen, in dem eine Clique sowie ihre Nachbarschaft hervorgehoben sind.

2.6 Markov Random Fields

In diesem Abschnitt wird eine spezielle Klasse von PGMs, die sog. *Markov Random Fields* (*MRF*), eingeführt. Für diese wird gezeigt, dass ihre Dichtefunktion aus den speziellen Eigenschaften eines MRFs hergeleitet werden kann.

Man unterscheidet gerichtete sowie ungerichtete Graphische Modelle. Spiegelt die Kantenmenge von G eine stochastische Abhängigkeitsstruktur der Zufallsvariablen wieder, wird G auch Abhängigkeitsgraph genannt. In korrekt spezifizierten Modellen entspricht die durch den Graph induzierte Abhängigkeitsstruktur tatsächlich derjenigen der Zufallsvariablen. Dies wird im Folgenden formalisiert.

Definition 2.2. Ein PGM $M = (G, p_{\boldsymbol{\theta}})$ hat genau dann die *Markov-Eigenschaft*, falls beliebige Mengen von Zufallsvariablen $U \subseteq V$, bei gegebener Realisation ihrer Nachbarknoten $\Delta(U)$, stochastisch unabhängig von allen anderen Knoten des Graphen sind. Damit gilt Gleichung (2.22).

$$p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\overline{U}}\right) \quad = \quad p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\Delta(U)}\right), \forall \mathbf{y}_U \in \mathcal{Y}^{|U|} \tag{2.22}$$

Hat ein PGM die Markov-Eigenschaft, so spricht man bei kreisfreien, gerichteten Graphen von *Bayes'schen Netzen* und im Fall von ungerichteten Graphen von *Markov-Netzwerken* (engl. *Markov Random Field*, MRF). Markov Random Fields wurden von Kindermann und Snell als Verallgemeinerung des Ising-Modells der statistischen Physik eingeführt [32, 36]. Obwohl die meisten der im Folgenden vorgestellten Ergebnisse ebenso für gerichtete Modelle gelten, werden diese hier

nicht explizit behandelt, da in dieser Diplomarbeit ausschließlich ungerichtete Modelle betrachtet werden.

Oft werden zwei Eigenschaften, nämlich lokale- und globale-Markov-Eigenschaft im Zusammenhang mit MRFs genannt, welche lediglich Spezialisierungen des Geltungsbereiches von Gleichung (2.22) darstellen.

- Gilt (2.22) für alle Knoten $u \in V$, so gilt die Markov-Eigenschaft *lokal*.
- Gilt (2.22) für alle Teilmengen $U \subseteq V$, so gilt die Markov-Eigenschaft *global*.

Beide Eigenschaften sind allerdings äquivalent. Beweise dafür sind zum Beispiel in [27] sowie in [25] zu finden. Bis jetzt wurde die Wahrscheinlichkeitsfunktion $p_{\boldsymbol{\theta}}$ stets als unbekannt angenommen. Allerdings besagt das folgende Theorem, dass die funktionale Form der Dichte durch die Semantik der Kantenmenge von MRFs bereits festgelegt wurde.

Theorem 2.3. *Hammersley–Clifford Theorem. Ein PGM $M = (G, p)$ hat genau dann die Markov Eigenschaft, wenn sich p entsprechend Gleichung (2.23) in Funktionen der Realisationen der Cliquen des Graphen G faktorisieren lässt. [27, S.20]*

$$p(\mathbf{y}) \quad \propto \quad \prod_{C \in \mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}_C\right)\right) \tag{2.23}$$

Mit diesem Theorem ist es klar, dass die Wahrscheinlichkeitsdichtefunktion eines MRFs proportional zu einer nicht-linearen Funktion der Realisationen der Cliquen des Graphen sein muss. Diese funktionale Repräsentation der Dichtefunktion legt den Grundstein für die Schätzung der Parameter von PGMs mit Markov Eigenschaft, zu denen auch CRFs gehören. Daher ist dieses Resultat essenziell für die Anwendung Graphischer Modelle im Bereich des Maschinellen Lernens und darüber hinaus. Da der Beweis mehrere Seiten umfasst und dort eine Notation verwendet wird, die für den Rest dieser Arbeit nicht relevant ist, findet sich dieser in Anhang C.

Nun steht fest, dass die Dichtefunktion von PGMs mit Markov-Eigenschaft die Form von Ungleichung (2.23) haben muss. Das in Abs. 2.3 erwähnte schwere Problem der Wahl einer passenden Dichtefunktion eines Probabilistischen Modells ist für MRFs so gut wie gelöst. Einzig die ϕ-Funktionen müssen noch genauer spezifiziert werden. Bis jetzt steht erst fest, dass der Wert einer Funktion ϕ_C von der Realisation der Clique C abhängen muss. Es liegt nahe, jeder möglichen Realisation $\mathbf{y}_C \in \mathcal{Y}^{|C|}$ ein *Gewicht* $\boldsymbol{\theta}_{\mathbf{y}_C}$ zuzuweisen und dieses als Wert der Funktion ϕ_C zurückzugeben, wodurch die Anforderungen des Theorems erfüllt wären. Um dies zu formalisieren werden in [41, 67] einfache Indikatorfunktionen 2.24 verwendet, die genau für eine Realisation den Wert 1 annehmen und ansonsten den Wert 0. Solche Indikatorfunktionen wurden bereits in Abs. 2.2 zur Formalisierung der Gütemaße verwendet.

$$1_{\mathbf{y}_C}\left(\mathbf{y}_C^{'}\right) = \begin{cases} 1 & \text{, falls } \mathbf{y}_C^{'} = \mathbf{y}_C \\ 0 & \text{, sonst} \end{cases} \tag{2.24}$$

Die Indikatorfunktionen einer Clique werden zu einer vektorwertigen Funktion $\mathbf{f}_C : \mathcal{Y}^{|C|} \rightarrow \{0,1\}^{|\mathcal{Y}|^{|C|}}$, dem so genannten *Merkmalsvektor* (engl. *feature vector*, *feature function*, *feature*) zusammengefasst, wobei jede Komponente des Vektors mit einer möglichen Realisation der Clique C korrespondiert. Der Vektor $\mathbf{f}_C(\mathbf{y}_C)$ hat ausschließlich an der Komponente der Realisation $\mathbf{y}_C$ eine 1 und ansonsten nur Nullen. Analog dazu wird für jede Clique C ein Gewichtsvektor $\boldsymbol{\theta}_C \in \mathbb{R}^{|\mathcal{Y}|^{|C|}}$ angelegt, der für jede mögliche Realisation der Clique C ein Gewicht enthält. Ein hohes Gewicht signalisiert, dass die Wahrscheinlichkeit dieser Realisation hoch ist und umgekehrt.

Realisation	$\mathbf{f}_C(\mathbf{y}_C)$	$\mathbf{f}_C(\mathbf{y'}_C)$	$\boldsymbol{\theta}_C$
$y_{v_1} = \mathrm{A}, y_{v_2} = \mathrm{A}$	0	0	0,35
$y_{v_1} = \mathrm{A}, y_{v_2} = \mathrm{B}$	1	0	0,25
$y_{v_1 1} = \mathrm{B}, y_{v_2} = \mathrm{A}$	0	0	1,45
$y_{v_1} = \mathrm{B}, y_{v_2} = \mathrm{B}$	0	1	−0,75

$$\phi_U(\mathbf{y}_C) = \langle \mathbf{f}_C(\mathbf{y}_C), \boldsymbol{\theta}_C \rangle = 0,25$$
$$\phi_C(\mathbf{y'}_C) = \langle \mathbf{f}_C(\mathbf{y'}_C), \boldsymbol{\theta}_C \rangle = -0,75$$

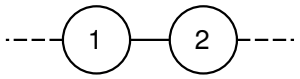


Tabelle 2.1: Beispielhafte Belegung zweier Merkmalsvektoren und eines Gewichtsvektors mit $C = \{v_1, v_2\}$ und $\mathcal{Y} = \{\mathrm{A}, \mathrm{B}\}$ für die Realisationen $\mathbf{y}_C = \{y_{v_1} = \mathrm{A}, y_{v_2} = \mathrm{B}\}$ sowie $\mathbf{y'}_C = \{y_{v_1} = \mathrm{B}, y_{v_2} = \mathrm{B}\}$. Die Vektoren enthalten $|\mathcal{Y}|^{|C|} = 4$ Einträge: einen für jede mögliche Realisation. Da jeder Vektor genau einen 1-Eintrag enthält, selektiert das Skalarprodukt genau ein Gewicht pro Realisation.

Gewichts- und Merkmalsvektoren der Cliquen werden identisch sortiert, das heißt wenn der k-te Eintrag eines Merkmalsvektors zur Realisation $\mathbf{y}'_C$ gehört, so enthält der k-te Eintrag des Gewichtsvektors $\boldsymbol{\theta}_C$ das Gewicht der Realisation $\mathbf{y}'_C$. Die Funktion ϕ_C entspricht dann dem Skalarprodukt dieser beiden Vektoren (2.25) und wird als *Potentialfunktion* bezeichnet. Die Funktionswerte von ϕ_C werden in diesem Kontext *Potentiale* genannt.

$$\phi_C(\mathbf{y}_C) = \langle \mathbf{f}_C(\mathbf{y}_C), \boldsymbol{\theta}_C \rangle \tag{2.25}$$

Die Modellparameter eines MRFs entsprechen also den Gewichten der Cliquenrealisationen (2.26). Diese können beispielsweise mit der Maximum-Likelihood Methode aus Abs. 2.4 geschätzt werden. Für den Rest dieses Kapitels wird aber vorübergehend angenommen, die Parameter wären bekannt. Das konkrete Vorgehen bei der Maximum-Likelihood Schätzung von $\boldsymbol{\theta}$ wird im folgenden Kapitel behandelt.

$$\boldsymbol{\theta} = (\boldsymbol{\theta}_C)_{C \in \mathcal{C}(G)} \tag{2.26}$$

Normalisierung. Die aus Theorem 2.3 gewonnene Funktion ist "nur" proportional zur gesuchten Dichte, da weder gefordert wurde, dass ihre Funktionswerte aus dem Intervall $[0, 1]$ stammen, noch dass sich ihre Funktionswerte zu 1 addieren. Damit stellt (C.35) im Allgemeinen kein valides Wahrscheinlichkeitsmaß dar. Dies kann allerdings erzwungen werden, indem die Funktion durch die Summe der Funktionswerte aller Realisationen dividiert wird. Der in (2.28) dargestellte Divisor Z heißt *Normalisierungskonstante*. Ist $\prod_{C \in \mathcal{C}(G)} \exp(\phi_C(\mathbf{y}_C))$ aufgrund einer geeigneten Wahl der Parameter bereits eine valide Dichtefunktion, so ist $Z = 1$ und die Normalisierung entfällt.

$$p_{\boldsymbol{\theta}}(\mathbf{y}) = \frac{1}{Z} \prod_{C \in \mathcal{C}(G)} \exp(\phi_C(\mathbf{y}_C)) \tag{2.27}$$

$$Z = \sum_{\mathbf{y}' \in \mathcal{Y}^n} p_{\boldsymbol{\theta}}\left(\mathbf{y}'\right) \tag{2.28}$$

Durch Anwendung der Normalisierung ergibt sich Gleichung (2.27) als Wahrscheinlichkeitsfunktion $p_{\boldsymbol{\theta}}$ von MRFs. Welche konkreten Werte die Dichtefunktion annimmt, hängt dabei nur vom Graphen G ab, da dieser die Cliquen enthält, wie auch von dem Gewichtsvektor $\boldsymbol{\theta}$. Zur Berechnung der Normalisierung muss offensichtlich eine Summe mit exponentiell vielen Summanden berechnet werden. Im folgenden Kapitel werden allerdings Algorithmen vorgestellt, welche die graphische Struktur ausnutzen, um diese Summe effizient zu berechnen.

Mit Gleichung (2.27) kann nun die gemeinsame Verteilung aller Knoten eines MRFs bestimmt werden. Dadurch ist es möglich, Aussagen über den Zustand des gesamten Modells zu treffen. Unter anderem kann gemäß Gleichung (2.29) eine wahrscheinlichste Realisation $\mathbf{y}^*$ berechnet werden, welche der Maximalstelle der Dichtefunktion entspricht. Dazu muss ebenfalls die Wahr-

scheinlichkeit einer exponentiellen Anzahl an Realisationen berechnet werden. Auch dazu werden im folgenden Kapitel effiziente Algorithmen vorgestellt.

$$\mathbf{y}^* = \arg\max_{\mathbf{y}\in\mathcal{Y}^n} p_{\boldsymbol{\theta}}(\mathbf{y}) \tag{2.29}$$

Das Problem der Wahl einer passenden Dichtefunktion ist also gelöst, sofern das PGM die Markov-Eigenschaft besitzt. Stattdessen stellt sich das Problem der Wahl eines geeigneten Abhängigkeitsgraphen. Wie zu Beginn dieses Abschnitts bereits erwähnt, stellt der Graph ein vereinfachtes Abbild realer Objekte dar. Wie dieser genau aussieht, hängt von der konkreten Lernaufgabe ab. Das Auffinden eines passenden Graphen ist dabei Teil der Modellselektion. Nach dem Wissen des Autors gibt es keine Methode um den "optimalen Abhängigkeitsgraphen" herzuleiten. Es existieren allerdings verschiedene Ansätze zur automatischen Generierung von Abhängigkeitsgraphen [59]. Diese bestehen aus der Anwendung statistischer Methoden zur Identifikation von Abhängigkeiten oder einer sukzessiven Modifikation der Kantenmenge eines Graphen. Trotzdem können auch solche Methoden nicht feststellen, ob überhaupt die richtigen Zufallsvariablen betrachtet werden. Unter der Annahme, dass der Graph des Modells bekannt wäre, können MRFs zu Lösung des Klassifikationsproblems mit strukturiertem Ausgaberaum verwendet werden. Dies wird im folgendem Abschnitt erläutert.

2.7 Klassifikation und Simulation mit MRFs

Mit den Resultaten des vorherigen Abschnitts sind alle Komponenten verfügbar, um das Klassifikationsproblem mit strukturiertem Ausgaberaum aus Abs. 2.2 mit MRFs zu lösen. Soll ein MRF zur Klassifikation eingesetzt werden, so wird die Knotenmenge V in die Mengen X und Y partitioniert, wobei X die sog. *beobachteten (*engl. *observed)* und Y die sog. *versteckten (*engl. *hidden)* Knoten enthält. Die Knoten in X entsprechen den Merkmalen einer zu klassifizierenden Beobachtung und diejenigen in Y den Komponenten eines strukturierten Labels. Jeder beobachtete Knoten $u \in X$ kann einen eigenen Wertebereich $\mathcal{X}_u$ besitzen, wobei auch hier (s. Abs. 2.2) zur Vereinfachung der Notation angenommen wird, dass alle Realisationen aus ein und derselben Menge $\mathcal{X}$ stammen. Der Wertebereich der Realisationen der versteckten Knoten sei $\mathcal{Y}$. Von nun an sei $\Delta(U)$ die Menge der versteckten Nachbarknoten der Knoten in U sowie $\tilde{\Delta}(U)$ die Menge der beobachteten Nachbarn. Durch diese Konvention ändert sich die Notation der Dichte eines MRFs (2.30).

$$p_{\boldsymbol{\theta}}(\mathbf{y},\mathbf{x}) = \frac{1}{Z_{XY}} \prod_{C\in\mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}_C, \mathbf{x}_{\tilde{\Delta}(C)}\right)\right) \tag{2.30}$$

Die Aufteilung in beobachtete und versteckte Knoten hat in MRFs keine formale Bedeutung. Hierbei handelt es sich um eine schlichte Umbennung einiger Knoten, damit die Notation besser zu der des Klassifikationsproblems passt. Dies ist in Abbildung 2.4 dargestellt. Die Tatsache das die Realisation der Knoten in X stets bekannt ist, wird von MRFs nicht abgebildet. Die Normalisierungskonstante aus Gleichung (2.31) ist äquivalent zu (2.28). Dabei bezeichnet die Schreibweise $\mathbf{y}_C$ eine Realisation aller versteckten Knoten einer Clique C sowie $\mathbf{x}_{\tilde{\Delta}(C)}$ die Realisationen der zu den Knoten in C benachbarten, beobachteten Knoten.

$$Z_{XY} = \sum_{\mathbf{x}'}\sum_{\mathbf{y}'} \prod_{C\in\mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}'_C, \mathbf{x}'_{\tilde{\Delta}(C)}\right)\right) \tag{2.31}$$

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{1}{Z_{XY}} \sum_{\mathbf{y}'} \prod_{C\in\mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}'_C, \mathbf{x}_{\tilde{\Delta}(C)}\right)\right) \tag{2.32}$$

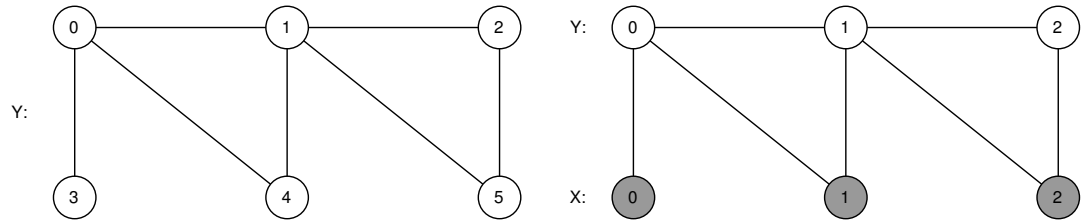


Abbildung 2.4: Auf der linken Seite ist ein sog. Linear-Chain MRF mit sechs Knoten dargestellt. Der rechte Graph zeigt das gleiche MRF, wobei die Knotenmenge in die Menge der versteckten Knoten (weiß) sowie die Menge der beobachteten Knoten (grau) partitioniert wurde. Dabei dient die Partitionierung der reinen Anschaulichkeit, die Dichtefunktionen der beiden MRFs sind identisch.

Mit Hilfe der Randverteilung der beobachteten Knoten (2.32) kann die Wahrscheinlichkeit einer Realisationen $\mathbf{x}$ berechnet werden. Mittels Division der Dichte (2.30) durch die Randverteilung erhält man die bedingte Verteilung der Label bei gegebenen Beobachtungen (2.33). Diese kann nun zur Klassifikation eingesetzt werden, indem gemäß (2.34) das Label mit maximaler Wahrscheinlichkeit vorhergesagt wird. Konkrete Algorithmen zur Berechnung von $\mathbf{y}^*$ werden in Abschnitt 3 vorgestellt.

$$p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{y},\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} \tag{2.33}$$

$$\mathbf{y}^* = \arg\max_{y\in\mathcal{Y}^n} \frac{p_{\boldsymbol{\theta}}(\mathbf{y},\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} \tag{2.34}$$

MRFs modellieren die gemeinsame Verteilung von beobachteten und versteckten Zufallsvariablen. Solche Modelle heißen *generativ*. Mit ihnen kann die Wahrscheinlichkeit der Beobachtungen bei gegebenen Labeln (Gleichung 2.35) berechnet werden. Damit können generative Modelle ohne weiteres zur Simulation der Beobachtungen benutzt werden. Zu den generativen Modellen zählen u.a. auch die *Hidden Markov Modelle* (HMM) [57].

$$\mathbf{x}^* = \text{argmax}_{x\in\mathcal{X}^n} \frac{p_{\boldsymbol{\theta}}(\mathbf{y},\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{y})} \tag{2.35}$$

Der Modellierungsaufwand ist bei diesen Modellen sehr hoch, da die Abhängigkeiten der beobachteten Knoten ebenfalls modelliert werden müssen. Die in Abschnitt 3 vorgestellten Conditional Random Fields vermeiden dies. Doch vorher wird eine alternative graphische Repräsentation für PGMs vorgestellt, die sich vor allem beim Algorithmenentwurf zur Berechnung der Randverteilung als nützlich erweisen wird.

2.8 Faktorgraphen

Abhängigkeitsgraphen stellen eine anschauliche Repräsentation der stochastischen Abhängigkeitsstruktur von multivariaten Zufallsvariablen dar. Die sog. Faktorgraphen [40, 67] sind auf die Darstellung der Faktorisierung einer Funktion ausgelegt. Dort repräsentiert die graphische Struktur den funktionalen Zusammenhang zwischen den einzelnen Zufallsvariablen.

Wird ein *Faktorgraph* $G = (V, F, E)$ als Basis eines PGMs verwendet, so werden die Knoten in $V = X \cup Y$ mit Zufallsvariablen identifiziert (vgl. [79, S.10]). Die Knoten in V heißen in diesem Kontext *Variablenknoten*, wobei die Knoten in Y ebenfalls (s.o.) als *versteckt* und die in X als *beobachtet* bezeichnet werden, mit $|Y| = n$ sowie $|X| = m$. Zusätzlich existiert eine zweite Knotenmenge F, welche die sog. *Faktorknoten* enthält. Der Graph G ist *bipartit* bezüglich V und F. Das heißt, es verläuft weder eine Kante innerhalb von V noch innerhalb von F. Die Kanten

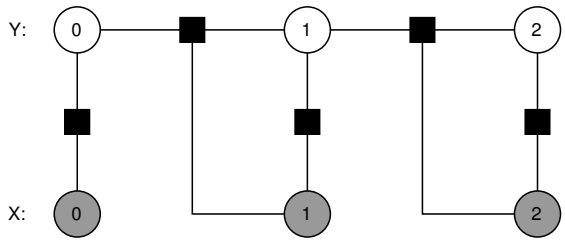


Abbildung 2.6: Ein Linear-Chain Faktorgraph. Faktorknoten werden als schwarze Quadrate dargestellt. Der dargestellte Faktorgraph ist äquivalent zu den Abhängigkeitsgraphen in Abbildung 2.4. Dabei repräsentiert jeder Faktorknoten eine Clique des Abhängigkeitsgraphen.

verlaufen ausschließlich zwischen den beiden Mengen. Ist $f \in F$ ein Faktorknoten, so bezeichnet $\Delta(f)$ die Menge der versteckten Nachbarknoten – die *versteckte Nachbarschaft* – sowie $\tilde{\Delta}(f)$ die Menge der beobachteten Nachbarknoten. Die Faktorknoten $f \in F$ werden mit Funktionen $f : \mathcal{Y}^{|\Delta(f)|} \times \mathcal{X}^{|\tilde{\Delta}(f)|} \to \mathbb{R}$ der Realisationen der zu ihnen adjazenten Variablenknoten identifiziert. Diese werden auch als *lokale Funktionen* bezeichnet.

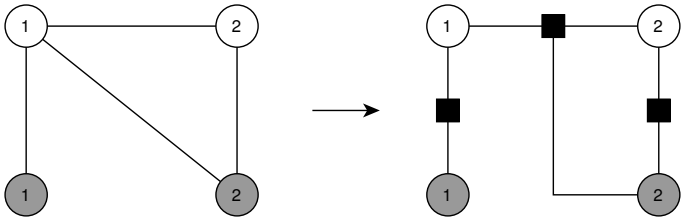


Abbildung 2.5: Transformation eines Abhängigkeits- (links) in einen Faktorgraphen (rechts). Die Cliquen der versteckten Knoten des Abhängigkeitsgraphen sind $\mathcal{C}(G) = \{\{1\}, \{2\}, \{1,2\}\}$. Aus den Cliquen werden drei Faktorknoten erzeugt die jeweils zu denselben beobachteten Knoten wie die korrespondierenden Cliquen adjazent sind.

Die Repräsentation von MRFs als Faktorgraph könnte im Kontext von Theorem 2.3 missverstanden werden, da Faktorgraphen per Definition keine Cliquen mit mehr als zwei Knoten enthalten können. Damit aber das Theorem für Faktorgraphen gilt, muss pro Clique C im Abhängigkeitsgraph ein Faktorknoten f existieren der zu den Variablenknoten in C adjazent ist. Die lokalen Funktionen der Faktorknoten $f \in F$ entsprechen in diesem Fall den exponenzierten Potentialfunktionen eines MRFs (2.36).

$$f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) = \exp\left(\phi_{\Delta(f)}\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right)\right) \tag{2.36}$$

Daher müssen alle Cliquen des Abhängigkeitsgraphen bekannt sein, um ein MRF mit Faktorgraph zu erzeugen, in dem das Hammersley-Clifford Theorem gilt. Das Wort "Clique" bezeichnet daher in dieser Diplomarbeit stets eine aus versteckten Knoten bestehende Clique des Abhängigkeitsgraphen und wird als Synonym für die versteckte Nachbarschaft eines Faktorknotens verwendet.

Abbildung 2.6 zeigt den aus Abb. 2.4 bekannten Linear-Chain Graph als Faktorgraph. In [79, S.11-13] wird eine Methode vorgestellt um beliebige Abhängigkeitsgraphen in äquivalente Faktorgraphen zu konvertieren (vgl. Abb. 2.5). Die Umkehrung gilt nicht, da ein Faktorknoten nicht notwendigerweise eine Clique des Abhängigkeitsgraphen repräsentieren muss. Nach Bishop [5, S.399-402] ist die Konvertierung zwischen Abhängigkeits- und Faktorgraph nicht eindeutig.

Neben der Anwendung von Faktorgraphen zur Modellierung von Wahrscheinlichkeitsdichtefunktionen existieren eine Vielzahl weiterer Anwendungsgebiete, wie z.B. der Berechnung von Paritätsfunktionen oder der Herleitung der schnellen Fourier-Transformation [40].

3 Conditional Random Fields

Dieses Kapitel handelt von den in dieser Diplomarbeit zentralen Modellen: *Conditional Random Fields* (CRF) [41]. CRFs gehören zu den Probabilistischen Graphischen Modellen und teilen sich viele Eigenschaften mit den aus dem vorherigen Kapitel bekannten MRFs. MRFs sind generative Modelle (s. Abs. 3) wohingegen CRFs zu den *diskriminativen* Modellen zählen. Abschnitt 3.1 zeigt die grundsätzlichen Unterschiede zwischen den beiden Modelltypen auf. In den darauf folgenden Abschnitten 3.2 bis 3.4 werden die Besonderheiten der CRF Merkmals- und Gewichtsvektoren erläutert, die am häufigsten für CRFs verwendeten graphischen Strukturen vorgestellt sowie einige wichtige Eigenschaften von CRFs erklärt. Zudem werden hier alle Algorithmen vorgestellt, auf denen die parallele Implementierung in Kapitel 5 basieren wird. Im Folgenden werden CRFs eingeführt und grundsätzliche Fragen im Bezug auf ihre Anwendung gestellt, die im Laufe dieses Kapitels beantwortet werden.

Definition 3.1. Ein PGM $M = (G, p)$ heißt *Conditional Random Field*, falls M bei gegebener Realisation der beobachteten Knoten $\mathbf{x}$ die Markov-Eigenschaft besitzt. Damit gilt Gleichung (3.1).

$$p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\overline{U}}, \mathbf{x}\right) \quad = \quad p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\Delta(U)}, \mathbf{x}\right), \forall \mathbf{y}_U \in \mathcal{Y}^{|U|} \tag{3.1}$$

Dabei sei G ein Abhängigkeits- oder äquivalenter Faktorgraph, dessen Knotenmenge in beobachtete und versteckte Knoten partitioniert ist.

Falls nicht anders angegeben, wird stets angenommen, dass G als Faktorgraph vorliegt und die versteckten Nachbarknoten eines Faktorknotens einer Clique des Abhängigkeitsgraphen entsprechen. Als Folge der Definition besitzen CRFs nur bei gegebener Realisation der beobachteten Knoten die Markov-Eigenschaft. Geht man also davon aus, dass diese Realisation stets gegeben ist, so lässt sich folgern, dass die Dichtefunktion von CRFs auf dieselbe Art und Weise wie diejenige von MRFs hergeleitet werden kann [41]. Demnach gilt das Hammersley-Clifford Theorem 2.3 und die Wahrscheinlichkeitsdichte von CRFs lässt sich entsprechend (3.2) faktorisieren. CRFs sind also "nur" dazu in der Lage, die Wahrscheinlichkeit der Label bei gegebenen Beobachtungen zu modellieren. Eine direkte Berechnung der Eintrittswahrscheinlichkeit einer Beobachtung ist mit CRFs daher nicht möglich.

Da die Realisation der beobachteten Knoten in CRFs stets gegeben ist, entfällt die Summation über diese bei der Berechnung der Normalisierung. Die Normalisierung $Z(\mathbf{x})$ von CRFs (3.3) ist damit einfacher zu berechnen als diejenige (Z_{XY}) von MRFs (2.31). Allerdings muss auch zur Berechnung von $Z(\mathbf{x})$ eine exponentielle Anzahl an Werten aufsummiert werden.

$$p_{\boldsymbol{\theta}}\left(\mathbf{y} \,|\, \mathbf{x}\right) \quad = \quad \frac{1}{Z\left(\mathbf{x}\right)} \prod_{f \in F} f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) \tag{3.2}$$

$$Z\left(\mathbf{x}\right) \quad = \quad \sum_{\mathbf{y}' \in \mathcal{Y}^n} \prod_{f \in F} f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) \tag{3.3}$$

Analog zu Rabiner [57] werden in diesem Kapitel drei fundamentale Problemstellungen in Bezug auf die Anwendung von CRFs behandelt:

1. Wie kann die Randverteilung einer Realisation $\mathbf{y}_{\Delta(f)} \in \mathcal{Y}^{|\Delta(f)|}$ effizient berechnet werden?
2. Wie kann die wahrscheinlichste Realisation aller versteckten Knoten für eine gegebene Realisation der beobachteten Knoten effizient berechnet werden?

3. Wie sollen die Modellparameter eines CRFs angepasst werden?

Die Beantwortung der ersten Frage ist eigentlich ein Vorrausgriff, da sich herausstellen wird, dass die effiziente Berechnung der Randverteilung ein essenzieller Bestandteil der Lösungen der beiden anderen Probleme darstellt. Algorithmen zur Lösung dieses Problems werden in Abs. 3.6 behandelt. Die zweite Frage ergibt sich aus der Tatsache, dass hier das Klassifikationsproblem mit strukturiertem Ausgaberaum gelöst werden soll. Entsprechende Algorithmen werden in Abs. 3.7 vorgestellt. Die Modellparameter von CRFs werden mit der aus Kapitel 2 bekannten Maximum-Likelihood Methode angepasst. Infolgedessen werden im letzten Abschnitt 3.8 verschiedene Algorithmen zur Maximierung der Likelihood-Funktion erläutert.

Die Frage nach einer effizienten Berechnung der Dichte $p(\mathbf{y} \mid \mathbf{x})$ wird hier nicht gestellt, da diese weder zum Training noch für die Klassifikation benötigt wird. Allerdings ist es möglich, diese mit den in Abs. 3.6 gezeigten Methoden "nebenbei" zu berechnen. An der entsprechenden Stelle wird der Vollständigkeit halber darauf hingewiesen.

Anstatt nach einem Weg zur effizienten Berechnung der Randverteilung zu fragen, fragte Rabiner nach der effizienten Berechnung der Wahrscheinlichkeit einer Beobachtung. Das diese Frage im Fall von CRFs nicht sinnvoll ist, wird der folgende Abschnitt über diskriminative Modelle zeigen.

3.1 Diskriminative Modelle

Die in Abschnitt 2.6 vorgestellten MRFs sind zwar zur Klassifikation geeignet, allerdings müssen dazu nicht nur die Abhängigkeiten der versteckten Knoten, sondern auch die der beobachteten Knoten modelliert werden. Dies erhöht den Modellierungsaufwand sowie die Komplexität der erforderlichen Berechnungen, obwohl ein Modell der Beobachtungen zur Klassifikation nicht benötigt wird. Falls das gelernte Modell ausschließlich zur Klassifikation eingesetzt werden soll, ist es zweckmäßig, die bedingte Verteilung $p(\mathbf{y}|\mathbf{x})$ direkt zu schätzen, anstatt den Umweg über (2.33) zu gehen. Modelle der bedingten Verteilung heißen *diskriminativ* [67]. Die Tatsache, dass jedes generative Modell ein diskriminatives Gegenstück besitzen muss, ist ein allgemeines Ergebnis von Ng und Jordan [48]. Abbildung 3.1 soll den Unterschied zwischen beiden Modellklassen visualisieren. Die erstmalig von Lafferty et al. [41] eingeführten Conditional Random Fields (CRF) zählen zu den diskriminativen Graphischen Modellen.

Anders ausgedrückt bedeutet dies, dass wenn das Ereignis $\mathbf{x}$ bereits eingetreten ist, die Abhängigkeiten zwischen den beobachteten Knoten bei der Berechnung der Eintrittswahrscheinlichkeit für $\mathbf{y}$ keine Rolle spielen [41, 67]. Nach Minka [45] müssen die Parameter $\boldsymbol{\theta}_g$ eines generativen Modells sowohl $p_{\boldsymbol{\theta}_g}(\mathbf{y} \mid \mathbf{x})$ als auch $p_{\boldsymbol{\theta}_g}(\mathbf{x})$ angemessen modellieren, um damit die gemeinsame Verteilung $p_{\boldsymbol{\theta}_g}(\mathbf{y}, \mathbf{x}) = p_{\boldsymbol{\theta}_g}(\mathbf{y} \mid \mathbf{x}) \cdot p_{\boldsymbol{\theta}_g}(\mathbf{x})$ zu berechnen. In diskriminativen Modellen der gemeinsamen Verteilung sind die Parameter dagegen aufgeteilt in die Parameter $\boldsymbol{\theta}_c$ der bedingten Verteilung $p_{\boldsymbol{\theta}_c}(\mathbf{y} \mid \mathbf{x})$ sowie die Parameter $\boldsymbol{\theta}_g$ der Verteilung von $\mathbf{x}$. Für die gemeinsame Verteilung gilt dann $p_{\boldsymbol{\theta}_c,\boldsymbol{\theta}_g}(\mathbf{y}, \mathbf{x}) = p_{\boldsymbol{\theta}_c}(\mathbf{y} \mid \mathbf{x}) \cdot p_{\boldsymbol{\theta}_g}(\mathbf{x})$, wobei die Parameter $\boldsymbol{\theta}_g$ in einem diskriminativen Modell nicht geschätzt werden. Das Training eines diskriminativen Modells sollte also einfacher als das eines generativen sein, da die implizite Nebenbedingung $\boldsymbol{\theta}_c = \boldsymbol{\theta}_g$ generativer Modelle in diskriminativen Modellen nicht existiert. Hierbei ist zu beachten, dass dies keinesfalls eine Unabhängigkeitsannahme für die beobachteten Knoten darstellt. Die Abhängigkeiten zwischen deren Realisationen dürfen beliebig komplex sein [41]. Der entsprechende Teil des Modells wird in CRFs lediglich nicht modelliert.

3.2 Merkmals- und Gewichtsvektoren

Aufgrund des oben beschriebenen Fakts, dass CRFs die bedingte Verteilung der Label direkt schätzen, unterscheiden sich ihre Merkmals - und Gewichtsvektoren von denjenigen von MRFs,

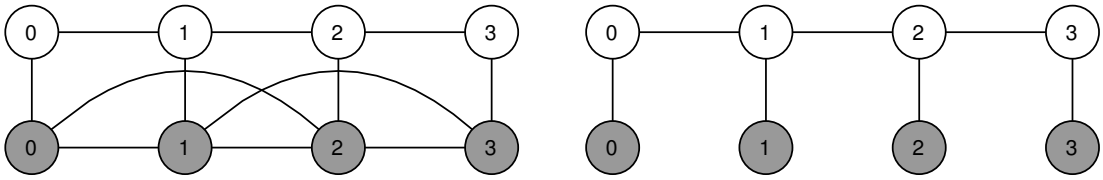


Abbildung 3.1: Dargestellt sind zwei Abhängigkeitsgraphen mit jeweils acht Knoten, wobei die Realisationen der weißen Knoten versteckt sind und die der grauen Knoten beobachtet wurden. Der linke Graph zeigt ein generatives Modell mit Abhängigkeiten zwischen den beobachteten Knoten. Der rechte Graph zeigt ein äquivalentes diskriminatives Modell, wobei die Abhängigkeiten zwischen den Beobachtungen nicht modelliert werden.

was an folgendem Beispiel erläutert wird. Angenommen ein Faktorgraph besitzt den Faktorknoten f mit den versteckten Nachbarn $\Delta(f) = \{v_1, v_2\} \subset Y$ sowie den beobachteten Nachbarn $\tilde{\Delta}(f) = \{v_3, v_4\} \subset X$. Die möglichen Realisationen der versteckten Knoten seien $\mathcal{Y} = \{\male, \female\}$ und die der beobachteten Knoten $\mathcal{X} = \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\}$. Dann enthält der Gewichtsvektor $\boldsymbol{\theta}_f$ des Faktorknotens in einem MRFs für jede der $|\mathcal{Y}|^{|\Delta(f)|} \cdot |\mathcal{X}|^{|\tilde{\Delta}(f)|} = 2^2 \cdot 4^2 = 64$ möglichen Realisationen der Nachbarn ein eigenes Gewicht. Das Modell "merkt" sich also, welche Realisationen alle Nachbarn von f gleichzeitig hatten. Das die Realisationen der Knoten v_3 und v_4 stets bekannt sind, fließt in diese Formalisierung nicht mit ein. Da die Abhängigkeiten der beobachteten Knoten in einem CRF nicht modelliert werden, kann die Information darüber welche Realisationen an den Knoten v_3 und v_4 simultan beobachtet wurde ignoriert werden. Daher enthält der Gewichtsvektor $\boldsymbol{\theta}_f$ eines CRFs für jede Realisation der Knoten v_3 und v_4 einen separaten Satz von $|\mathcal{Y}|^{|\Delta(f)|}$ Gewichten. Also insgesamt $\left|\tilde{\Delta}(f)\right| \cdot |\mathcal{X}| \cdot |\mathcal{Y}|^{|\Delta(f)|} = 2 \cdot 4 \cdot 2^2 = 32$. Selbiges gilt für den Merkmalsvektor $\mathbf{f}_f$ des Faktorknotens.

Allerdings sind in realen Anwendungen nur wenige der $\left|\tilde{\Delta}(f)\right| \cdot |\mathcal{X}| \cdot |\mathcal{Y}|^{|\Delta(f)|}$ möglichen Realisationen in der Trainingsmenge tatsächlich enthalten. Die Berücksichtigung unbeobachteter Realisationen führt laut Sutton und McCallum [67] zu sehr großen Modellen mit geringem Trainingsfehler. Realisationen die nicht in den Trainingsdaten enthalten sind, erhalten hohe negative Gewichte und werden somit "unwahrscheinlich". Ist die Trainingsmenge allerdings klein, so führt dieses Vorgehen zwangsweise zu einer Überanpassung an die Trainingsdaten, was wiederum zu hohen Testfehlern führen kann. Zur Vermeidung einer solchen Überanpassung sowie zur Reduktion der Speicherplatzkomplexität werden die Parameter unbeobachteter Realisationen in dieser Diplomarbeit nicht weiter betrachtet.

Eine Realisation $\mathbf{y}_{\Delta(f)}$ von versteckten Knoten heißt *bekannt unter* $\mathbf{x}_v$, falls die Realisation $\mathbf{x}_v \in \mathcal{X}$ am Nachbarknoten $v \in X$ von f beobachtet wurde, während die versteckten Knoten der Menge $\Delta(f)$ die Realisation $\mathbf{y}_{\Delta(f)}$ hatten. Ein Gewichtsvektor $\boldsymbol{\theta}_f$ eines CRF enthält das Gewicht $\boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)}, \mathbf{x}_v}$ also genau dann, wenn $\mathbf{y}_{\Delta(f)}$ unter $\mathbf{x}_v$ bekannt ist. Um nun das Potential $\ln f(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)})$ zu berechnen, werden $|\tilde{\Delta}(f)|$ Gewichte aufaddiert – eines für jede Realisation, eines jeden beobachteten Nachbarknotens des Faktorknotens f (3.4).

$$\ln f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right) = \sum_{v \in \tilde{\Delta}(f)} f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_v\right) \tag{3.4}$$

Tabelle 3.1 zeigt exemplarisch zwei CRF-Merkmalsvektoren, einen Gewichtsvektor sowie die Berechnung zweier Potentiale. Die Potentialfunktionen eines CRFs und mit ihnen die Dichtefunktion $p_{\boldsymbol{\theta}}(\mathbf{y} \,|\, \mathbf{x})$ werden also aus den Gewichten einzelner Teilstrukturen "gemischt". Zusammen mit der Normalisierungskonstante offenbart dies den konditionalen Charakter von Conditional Random Fields. Da die Realisationen der beobachteten Knoten die Auswahl der Parameter und damit die konkrete Verteilung der Label steuern, ist ein CRF auf die Realisationen dieser Knoten

konditioniert.

Index	Vers. Realisation	Beob. Realisation	$\mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right)$	$\mathbf{f}_f\left(\mathbf{y}'_{\Delta(f)}, \mathbf{x}'_{\tilde{\Delta}(f)}\right)$	$\boldsymbol{\theta}_f$
0	$v_1 = ♂, v_2 = ♂$	$v_3 = ♡$	0	0	0,35
1	$v_1 = ♂, v_2 = ♀$	$v_3 = ♡$	**1**	0	0,25
2	$v_1 = ♀, v_2 = ♀$	$v_3 = ♡$	0	0	−0,75
3	$v_1 = ♀, v_2 = ♂$	$v_3 = ♡$	0	0	0,55
4	$v_1 = ♀, v_2 = ♀$	$v_3 = ♢$	0	**1**	−0,09
5	$v_1 = ♂, v_2 = ♀$	$v_4 = ♡$	**1**	0	0,5
6	$v_1 = ♀, v_2 = ♀$	$v_4 = ♡$	0	0	−0,8
7	$v_1 = ♀, v_2 = ♀$	$v_4 = ♣$	0	**1**	0,4

$$\ln f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) = \left\langle \mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_f \right\rangle = 0,25 + 0,5 = 0,75$$
$$\ln f\left(\mathbf{y}'_{\Delta(f)} \,|\, \mathbf{x}'_{\tilde{\Delta}(f)}\right) = \left\langle \mathbf{f}_f\left(\mathbf{y}'_{\Delta(f)}, \mathbf{x}'_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_f \right\rangle = -0,09 + 0,4 = 0,31$$

Tabelle 3.1: Beispielhafte Belegung zweier CRF-Merkmalsvektoren und des korrespondierenden Gewichtsvektors für den Faktorknoten f mit $\Delta(f) = \{v_1, v_2\}$ sowie $\tilde{\Delta}(f) = \{v_3, v_4\}$ aus Abb. 3.2 (links). Die Menge der Label ist $\mathcal{Y} = \{♂, ♀\}$ und die der beobachteten Realisationen $\mathcal{X} = \{♢, ♡, ♠, ♣\}$. Es seien $\mathbf{y}_{\Delta(f)} = \{v_1 = ♂, v_2 = ♀\}$, $\mathbf{x}_{\tilde{\Delta}(f)} = \{v_3 = ♡, v_4 = ♡\}$ sowie $\mathbf{y}'_{\Delta(f)} = \{v_1 = ♀, v_2 = ♀\}$, $\mathbf{x}'_{\tilde{\Delta}(C)} = \{v_3 = ♢, v_4 = ♣\}$. Jeder Merkmalsvektor enthält pro beobachtetem Knoten einen 1-Eintrag. Das Potential der Realisationen entspricht dann der Summe der durch die Merkmalsvektoren selektierten Gewichte.

In praktischen Anwendungen sind die Faktorknoten mit einer Vielzahl an beobachteten Knoten verbunden (Abb. 3.2, mitte). Es fällt auf, dass viele beobachtete Knoten mit denselben Faktorknoten adjazent sind. Dies wird beim Entwurf der Datenstrukturen in Kapitel 5 ausgenutzt, um eine platzsparende Repräsentation der beobachteten Realisationen zu erhalten. Dazu werden alle beobachteten Knoten gleicher Adjazenz zu einem *Aggregat* zusammengefasst, so dass anstatt Abb. 3.2 (mitte), die in Abb. 3.2 (rechts) gezeigte Darstellung entsteht. Dies kann so interpretiert werden, dass in (3.4) nicht über die verschiedenen beobachteten Nachbarknoten, sondern stattdessen über mehrere Realisationen ein und desselben Nachbarknotens iteriert wird.

Falls die Knoten eines CRF nummeriert sind, ist es üblich [41], sowohl versteckte als auch beobachtete Knoten mit denselben Indizes zu nummerieren. In 3.2 (rechts) ist es daher kein Versehen, dass der dort gezeigte Faktorknoten mit dem versteckten Knoten $v_1 \in Y$ sowie dem beobachteten Knoten $u_1 \in X$ verbunden ist.

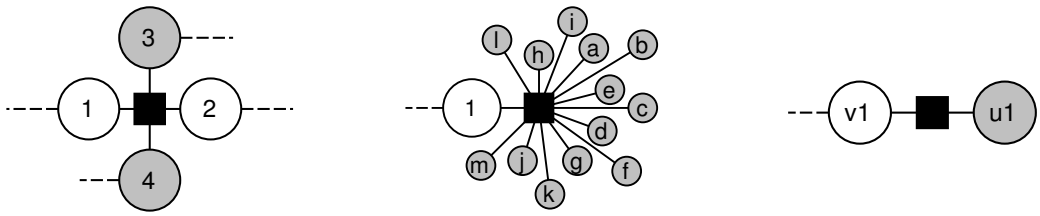


Abbildung 3.2: Links: Ein Faktorknoten (schwarzes Rechteck) mit zwei benachbarten versteckten Knoten (weiß) sowie zwei benachbarten beobachteten Knoten (grau). Mitte: Ein Faktorknoten mit 13 beobachteten Knoten sowie einem versteckten. Rechts: Ein Faktorknoten mit einem beobachteten und einem versteckten Knoten. Die gestrichelten Kanten deuten weitere Nachbarn der Knoten an.

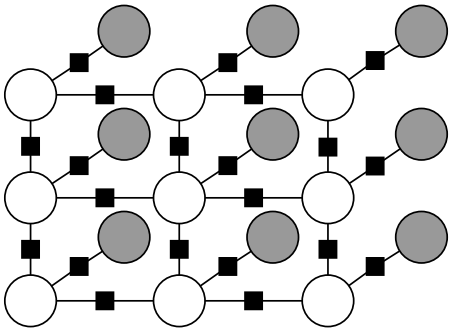


Abbildung 3.3: Dargestellt ist ein Gitter Graph mit neun versteckten sowie neun beobachteten Knoten. Die Abhängigkeiten zwischen den beobachteten Knoten (grau) können beliebig komplex sein, ohne im Modell erfasst sein zu müssen. Repräsentiert die Beobachtung $\mathbf{x}$ beispielsweise ein Foto, so können die observablen Knoten beliebige Bildinformationen aller Pixel des gesamten Fotos beinhalten. Damit wären ihre Realisationen sicherlich abhängig voneinander, diese teilweise komplexen Abhängigkeiten müssen in einem CRF aber nicht modelliert werden.

3.3 Parameter Tying

Abhängig von der konkreten Lernaufgabe kann es sinnvoll sein, die Parameter einiger Faktorknoten zusammenzufassen. Diese Technik wird *Parameter Tying* genannt. Eine Menge von Cliquen, die denselben Parametervektor verwenden, wird von Sutton und McCallum [67] auch als *Cliquen-Template* bezeichnet. Da hier hauptsächlich Faktorgraphen zur Darstellung verwendet werden, wird stattdessen die Bezeichnung *Faktor-Template* oder einfach nur *Template* verwendet. Dazu wird jedem Faktorknoten $f \in F$ genau ein Template $\mathcal{F}_h \in \mathcal{F}$ zugewiesen, wobei alle Faktorknoten des h-ten Templates denselben Parametervektor verwenden (s. Gleichung 3.5). Die Anzahl der Templates ist dabei frei wählbar. Gilt beispielsweise $|F| = |\mathcal{F}|$, so besitzt jeder Faktorknoten einen eigenen Parametervektor. In den meisten Anwendungen wird der gleiche Parametersatz für alle Faktorknoten mit gleicher Anzahl versteckter Nachbarknoten verwendet. In diesem Fall gilt $\Delta(f) = \Delta(f') \Rightarrow \boldsymbol{\theta}_f = \boldsymbol{\theta}_{f'}$. Formal wird dieser Unterschied erst in Abschnitt 3.8 sichtbar, wenn es darum geht, die Parameter des Modells mit der Maximum-Likelihood Methode zu schätzen.

$$f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right) = \left\langle \mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_{\mathcal{F}_h} \right\rangle \Leftrightarrow f \in \mathcal{F}_h \tag{3.5}$$

3.4 Graphische Strukturen

Eine der beliebtesten graphischen Strukturen von CRFs ist die einer linearen Liste (Abb. 2.6). Sie wird immer dann verwendet, wenn ein zeitdiskreter Prozess modelliert werden soll. In diesem Fall werden die Knoten aufsteigend nummeriert und anstatt mit v (engl. *vertex*) mit t (engl. *time*) indiziert. Eine Realisation $\mathbf{y}_t$ entspricht dann dem Label eines versteckten Knotens zum Zeitpunkt t und $\mathbf{x}_t$ enthält die Realisation der beobachteten Nachbarknoten. Diese Struktur wird insbesondere zur Verarbeitung natürlicher Sprache, sowohl von Text- [67], als auch von Audiodaten [57] verwendet.

Anstatt einer zeitlichen Abhängigkeitsstrukture kann ebenso eine räumliche modelliert werden. Für 2-dimensionale Daten wird die graphische Struktur eines Gitters verwendet. Abbildung 3.3 zeigt einen solchen Gitter-Graphen, der vor allem bei der Klassifikation von Bilddaten eingesetzt wird.

Ist die graphische Struktur des Modells festgelegt, so kann diese explizit in der Dichte von CRFs berücksichtigt werden. Hat der Graph zum Beispiel die Form von Abbildung 3.1 (rechts), so ergibt sich die Dichtefunktion in Gleichung (3.6). Sind dagegen die Kanten ebenfalls von den

beobachteten Knoten abhängig (Abb. 2.6), so ergibt sich die Dichte als Gleichung (3.7). Das Subskript eines Faktorknotens $f_{\{t-1,t\}}$ entspricht hier den Indizes seiner versteckten Nachbarn.

$$p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^{|\mathbf{y}|} f_t(\mathbf{y}_t \mid \mathbf{x}_t) \cdot f_{\{t-1,t\}}(\mathbf{y}_{t-1}, \mathbf{y}_t) \tag{3.6}$$

$$p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^{|\mathbf{y}|} f_t(\mathbf{y}_t \mid \mathbf{x}_t) \cdot f_{\{t-1,t\}}(\mathbf{y}_{t-1}, \mathbf{y}_t \mid \mathbf{x}_t) \tag{3.7}$$

Beide Varianten sind in der Literatur als Linear-Chain CRF bekannt, wobei in dieser Diplomarbeit aus Gründen der Eindeutigkeit ausschließlich Modelle mit Dichtefunktion (3.7) *Linear-Chain CRF* genannt werden. Modelle mit Dichte (3.6) werden nach [67] als *HMM-ähnliches Linear-Chain CRF* bezeichnet. Die Bezeichnung ist auf die sog. *Hidden-Markov-Modelle* [57] zurückzuführen. Sie entsprechen einer generativen Variante von Linear-Chain CRFs und vernachlässigen zur Komplexitätsreduktion die Abhängigkeiten zwischen einem Zustandsübergang $(t-1) \to t$ und der korrespondierenden Beobachtung x_t. Hierbei ist zu beachten, dass x_t in HMMs tatsächlich eine einzelne Beobachtung und keinen Vektor von Beobachtungen bezeichnet, da in generativen Modellen die Abhängigkeiten zwischen einzelnen Beobachtungen im Vektor $\mathbf{x}_t$ modelliert werden müssen. Daher wird üblicherweise darauf verzichtet, eine Vielzahl von Attributen zu verwenden. Bei Rabiner [57] ist daher auch von *dem beobachteten Symbol* die Rede.

In Linear-Chain CRFs kommt das oben erwähnte Parameter-Tying für Faktorknoten mit gleicher Anzahl versteckter Nachbarn zum Einsatz. Das heißt unabhängig davon "wann" eine Beobachtung gemacht wurde, wird stets derselbe Gewichtsvektor verwendet.

In den Gleichungen (3.6) und (3.7) bezeichnet $|\mathbf{y}|$ nicht etwa die Norm von $\mathbf{y}$, sondern die Dimension des Labels, also die Anzahl der versteckten Knoten. Dies soll verdeutlichen, dass jede Realisation (jedes Beispiel), wie im vorangegangenen Abschnitt erwähnt, aus einer variablen Anzahl von Labeln bestehen kann.

3.5 Strukturen mit variabler Länge

Obwohl zur Berechnung der Dichtefunktion (3.2) formal über alle Faktorknoten des Graphen iteriert wird, kommt es in praktischen Anwendungen häufig vor, dass die Trainingsbeispiele aus einer unterschiedlichen Anzahl an Knoten bestehen. Dies wird zur Vereinfachung der Notation jedoch nur selten explizit berücksichtigt. Beispielsweise wird von Lafferty angenommen, dass jedes Beispiel aus n Knoten besteht. Sutton und McCallum [67] gehen davon aus, jede der dort betrachteten Sequenzen hätte die Länge T. Allerdings sind diese Annahmen implizit. Der Fakt, dass jedes Beispiel aus einer unterschiedlichen Anzahl an Knoten bestehen kann, wird in keiner der beiden Veröffentlichungen formal berücksichtigt.

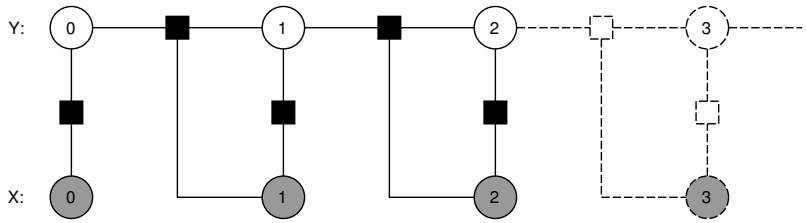


Abbildung 3.4: Ein Linear-Chain Subgraph. Die inaktiven Knoten sowie die zu ihnen inzidenten Kanten sind gestrichelt dargestellt. Das eigentliche CRF enthält zwar Parameter zur Gewichtung des versteckten Knotens v_3, allerdings ist dieser in dem Graphen des hier gezeigten Beispiels inaktiv.

Anstatt in der Dichtefunktion also über alle Faktorknoten $F(\mathbf{y}, \mathbf{x})$ der Realisation $\mathbf{y}, \mathbf{x}$ zu iterieren, wird hier angenommen, dass alle Beispiele zusammenhängende Subgraphen ein und desselben Graphen sind. Die Knoten, die in einem Beispiel nicht vorhanden sind, werden bei der Berechnung einfach ausgelassen. Solche ausgelassenen Knoten heißen *inaktiv*, ansonsten *aktiv*. Diese Situation ist in Abbildung 3.4 dargestellt. Die Gewichte der Realisationen der inaktiven Knoten werden konstant auf 0 gesetzt. Durch diese Annahme ist der Fall unterschiedlich langer Beispiele angemessen abgedeckt, ohne die formale Notation anpassen zu müssen. Es wird also stets über die Faktorknoten der Menge F iteriert und angenommen, dass diese für alle Realisationen bzw. Beispiele dieselbe ist. Dies wird für den Entwurf der Datenstrukturen und Algorithmen in Kapitel 5 von Bedeutung sein.

3.6 Berechnung der Randverteilung

In diesem Abschnitt wird die erste zu Anfang des Kapitels erwähnte Problemstellung – die effiziente Berechnung der Randverteilung – behandelt. Dazu wird ein allgemeiner Inferenzalgorithmus – die *Belief-Propagation* (BP) – angegeben. Dieser kann für CRFs mit verschiedensten graphischen Strukturen eingesetzt werden. Es werden BP Varianten für allgemeine Faktorgraphen, kreisfreie Faktorgraphen (Faktorbäume) sowie Linear-Chain Graphen (*Forward-Backward* Algorithmus, FB) erläutert. Diese Inferenzalgorithmen werden in Kapitel 5 parallelisiert und in Kapitel 6 evaluiert.

Die Evaluation der parallelen Algorithmen in Kapitel 6 erfolgt ausschließlich auf Linear-Chain Datensätzen. Trotzdem werden hier allgemeinere Algorithmen in Betracht gezogen, da sowohl Bishop [5] als auch Kschischang [40] darauf hinweisen, dass die allgemeinste BP Variante (*Loopy-Belief-Propagation*, LBP) eine inhärent parallele Struktur aufweist. Der *Forward-Backward* Algorithmus besitzt zwar grundsätzlich eine sequenzielle Struktur, wird aber dennoch hier betrachtet, da dieser die Standardmethode zur Berechnung der Randverteilung in Linear-Chain Graphen darstellt [41, 57, 67]. Die Variante der Belief-Propagation für Bäume wird ebenfalls betrachtet, da diese einen notwendigen Schritt auf dem Weg zu LBP darstellt.

Neben BP existieren weitere Algorithmen zur Inferenz in Faktorgraphen. Die Mean-Field Berechnung [79], das Gibbs-Sampling [24], der Variablen-Eliminationsalgorithmus [80] sowie der Junction-Tree-Algorithmus [42]. Grundsätzlich könnte jeder dieser Algorithmen im Hinblick auf seine Parallelisierbarkeit untersucht werden. Da die Behandlung aller Verfahren den Rahmen einer Diplomarbeit überschritten hätte, wurden hier aufgrund der oben genannten Bemerkungen von Bishop und Kschischang die Varianten der Belief-Propagation gewählt.

Wie am Anfang des Kapitels erwähnt, wird die Randverteilung sowohl zur Klassifikation als auch zur Anpassung der Gewichte benötigt. Betrachtet man die formale Definition der Randverteilung (2.19), so ist nicht unmittelbar klar, wie diese effizient berechnet werden soll. Zum einen summiert die dort gezeigte Summe exponentiell viele Terme auf, zum anderen muss zur Berechnung der Dichte $p_{\boldsymbol{\theta}}\left(\mathbf{y}_U, \mathbf{y}_{\overline{U}} \,|\, \mathbf{x}\right)$ die Normalisierung $Z(\mathbf{x})$ berechnet werden.

Abbildung 3.5: Schematische Darstellung der Berechnung von Faktor- und Variablennachrichten. Dabei ist zu beachten, dass ein Nachrichtenaustausch ausschließlich zwischen versteckten- und Faktorknoten stattfindet, da die Belegung der beobachteten Knoten gegeben ist und daher auch keine Wahrscheinlichkeiten für ihre möglichen Realisationen berechnet werden müssen.

$$p_{\boldsymbol{\theta}}\left(\mathbf{y}_U \,|\, \mathbf{x}\right) = \sum_{\mathbf{y}_{\overline{U}} \in \mathcal{Y}^{|\overline{U}|}} p_{\boldsymbol{\theta}}\left(\mathbf{y}_U, \mathbf{y}_{\overline{U}} \,|\, \mathbf{x}\right), U \subset V \tag{3.8}$$

Als Alternative zur naiven Summation aller möglichen Labelkombinationen wird im Folgenden die Belief-Propagation [53] vorgestellt. Mit dieser können die Randverteilungen sowie die Normalisierung von CRFs in einer von der graphischen Struktur abhängigen Laufzeit berechnet werden. Eine ausführliche Herleitung des Algorithmus ist in [5, 40] sowie in [53] zu finden.

Belief-Propagation. Sei $M = (G, p_{\boldsymbol{\theta}})$ ein CRF mit Faktorgraph $G = (V, F, E)$, $\mathbf{x}$ eine beliebige, aber feste, Realisation der observablen Knoten und $\mathbf{y}$ eine Realisation der versteckten Knoten. Die Belief-Propagation besteht im Wesentlichen aus der abwechselnden Berechnung zweier Funktionen, den sog. *Faktor-* und *Variablennachrichten*. Der Nachrichtenaustausch findet nur zwischen Faktorknoten und versteckten Knoten statt. Eine das Label $y \in \mathcal{Y}$ betreffende Nachricht $m_{f \to v}\left(y \,|\, \mathbf{x}\right)$ eines Faktorknotens $f \in F$ an seinen versteckten Nachbarknoten $v \in \Delta\left(f\right)$ bei gegebener Beobachtung $\mathbf{x}$, wird entsprechend Gleichung (3.9) als Summe der Produkte aus der lokalen Funktion des Faktorknotens sowie den Variablennachrichten der anderen versteckten Nachbarknoten gebildet. Aufgrund dieser funktionalen Form wird die Belief-Propagation auch als *Summe-Produkt-Algorithmus* [5, 40] bezeichnet. Die Laufzeit zur Berechnung einer Faktornachricht wird durch die Summation dominiert und beträgt $\mathcal{O}(|\mathcal{Y}|^{\Delta_{max}})$. Hier ist zu beachten, dass die Summation tatsächlich über alle möglichen Realisationen der anderen versteckten Nachbarn eines Faktorknotens verläuft und nicht nur über die bekannten Realisationen. Die lokalen Funktionen $f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right)$ können zwar nur für die bekannten Realisationen berechnet werden, allerdings existieren stets eingehende Nachrichten der versteckten Nachbarn bezüglich aller Label aus $\mathcal{Y}$. Daher muss das Produkt in (3.9) für jede Kombination der Label berechnet werden. Eine Variablennachricht (3.10) eines versteckten Knotens v an seinen benachbarten Faktorknoten f entspricht dem Produkt aller eingehenden Nachrichten der ebenfalls mit v benachbarten Faktorknoten.

$$m_{f \to v}\left(y \,|\, \mathbf{x}\right) = \sum_{\mathbf{y'}_{\Delta(f)-v}} f\left(v = y, \mathbf{y'}_{\Delta(f)-v} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) \prod_{u \in \Delta(f)-v} m_{u \to f}\left(\mathbf{y'}_u \,|\, \mathbf{x}\right) \tag{3.9}$$

$$m_{v \to f}\left(y \,|\, \mathbf{x}\right) = \prod_{g \in \Delta(u)-f} m_{g \to v}\left(y \,|\, \mathbf{x}\right) \tag{3.10}$$

Die Nachricht eines Faktorknotens f an einen versteckten Knoten v kann als "kleine Randverteilung" verstanden werden, da das Label v festgehalten wird und die Summation über alle möglichen Realisationen der anderen benachbarten versteckten Knoten verläuft. Diese Berechnung

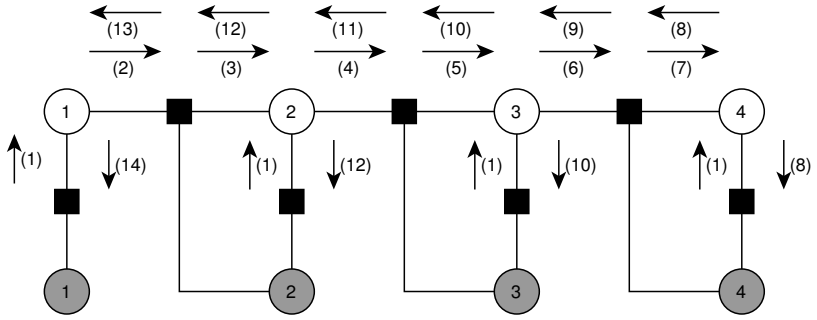


Abbildung 3.6: Dargestellt ist der Nachrichtenfluss beim seriellen Scheduling einer Belief-Propagation auf einem Linear-Chain CRF mit vier versteckten Knoten. Die Pfeile geben die Richtungen an, in denen die Nachrichten verschickt werden. Die Zahl in Klammern steht für die Iteration ab der die Berechnung der entsprechenden Nachricht möglich ist.

wird für alle möglichen Label von v durchgeführt. Eine Variablennachricht kann dementsprechend als "kleine Dichtefunktion" interpretiert werden, da sie dem Produkt der Faktornachrichten – also dem Produkt der "kleinen Randverteilungen" – entspricht. Eine Nachricht zu verschicken bedeutet schlicht, dass eine neu berechnete Nachricht eine eventuell zuvor berechnete Nachricht ersetzt. Man sagt auch, dass die Nachrichten durch den Graphen *propagiert* werden. Das Verschicken der Faktor- und Variablennachrichten wird für eine von der graphischen Stuktur abhängigen Anzahl (s.u.) an Iterationen wiederholt. Dabei ist zu beachten, dass sich die Werte der Potentialfunktionen während dieser Iterationen nicht ändern. Daher müssen diese nur einmal vor Beginn der Belief-Propagation berechnet werden, anschließend kann jede Berechnung der Faktornachrichten auf die gespeicherten Potentiale zurückgreifen. Nach der letzten Iteration kann der *Belief* (3.12) eines Knotens bezüglich eines Labels, als Produkt aller bezüglich dieses Labels eingehenden Faktornachrichten berechnet werden. Analog dazu wird der Belief von Faktorknoten (3.11) als Produkt der entsprechenden Variablennachrichten, multipliziert mit der lokalen Funktion f, berechnet.

$$b_f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}\right) = f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) \prod_{v \in \Delta(f)} m_{v \to f}\left(\mathbf{y}_v \,|\, \mathbf{x}\right) \tag{3.11}$$

$$b_v\left(y \,|\, \mathbf{x}\right) = \prod_{f \in \Delta(v)} m_{f \to v}\left(y \,|\, \mathbf{x}\right) \tag{3.12}$$

Ist der Graph des CRF kreisfrei, so stimmen die normalisierten Beliefs (3.13, 3.14) exakt mit den entsprechenden Randverteilungen überein [40, 53, 79]. In diesem Fall bedeutet *kreisfrei*, dass der Nachrichtenfluss keinen Kreis enthält. Formal besitzt ein Graph, in dem mehrere Faktorknoten zu demselben beobachteten Knoten adjazent sind, zwar Kreise, jedoch erfolgt der Nachrichtenaustausch ausschließlich zwischen Faktorknoten und versteckten Knoten. Daher werden solche Graphen hier auch als kreisfrei bezeichnet. Die Beliefs werden in Abs. 3.7 zur Klassifikation verwendet. Wie die Normalisierungen berechnet werden, wird im folgenden Abschnitt erläutert.

$$p\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}\right) = \frac{1}{Z\left(\mathbf{x}\right)} b\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}\right) \tag{3.13}$$

$$p\left(v = y \,|\, \mathbf{x}\right) = \frac{1}{Z\left(\mathbf{x}\right)} b_v\left(y \,|\, \mathbf{x}\right) \tag{3.14}$$

3.6.1 Inferenz in kreisfreien Graphen

Die Berechnung der Randverteilungen sowie der Normalisierung $Z(\mathbf{x})$ ist abhängig von der konkreten Reihenfolge, in der die Nachrichten verschickt werden. Die Reihenfolge in der die Knoten ihre Nachrichten verschicken und empfangen heißt *Scheduling* oder auch *Schedule*. Es sind verschiedene Arten von Schedulings denkbar. In kreisfreien Graphen (Bäumen) liefert beispielsweise das sog. *serielle* Scheduling die exakten Randverteilungen. Dabei werden die Nachrichten eines Knotens berechnet, sobald auf allen mit ihm inzidenten Kanten, mit Ausnahme einer Kante, neue Nachrichten eingegangen sind. Da Blätter per Definition nur eine Kante besitzten, kann die Berechnung an diesen sofort beginnen. Die ausgehende Nachricht eines Variablenblatts wird konstant auf 1 gesetzt. Ausgehende Nachrichten von Faktorblättern entsprechen einfach ihrer lokalen Funktion, da keine weiteren Nachbarn existieren, deren Nachrichten multipliziert werden könnten. Jeder innere Knoten w (w kann sowohl Variablen- als auch Faktorknoten sein) befindet sich solange im *Leerlauf*, bis auf allen mit ihm inzidenten Kanten $e \in E$, mit Ausnahme einer Kante $\{w, q\} \in E$, neue Nachrichten eingegangen sind. Anschließend können die Nachrichten $m_{w \to q}(y \mid \mathbf{x})$ für alle $y \in \mathcal{Y}$ berechnet und verschickt werden. Daraufhin befindet sich w erneut solange im Leerlauf, bis eine Nachricht auf der Kante $\{w, q\}$ eintrifft. Nun können die Nachrichten an alle anderen Nachbarn von w berechnet werden. Der Algorithmus terminiert, sobald über jede Kante des Graphen zwei Nachrichten für jedes $y \in \mathcal{Y}$ verschickt wurden [40, S.502]. Abbildung (3.6) veranschaulicht dieses Scheduling am Beispiel eines Linear-Chain CRF. Die Normalisierung $Z(\mathbf{x})$ kann nach der Terminierung der seriellen Belief-Propagation an jedem Knoten w, als Summe all seiner Beliefs berechnet werden. Dabei spielt es keine Rolle, ob w ein Faktor- oder ein Variablenknoten ist. Die Laufzeit zur Berechnung der Randverteilungen sowie der Normalisierung beträgt somit $\mathcal{O}(|E|\,|\mathcal{Y}|^{\Delta_{max}})$. Durch Einsetzen der Definition des Beliefs (3.11 bzw. 3.12) in Gleichung (3.15) folgt $Z_w(\mathbf{x}) = Z(\mathbf{x})$.

$$Z_v(\mathbf{x}) = \sum_{y \in \mathcal{Y}} b_v(y \mid \mathbf{x}), v \in Y \tag{3.15}$$

$$Z_f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathcal{Y}^{|\Delta(f)|}} b_f\left(\mathbf{y}_{\Delta(f)} \mid \mathbf{x}\right), f \in F \tag{3.16}$$

Die Gleichungen (3.17) bis (3.20) zeigen die Herleitung der Normalisierung am Beispiel des in Abbildung 3.7 dargestellten Faktorgraphen. Die Faktorknoten werden im Folgenden mit ihren versteckten Nachbarschaften indiziert, da sie so innerhalb des Graphen eindeutig identifizierbar sind. Des Weiteren wird der Index $\tilde{\Delta}(f)$ einer Relisation $\mathbf{x}_{\tilde{\Delta}(f)}$ aus Platzgründen nicht dargestellt. Für w wurde in diesem Fall der Faktorknoten f_1 gewählt.

Zuerst wird die Definition der lokalen Funktionen in die Formel für die Normalisierung eingesetzt (3.18). Anschließend wird die Summe über alle möglichen Labelkombinationen mit Hilfe des Distributivgesetzes entsprechend den Argumenten der lokalen Funktionen aufgeteilt.

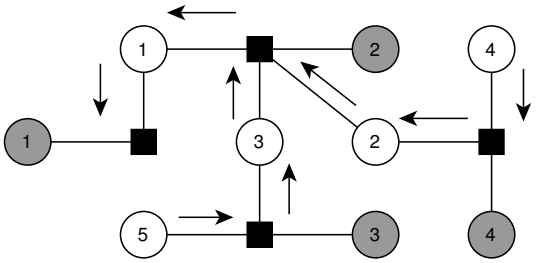


Abbildung 3.7: Ein Faktorgraph mit fünf versteckten Knoten. Dieser soll die Herleitung der Normalisierung aus den eingehenden Nachrichten veranschaulichen. In diesem Fall wird die Normalisierung an Knoten f_1 berechnet.

$$
\begin{aligned}
& Z(\mathbf{x}) && (3.17)\\
&= \sum_{\mathbf{y}} \prod_{f \in F} f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}\right) \\
&= \sum_{\mathbf{y}} f_1(y_1 \,|\, \mathbf{x}) \cdot f_{\{1,2,3\}}(y_1, y_2, y_3 \,|\, \mathbf{x}) \cdot f_{\{2,4\}}(y_2, y_4 \,|\, \mathbf{x}) \cdot f_{\{3,5\}}(y_3, y_5 \,|\, \mathbf{x}) && (3.18)\\
&= \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot \sum_{\mathbf{y}_{V-1}} \left(f_{\{1,2,3\}}(y_1, y_2, y_3 \,|\, \mathbf{x}) \cdot f_{\{2,4\}}(y_2, y_4 \,|\, \mathbf{x}) \cdot f_{\{3,5\}}(y_3, y_5 \,|\, \mathbf{x}) \right) \\
&= \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot \left(\sum_{y_2, y_3} f_{\{1,2,3\}}(y_1, y_2, y_3 \,|\, \mathbf{x}) \cdot \left(\sum_{y_4} f_{\{2,4\}}(y_2, y_4 \,|\, \mathbf{x}) \right) \cdot \left(\sum_{y_5} f_{\{3,5\}}(y_3, y_5 \,|\, \mathbf{x}) \right) \right) \\
&= \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot \left(\sum_{y_2, y_3} f_{\{1,2,3\}}(y_1, y_2, y_3 \,|\, \mathbf{x}) \cdot m_{f_{\{2,4\}} \to 2}(y_2 \,|\, \mathbf{x}) \cdot m_{f_{\{3,5\}} \to 3}(y_3 \,|\, \mathbf{x}) \right) && (3.19)\\
&= \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot \left(\sum_{y_2, y_3} f_{\{1,2,3\}}(y_1, y_2, y_3 \,|\, \mathbf{x}) \cdot m_{2 \to f_{\{1,2,3\}}}(y_2 \,|\, \mathbf{x}) \cdot m_{3 \to f_{\{1,2,3\}}}(y_3 \,|\, \mathbf{x}) \right) \\
&= \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot m_{f_{\{1,2,3\}} \to 1}(y_1 \,|\, \mathbf{x}) = \sum_{y_1} f_1(y_1 \,|\, \mathbf{x}) \cdot m_{1 \to f_1}(y_1 \,|\, \mathbf{x}) \\
&= \sum_{y_1} b_{f_1}(y_1 \,|\, \mathbf{x}) = Z_{f_1}(\mathbf{x}) && (3.20)
\end{aligned}
$$

Die übrigen Summen spiegeln die Definition der Variablen- und Faktornachrichten wieder (3.19). Durch wiederholte Substitution der entsprechenden Definitionen bleibt letztendlich die Summe über den Beliefs von Faktorknoten f_1 übrig (3.20). Äquivalent dazu hätten die Summanden auch so umgestellt werden können, dass sich die Normalisierung als Summe der Beliefs jedes anderen Knotens ergibt.

In [41, 67] erfolgt die Berechnung von $Z(\mathbf{x})$ am ersten oder letzten Blatt an dem diese möglich ist (vgl. Abb. 3.6). Im Folgenden wird der dort verwendete Inferenzalgorithmus erläutert.

Forward-Backward Algorithmus. Bei der Betrachtung des Nachrichtenflusses in Abbildung (3.6) ist es nur allzu naheliegend, dass die Belief-Propagation in diesem Spezialfall, also dem seriellen Scheduling auf einem Linear-Chain Graphen, auch *Forward-Backward Algorithmus* (FB Algorithmus) genannt wird.

Da der Forward-Backward-Algorithmus ursprünglich auf Abhängigkeits- und nicht auf Faktorgraphen formuliert wurde, werden die Faktor- und Variablennachrichten zu sog. *Forward-* (3.21)

und *Backward*-Nachrichten (3.22) zusammengefasst. Dies ist bedenkenlos möglich, da die Faktorknoten $f_{\{t-1,t\}}$ eine Nachricht des versteckten Knotens v_{t-1} lediglich mit ihrer lokalen Funktion multiplizieren und an den versteckten Knoten v_t weiterleiten (und umgekehrt). Alle übrigen Faktorknoten sind Blätter, wodurch sie stets dieselben Nachrichten verschicken. Damit gilt $\alpha_t(y \mid \mathbf{x}) = m_{v_t \to f_{\{t,t+1\}}}(y \mid \mathbf{x})$ sowie $\beta_t(y \mid \mathbf{x}) = m_{f_{\{t,t+1\}} \to v_t}(y \mid \mathbf{x})$. Die Folge der Knoten wird dabei oftmals als zeitliche Abfolge von Ereignissen interpretiert, daher werden die Knoten üblicherweise mit t indiziert. Die Normalisierung (3.23) wird in diesem Fall als Summe der Forward-Nachrichten des letzten Knotens berechnet [67]. Da CRFs allerdings ungerichtete Modelle sind, existiert tatsächlich keine echte Vorgängerrelation. Man würde also dieselbe Dichtefunktion erhalten, wenn die Berechnung der Nachrichten am "letzten" Knoten beginnen würde.

$$\alpha_t(y \mid \mathbf{x}) = \sum_{y' \in \mathcal{Y}} \exp\left(\phi_t(y, \mathbf{x}_t) + \phi_{\{t-1,t\}}\left(y', y, \mathbf{x}_t\right)\right) \cdot \alpha_{t-1}\left(y' \mid \mathbf{x}\right) \tag{3.21}$$

$$\beta_t(y \mid \mathbf{x}) = \sum_{y' \in \mathcal{Y}} \exp\left(\phi_{t+1}\left(y', \mathbf{x}_t\right) + \phi_{\{t,t+1\}}\left(y, y', \mathbf{x}_t\right)\right) \cdot \beta_{t+1}\left(y' \mid \mathbf{x}\right) \tag{3.22}$$

$$Z(\mathbf{x}) = \sum_{y \in \mathcal{Y}} \alpha_T(y \mid \mathbf{x}) \tag{3.23}$$

Die Randverteilungen (3.24, 3.25) ergeben sich analog zu denen der Faktorknoten der allgemeinen Belief-Propagation, als normalisierte Faktorbeliefs [57, 41].

$$p(v_t = y \mid \mathbf{x}) = \frac{\alpha_t(y \mid \mathbf{x})\, \beta_t(y \mid \mathbf{x})}{Z(\mathbf{x})} \tag{3.24}$$

$$p\left(v_{t-1} = y', v_t = y \mid \mathbf{x}\right) = \frac{\alpha_{t-1}(y' \mid \mathbf{x}) \exp\left(\phi_t(y, \mathbf{x}_t) + \phi_{\{t-1,t\}}(y', y, \mathbf{x}_t)\right) \beta_t(y \mid \mathbf{x})}{Z(\mathbf{x})} \tag{3.25}$$

Der Forward-Backward-Algorithmus wurde in dieser Form bereits von Lafferty et al. [41] zur Inferenz in Linear-Chain CRFs vorgeschlagen. Allerdings wurde dort eine Matrix-Notation zur Formulierung der Nachrichten verwendet. Die hier (sowie in [40, 79]) verwendete Mengen-Notation ist allgemeiner, da sie auch die Darstellung von Cliquen mit mehr als zwei Knoten erlaubt.

Der Pseudocode des FB Algorithmus ist in Alg. 1 dargestellt. Sei T die Anzahl der Knoten des größten Beispiels. Die benötigte Laufzeit zur Berechnung aller Randverteilungen sowie der Normalisierung $Z(\mathbf{x})$ beträgt $\mathcal{O}(T\,|\mathcal{Y}|^2)$, da für jeden Zeitpunkt $|\mathcal{Y}|$ Nachrichten berechnet werden müssen und die Berechnung einer Nachricht aus der Summation von $|\mathcal{Y}|$ Termen besteht. Dies ist ein erheblicher Gewinn im Vergleich zur exponentiellen Laufzeit $\mathcal{O}(|\mathcal{Y}|^T)$ der naiven Summation der Wahrscheinlichkeiten aller Belegungen. Damit liefert der Forward-Backward Algorithmus alle Größen, die zur Klassifikation sowie zum Training von Linear-Chain CRFs erforderlich sind.

Wie bereits zu Anfang dieses Kapitels erwähnt, wird in Kapitel 5 eine parallele Variante des Forward-Backward Algorithmus sowie der seriellen Belief-Propagation entwickelt, auch wenn die sequenzielle Struktur der beiden Algorithmen zum jetzigen Zeitpunkt noch kein großes Parallelisierungspotential vermuten lässt.

3.6.2 Inferenz in Graphen mit Kreisen

Im Folgenden wird die Belief-Propagation auf allgemeinen Graphen vorgestellt. Das Versenden der Nachrichten beginnt beim seriellen Scheduling an den Blättern. Enthält der zugrundeliegende Graph des Modells Kreise, so existiert möglicherweise kein einziges Blatt im Graphen, wie es beispielsweise im Gitter-Graph (Abb. 3.3) der Fall ist. Diese Beobachtung legt die Frage nahe, wie die

Algorithmus 1 Pseudocode des Forward-Backward Algorithmus. y_k bezeichnet das k-te Label aus $\mathcal{Y}$.

```
1: for t=1 to T
2:   for k=1 to |𝒴|
3:     berechne Forward-Nachricht α_t(y_k | x)
4: berechne Z(x)
6: for t=T to 1
7:   for k=1 to |𝒴|
8:     berechne Backward-Nachricht β_t(y_k | x)
9:     berechne Randverteilungen
```

Berechnung der Nachrichten in solchen Graphen initiiert werden soll. Dieses Problem wird umgangen, indem alle Knoten auf allen inzidenten Kanten initial eine 1 empfangen. Mit dieser Konvention verschickt jeder Knoten an jeden seiner Nachbarn in jeder Iteration eine neue Nachricht. Dieses Scheduling wird nach [39] auch *Flooding*-Schedule genannt, da der Graph in jeder Iteration mit neuen Nachrichten "geflutet" wird. Aufgrund der Tatsache, dass der Nachrichtenaustausch an allen Knoten parallel stattfindet wurde in [5, 40] angemerkt, dass die Belief-Propagation eine inhärent parallele Struktur besitzt. Die Belief-Propagation mit Flooding-Schedule auf Graphen mit Kreisen wird üblicherweise als Loopy-Belief-Propagation bezeichnet.

Das versenden neuer Nachrichten wird dabei solange wiederholt bis Konvergenz eintritt, das heißt bis alle Nachrichten der i-ten Iteration mit den Nachrichten der $(i-1)$-ten Iteration übereinstimmen. Die Berechnung der Anzahl benötigter LBP-Iterationen bis zur Konvergenz (falls diese möglich ist, s.u.) ist im allgemeinen NP-vollständig, da dazu die Baumweite des Graphen berechnet werden muss [2, 72]. Daher wird LBP oft nach einer vorher festgelegten Anzahl an Iterationen I abgebrochen. Wie auch bei der seriellen BP können anschließend die Beliefs und damit die Randverteilungen sowie die Normalisierung berechnet werden.

Die Laufzeit von LBP ist allerdings wesentlich höher als diejenige von BP oder FB. In jeder Iteration verschickt jeder Faktorknoten an alle seine Nachbarn $|\mathcal{Y}|$ Nachrichten. Dementsprechend müssen in jeder Iteration $\sum_{f\in F} |\Delta(f)| \cdot |\mathcal{Y}|$ Faktornachrichten berechnet werden. Analog dazu erzeugen die Variablenknoten pro Iteration $\sum_{v\in V} |\Delta(v)| \cdot |\mathcal{Y}|$ Nachrichten an die Faktorknoten. Daher beträgt die Gesamtlaufzeit $\mathcal{O}(I \cdot |F \cup V| \cdot \Delta_{max} \cdot |\mathcal{Y}|^{\Delta_{max}})$, wobei Δ_{max} die Größe der größten Nachbarschaft aller Knoten aus $|F \cup V|$ bezeichnet und I wahlweise einer vorher festgelegten Anzahl an Iteration oder der Anzahl benötigter Iterationen bis zur Konvergenz entspricht. Der Algorithmus ist also exponentiell in Δ_{max}. Da Δ_{max} in praktisch relevanten Graphen [61, 67, 72] viel kleiner ist als n, ist dies eine nicht zu vernachlässigende Verbesserung der asymptotischen Laufzeit im Vergleich zur naiven Summation über alle $|\mathcal{Y}|^n$ möglichen Realisationen.

Dieses Vorgehen liefert allerdings nur eine Approximation der Randverteilung und muss nicht notwendigerweise konvergieren. In [47] werden verschiedene Beispiele für Graphische Modelle vorgestellt, in denen die LBP nicht konvergiert oder die approximierte Dichtefunktion sehr ungenau ist. Bereits 1988 heißt es in einem Zitat von Pearl "*If we ignore the existence of loops and permit the nodes to continue communicating with each other as if the network were singly connected, messages may circulate indefinitely around these loops, and the process may not converge to a stable equilibrium.*" [53]. Trotzdem gibt es viele erfolgreiche Anwendungen von LBP auf nicht kreisfreien Graphen in den Bereichen der Bitfehlerkorrektur [22] sowie der Computer-Vision [21], wobei die dort verwendeten Gittergraphen (Abb. 3.3) ausschließlich aus Kreisen bestehen.

Der Pseudo-Code des LBP-Algorithmus mit Flooding-Schedule ist in Alg. 2 dargestellt. Die Beliefs und Normalisierungen werden genau wie die der seriellen BP aus den Nachrichten berechnet.

Algorithmus 2 Pseudocode der Loopy-Belief-Propagation. y_k bezeichnet das k-te Label, f_i den i-ten Faktorknoten, v_i den i-ten Variablenknoten, v_j den j-ten Nachbarn des Faktorknotens f_i sowie f_j den j-ten Nachbarknoten des Variablenknotens v_i.

```
 1: Solange nicht konvergiert:
 2:   for i=1 to |F|
 3:     for j=1 to |Δ(f_i)|
 4:       for k=1 to |𝒴|
 5:         berechne Faktornachricht m_{f_i→u_j}(y_k | x)
 6:   for i=1 to |V|
 7:     for j=1 to |Δ(v_i)|
 8:       for k=1 to |𝒴|
 9:         berechne Variablennachricht m_{u_i→f_j}(y_k | x)
10: berechne Beliefs
11: berechne Z(x)
14: berechne Randverteilungen
```

Da zur Berechnung der Faktornachrichten in allen hier gezeigten Varianten der Belief-Propagation jede lokale Funktion mit jeder bekannten Realisation mindestens einmal ausgewertet wird, können die Potentiale der Realisation $\mathbf{y}$ während der Berechnung der Faktornachrichten multipliziert und abschließend durch $Z(\mathbf{x})$ dividiert werden um die Wahrscheinlichkeit $p(\mathbf{y} \mid \mathbf{x})$ zu berechnen. Die asymptotische Laufzeit der jeweiligen Algorithmen bleibt von dieser Berechnung unberührt.

In diesem Abschnitt wurden die verwandten Algorithmen FB, BP und LBP zur Berechnung der Randverteilungen vorgestellt. Für jeden dieser drei wird in Kapitel 5 versucht, eine parallele Variante zu entwickeln.

3.7 Klassifikation mit CRFs

In diesem Abschnitt wird die Frage nach einer effizienten Berechnung der wahrscheinlichsten Realisation aller Knoten beantwortet. Soll eine ungelabelte Beobachtung $\mathbf{x}$ mit einem CRF klassifiziert werden, so kommen für das struturelle Label $\mathbf{y} \in \mathcal{Y}^n$ exponentiell viele Möglichkeiten in Betracht (3.26). Mit Hilfe der Belief-Propagation ist es jedoch möglich die wahrscheinlichsten Realisationen der Nachbarn eines Faktorknotens in wesentlich kürzerer Zeit zu berechnen (3.27).

$$\mathbf{y}^* = \arg\max_{\mathbf{y} \in \mathcal{Y}^n} p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}) \tag{3.26}$$

$$\mathbf{y}^*_{\Delta(f)} = \arg\max_{\mathbf{y}_{\Delta(f)} \in \mathcal{Y}^{|\Delta(f)|}} p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(f)} \mid \mathbf{x}\right) \tag{3.27}$$

Sofern der Graph des Modells kreisfrei ist kann das Distributivgesetzt der Maximumsbildung (3.28) angewendet werden.

$$a \cdot \max(b, c) = \max(ab, ac) \tag{3.28}$$

Das heißt, nachdem die wahrscheinlichste Realisation $\mathbf{y}^*_{\Delta(f)}$ der Nachbarn eines beliebigen[2] Faktorknotens $f \in F$ – dem *Startknoten* – mit Hilfe der Randverteilung berechnet wurde, stehen

[2] Hier kann z.B. der Faktorknoten mit der wahrscheinlichsten Realisation unter allen Realisationen aller Faktorknoten verwendet werden.

die Label der Knoten in $\Delta(f)$ fest. Infolgedessen muss an deren benachbarten Faktorknoten $\Delta(\Delta(f)) - f$ lediglich die wahrscheinlichste Realisation bestimmt werden, die mit den Labeln aus $\mathbf{y}^*_{\Delta(f)}$ übereinstimmt. Dies wird wiederholt, bis allen Knoten ein Label zugewiesen wurde. Eine so erfolgte Klassifikation liefert die wahrscheinlichste Realisation der gesamten Struktur und heißt hier *serielle* Klassifikation. Die Wahl eines Startknotens induziert allerdings eine "Richtung" des Modells. Laut Rabiner [57] ist es jedoch nicht zwingend notwendig, eine "Richtung" anzunehmen. Ist die wahrscheinlichste Realisation eines Knotens zu bestimmen, so kann auch diejenge mit dem maximalen Belief gewählt werden. Das heißt $\mathbf{y}^*_v = \arg\max_{y \in \mathcal{Y}} b_v(y \mid \mathbf{x})$. Allerdings stimmen die so erhaltenen Label nicht notwendigerweise mit den Labeln überein, die die oben beschriebene Vorgehensweise liefert [57]. Dafür erlaubt diese Methode die parallele Klassifikation aller Knoten des Graphen, was sie wiederum interessant für eine parallele Implementierung von CRFs macht. Gerade für nicht kreisfreie Graphen ist diese Vorgehensweise von Bedeutung, da die sinnvolle Wahl eines Startknotens in einem Graph ohne Blätter nicht immer möglich ist. Da dieses Vorgehen zwar in [57] erwähnt aber nicht benannt wurde, wird es hier als *Maximum-Belief Klassifikation* (MBK) bezeichnet.

Klassifikation in Linear-Chain CRFs. In einer reinen Klassifikationsanwendung ist man am Wert der Randverteilung im Grunde nicht interessiert. Wie in [1, 40] beschrieben, kann in diesem Fall der maximale Summand einer Faktornachricht – die sog. *Max-Faktornachricht* (3.29) – durch den Graphen propagiert werden. Berechnet man auf Basis dieser Max-Faktornachrichten die Beliefs und bestimmt mit diesen eine wahrscheinlichste Realisation, so stimmt diese mit derjenigen überein, die man erhält, wenn man die Klassifikation auf Basis der gewöhnlichen Faktornachrichten (3.9) bestimmt.

$$m^*_{f \to v}(y \mid \mathbf{x}) = \max_{\mathbf{y'}_{\Delta(f)-v}} f\left(v = y, \mathbf{y'}_{\Delta(f)-v} \mid \mathbf{x}_{\tilde{\Delta}(f)}\right) \prod_{u \in \Delta(f)-v} m_{u \to f}\left(\mathbf{y'}_u \mid \mathbf{x}\right) \tag{3.29}$$

Im Fall von Linear-Chain CRFs wird üblicherweise der sog. *Viterbi-Algorithmus* zur Bestimmung der wahrscheinlichsten Realisation verwendet. Er entspricht der oben beschriebenen seriellen Klassifikation, wobei die Max-Forward-Nachrichten (3.30) zur Berechnung der Maximalstelle verwendet werden. Dabei wird während der Berechnung der Nachrichten ebenfalls der wahrscheinlichste Vorgänger (3.31) eines Knotens bestimmt, so dass Berechnung und Maximumsbildung in derselben Iteration durchgeführt werden.

$$\alpha^*_t(y \mid \mathbf{x}) = \max_{y' \in \mathcal{Y}} \exp\left(\phi_t(y, \mathbf{x}_t) + \phi_{\{t-1,t\}}\left(y', y, \mathbf{x}_t\right)\right) \cdot \alpha_{t-1}\left(y' \mid \mathbf{x}\right) \tag{3.30}$$

$$\delta_t(y \mid \mathbf{x}) = \arg\max_{y' \in \mathcal{Y}} \exp\left(\phi_t(y, \mathbf{x}_t) + \phi_{\{t-1,t\}}\left(y', y, \mathbf{x}_t\right)\right) \cdot \alpha_{t-1}\left(y' \mid \mathbf{x}\right) \tag{3.31}$$

Nach Berechnung von δ^* wird das wahrscheinlichste Label $\mathbf{y}^*_T$ des letzten Knotens als Maximum der $\alpha^*_T(y \mid \mathbf{x})$ bestimmt. Die Label der anderen Knoten können daraufhin rekursiv aus $\delta_t(y \mid \mathbf{x})$ abgelesen werden. Das Label $\mathbf{y}^*_{T-1}$ wird also als wahrscheinlichster Vorgänger $\delta_t(\mathbf{y}^*_T \mid \mathbf{x})$ des Labels $\mathbf{y}^*_T$ gewählt, usw. Der Pseudocode des Viterbi-Algorithmus ist in Alg. 3 zu sehen.

Die beiden in diesem Abschnitt vorgestellten Klassifikationsalgorithmen – MBK sowie der Viterbi-Algorithmus – werden in Kapitel 5 für die GPGPU Implementierung von CRFs verwendet.

3.8 Training von CRFs

In diesem Abschnitt wird erklärt, wie die Parameter eines CRFs an eine Trainingsmenge angepasst werden. Dies entspricht der dritten Frage vom Anfang dieses Kapitels.

Algorithmus 3 Pseudocode des Viterbi-Algorithmus. y_k bezeichnet das k-te Label aus $\mathcal{Y}$.

```
1: for t=1 to T
2:   for k=1 to |𝒴|
3:     berechne Max-Forward-Nachricht α*_t(y_k | x)
4:     berechne wahrscheinlichsten Vorgänger δ_t(y | x)
4: y*_T = arg max_{y∈𝒴} α*_T(y | x)
5: for t=T-1 to 1
6:     y*_t = δ_t(y | x, y*_{t+1})
```

Um ein Conditional Random Field zu trainieren, das heißt die Verteilung der Label bei gegebenen Beobachtungen zu lernen, wird die aus Abschnitt 2.4 bekannte Maximum-Likelihood Methode angewendet. Die Gleichungen (3.32) und (3.33) entstehen, wenn die CRF-Dichtefunktion in die Likelihood- bzw. Log-Likelihood-Funktion eingesetzt wird. Diese Zielfunktionen sind nicht-linear und müssen daher mit Hilfe iterativer Optimierungsverfahren maximiert werden. Vorweg sei dazu angemerkt, dass die Berechnung der Likelihood für nicht-triviale CRFs sehr zeitaufwändig ist, da die Modelle teilweise weit mehr als 10^6 Parametern besitzen, von denen jeder einzelne an die Trainingsdaten angepasst werden muss. Daher sind effiziente Optimierungsverfahren für ein schnelles Training unerlässlich.

$$\mathcal{L}_{CRF}(\boldsymbol{\theta};\mathcal{T}) = \prod_{i=1}^{N} \frac{1}{Z\left(\mathbf{x}^{(i)}\right)} \prod_{f\in F} f\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) \tag{3.32}$$

$$l_{CRF}(\boldsymbol{\theta};\mathcal{T}) = \sum_{i=1}^{N} \left(\sum_{f\in F} \ln f\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) - \ln Z\left(\mathbf{x}^{(i)}\right) \right) \tag{3.33}$$

Lafferty et al. [41] verwendeten ursprünglich das von Della Pietra et al. [17] postulierte Improved-Iterative-Scaling (IIS) zur Adaption der Modellparameter. Da allerdings schon nach kurzer Zeit u.a. Sha und Pereira [62] wesentlich bessere Ergebnisse (im Sinne von F_1-Score) mit numerischen Optimierungsverfahren erzielten, wurde IIS von diesen verdrängt. Heute lässt sich keine CRF Implementierung finden, die IIS zur Optimierung der Likelihood verwendet. Aus diesem Grund wird dieses Verfahren hier nicht weiter betrachtet. Stattdessen wird die Funktionsweise numerischer Optimierungsverfahren erläutert und deren Verwendung motiviert, sowie die in Kapitel 5 verwendeten Verfahren vorgestellt.

Bei der Wahl eines Optimierungsverfahrens ist es hilfreich, einige analytische Eigenschaften der zu optimierenden Zielfunktion zu kennen. Glücklicherweise lässt sich die Verteilungsfunktion von CRFs in eine bekannte Familie von Verteilungen einordnen.

Definition 3.2. Eine parametrisierte Wahrscheinlichkeitsfunktion $p_{\boldsymbol{\theta}}$ gehört zu einer *Exponentialfamilie in kanonischer Form*, falls sie die funktionale Form von Gleichung (3.34) besitzt.

$$p_{\boldsymbol{\theta}}(\mathbf{x}) \quad = \quad h(\mathbf{x}) \exp\left(\langle \mathbf{f}(\mathbf{x}), \boldsymbol{\theta}\rangle - \ln Z\right) \tag{3.34}$$

Setzt man $h(\mathbf{x}) := 1$ lässt sich die Dichte von CRFs (3.35) für ein gegebenes $\mathbf{x}$ zu Gleichung (3.36) umformen. Dabei ist $\mathbf{f} = (\mathbf{f}_f)_{f\in F}$ eine Konkatenation der Merkmalsvektoren aller Faktorknoten des Graphen.

$$
\begin{aligned}
p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}) &= \frac{1}{Z(\mathbf{x})} \prod_{f \in F} f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right) && (3.35)\\
&= \frac{1}{Z(\mathbf{x})} \prod_{f \in F} \exp\left(\left\langle \mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_f \right\rangle\right) \\
&= Z(\mathbf{x})^{-1} \exp\left(\sum_{f \in F} \left\langle \mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_f \right\rangle\right) \\
&= \exp\left(\sum_{f \in F} \left\langle \mathbf{f}_f\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}\right), \boldsymbol{\theta}_f \right\rangle - \ln Z(\mathbf{x})\right) \\
&= \exp\left(\langle \mathbf{f}(\mathbf{y}, \mathbf{x}), \boldsymbol{\theta} \rangle - \ln Z(\mathbf{x})\right) && (3.36)
\end{aligned}
$$

Diese Darstellung der Dichte zeigt, dass CRFs zu einer Exponentialfamilie in kanonischer Form gehören. Laut Barndorff-Nielsen [3] ist die Normalisierung $Z(\mathbf{x})$ einer Exponentialfamilie eine unendlich oft differenzierbare, konvexe Funktion. Des Weiteren gilt:

- Ist l eine konvexe Funktion, so ist $-l$ konkav.
- Die Summation konvexer (konkaver) Funktionen erhält die Konvexität (Konkavität).

Wird das in den Potentialfunktion enthaltene Skalarprodukt zu einer Summe umgeformt (3.37) und die soeben aufgezählten Punkte berücksichtigt, so folgt, dass die Log-Likelihood-Funktion l_{CRF} konkav ist [67]. Hierbei bezeichnet $\mathbf{f}_{f,k}$ den k-ten Eintrag des Merkmalsvektors des Faktorknotens f und $\boldsymbol{\theta}_{f,k}$ den k-ten Eintrag seines entsprechenden Gewichtsvektors.

$$
l_{CRF}(\boldsymbol{\theta}; \mathcal{T}) = \sum_{i=1}^{N} \sum_{f \in F} \sum_{k=1}^{|\boldsymbol{\theta}_f|} \mathbf{f}_{f,k}\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) \cdot \boldsymbol{\theta}_{f,k} - \sum_{i=1}^{N} \ln Z\left(\mathbf{x}^{(i)}\right) \tag{3.37}
$$

Nun wird analog zu dem in Abschnitt über die Maximum-Likelihood Methode (Abs. 2.4) gezeigten Beispiel vorgegangen. Dort wurde lediglich der ML-Schätzer eines Parameters gesucht. Hier sollen Schätzer für alle Parameter der Log-Likelihood gefunden werden.

Der Gradient. Der Vektor der partiellen Ableitungen nach allen Parametern einer Funktion $l : \mathbb{R}^K \to \mathbb{R}$ heißt *Gradient* (3.38). Dieser zeigt, ausgehend vom K-dimensionalen Punkt $\boldsymbol{\theta}$, in die Richtung des steilsten Anstiegs der Funktion l.

$$
\nabla l(\boldsymbol{\theta}) := \begin{pmatrix} \frac{\partial f}{\partial \boldsymbol{\theta}_1} \\ \frac{\partial f}{\partial \boldsymbol{\theta}_2} \\ \cdots \\ \frac{\partial f}{\partial \boldsymbol{\theta}_K} \end{pmatrix} \tag{3.38}
$$

Auf dieser Tatsache basiert eine große Klasse numerischer Optimierungsverfahren, die sog. *Gradientenverfahren*. Vor der Behandlung solcher Verfahren soll zunächst der Gradient der CRF-Log-Likelihood Funktion betrachtet werden. Dazu wird $l_{CRF}(\boldsymbol{\theta}; \mathcal{T})$ partiell nach jedem Parameter $\boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)}, \mathbf{x}_v}$ abgeleitet. Dabei ist $\mathbf{x}_v$ eine Realisation des beobachteten Nachbarknotens $v \in \tilde{\Delta}(f)$ des Faktorknotens $f \in F$. $\mathbf{f}_{\mathbf{y}_{\Delta(f)}, \mathbf{x}_v}$ bezeichnet den Eintrag des Merkmalsvektors von f für die

Realisation $\mathbf{y}_{\Delta(f)}, \mathbf{x}_v$. Der Ausdruck $\mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right)$ entspricht einer Indikatorfunktion, die genau dann 1 ist, falls $\mathbf{y}_{\Delta(f)}$ und $\mathbf{y}^{(i)}_{\Delta(f)}$ sowie $\mathbf{x}_v$ und $\mathbf{x}^{(i)}_v$ identisch sind mit $v \in \tilde{\Delta}(f)$.

$$\frac{\partial l_{CRF}(\boldsymbol{\theta};\mathcal{T})}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}} = \sum_{i=1}^{N} \mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) - \sum_{i=1}^{N} \frac{\partial \ln Z\left(\mathbf{x}^{(i)}\right)}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}} \tag{3.39}$$

Entsprechend der Kettenregel der Differentialrechnung gilt $\frac{\partial \ln f(x)}{\partial x} = \frac{\partial f(x)}{\partial x} \cdot f(x)^{-1}$. Dies kann laut [67] auf die partielle Ableitung von $\ln Z(\mathbf{x})$ in der rechten Seite von (3.39) angewendet werden, was nach einigen algebraischen Umformungen zu (3.40) führt.

$$\frac{\partial l_{CRF}(\boldsymbol{\theta};\mathcal{T})}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}} = \sum_{i=1}^{N} \mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) - \sum_{i=1}^{N} \mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) \cdot p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(f)} \mid \mathbf{x}^{(i)}\right) \tag{3.40}$$

An dieser Gleichung kann man erkennen, warum eine schnelle Berechnung der Randverteilungen essenziell für eine jede praxistaugliche CRF-Implementierung ist. Auf der rechten Seite von Gleichung (3.40) wird die Wahrscheinlichkeit der Realisation $\mathbf{y}_{\Delta(f)}$ bei den gegebenen Beobachtungen des i-ten Beispiels – also die Randverteilung – benötigt, auch wenn die Realisation im i-ten Beispiel gar nicht vorkommt. Die Randverteilungen müssen also für alle jemals beobachteten Realisationen berechnet werden, und nicht nur für diejenigen die unter $\mathbf{x}_v$ bekannt sind. Beim Einsatz von Parameter-Tying ergibt sich eine Änderung in der formalen Darstellung der partiellen Ableitungen welche im Folgenden erläutert wird.

Gradient und Parameter-Tying. Wird das aus Abschnitt 3 bekannte Parameter-Tying angewendet, so wird derselbe Gewichtsvektor für mehrere Faktorknoten verwendet. Ist also $\mathcal{F}_h \in \mathcal{F}$ das h-te Faktortemplate und $f \in \mathcal{F}_h$ ein Faktor dieses Templates, so ist $\boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h},\mathbf{x}_v}$ das Gewicht der Realisation $\mathbf{y}_{\Delta(f)}, \mathbf{x}_v$. In diesem Fall ergibt sich die partielle Ableitung als Gleichung (3.41). Die Herleitung verläuft analog zu der von Gleichung (3.40), der einzige technische Unterschied ist, dass beim Differenzieren von 3.37 mehr als ein Term pro Beispiel übrig bleiben kann. Dies äußert sich in Gleichung (3.41) durch die zusätzliche Summation über alle Faktorknoten des Templates.

$$\frac{\partial l_{CRF}(\boldsymbol{\theta};\mathcal{T})}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h},\mathbf{x}_v}} = \sum_{f \in \mathcal{F}_h} \left(\sum_{i=1}^{N} \mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) - \sum_{i=1}^{N} \mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}_{\Delta(f)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) \cdot p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(f)} \mid \mathbf{x}^{(i)}\right) \right) \tag{3.41}$$

Da Gleichung (3.40) laut Abschnitt 3 lediglich ein Spezialfall von Gleichung (3.41) ist, wird im Folgenden stets Gleichung (3.41) als Berechnungsvorschrift für die Komponenten des Gradienten betrachtet.

Erwartung. Eine Substitution der Definition des Erwartungswertes $\mathbb{E}(x) = \sum_i x^{(i)} p\left(x^{(i)}\right)$ in (3.41) offenbart eine sehr anschauliche Interpretation (3.42) der partiellen Ableitung [67].

$$\frac{\partial l_{CRF}(\boldsymbol{\theta};\mathcal{T})}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h},\mathbf{x}_v}} = \sum_{f \in \mathcal{F}_h} \left(\widetilde{\mathbb{E}}\left[\mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v} \mid \mathcal{T}\right] - \widehat{\mathbb{E}}\left[\mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v} \mid \mathcal{T}\right] \right) \tag{3.42}$$

Diese entspricht der Differenz der empirischen Erwartung $\widetilde{\mathbb{E}}$ und der Erwartung des Modells $\widehat{\mathbb{E}}$, bezüglich der Häufigkeit der durch den Parameter gewichteten Realisation in den Trainingsdaten. Dabei kann die empirische Erwartung einfach durch Abzählen der Häufigkeit der jeweiligen

Realisation in den Trainingsdaten bestimmt (3.41, links) und die Erwartung des Modells durch Summation der Wahrscheinlichkeiten, die das Modell der entsprechenden Realisation zugewiesen hat, berechnet werden (3.41, rechts). Die partielle Ableitung nach einem Parameter $\boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h},\mathbf{x}_v}$ misst also, ob dieser für die gegebene Trainingsmenge tendenziell zu groß oder zu klein gewählt ist. Im globalen Optimum, also bei maximaler Likelihood, ist der Gradient $\nabla l_{CRF}(\boldsymbol{\theta};\mathcal{T})$ identisch zum $\mathbf{0}$-Vektor. Also sind im Optimum alle partiellen Ableitungen 0 und die Erwartung des Modells stimmt mit der empirischen Erwartung exakt überein.

Da die zu maximierende Zielfunktion $l_{CRF}(\boldsymbol{\theta};\mathcal{T})$ nicht-linear ist, können die partiellen Ableitungen nicht einfach wie im Beispiel aus Abschnitt 2.4 mit 0 gleichgesetzt und aufgelöst werden, um die optimale Lösung zu finden. Trotzdem können die Ableitungsinformationen genutzt werden, um sich dem Optimum mit Hilfe iterativer Optimierungsverfahren anzunähren.

Algorithmus 4 Pseudeocode des Gradientenabstiegs

1: t := 0
2: f_t := $f\left(\boldsymbol{\theta}^{(t)}\right)$
3: while((($f^{(t-1)}$ - $f^{(t)}$)$> \epsilon$) or (t == 0))
4: t := $t+1$
5: $d^{(t)}$:= $\nabla f\left(\boldsymbol{\theta}^{(t)}\right)$
6: $\eta^{(t)}$:= $\min_\eta \nabla f\left(\boldsymbol{\theta}^{(t)} - \eta d^{(t)}\right)$
7: $\boldsymbol{\theta}^{(t+1)}$:= $\boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla f\left(\boldsymbol{\theta}^{(t)}\right)$
8: f_t := $f\left(\boldsymbol{\theta}^{(t)}\right)$
9: $\boldsymbol{\theta}^*$:= $\boldsymbol{\theta}^{(t)}$

3.8.1 Gradienten basierte Verfahren

Der *Gradientenabstieg* (*Verfahren des steilsten Abstiegs*, engl. *Gradient Descent*, *GD*) zählt zu den iterativen numerischen Optimierungsverfahren zur Minimierung differenzierbarer Zielfunktionen. Da die Log-Likelihood eigentlich maximiert werden soll, wird stattdessen $f := -l_{CRF}$ minimiert, was offensichtlich zur gleichen Optimalstelle $\boldsymbol{\theta}^*$ führt. Der Gradient $\nabla f(\boldsymbol{\theta})$ an der Stelle $\boldsymbol{\theta}$ zeigt in die Richtung des steilsten Anstiegs (und damit $-\nabla f(\boldsymbol{\theta})$ in die des steilsten Abstiegs) der zu optimierenden Funktion. Also muss festgestellt werden, wie weit die beste erreichbare Stelle entfernt ist. Dazu wird eine *Schrittweite* (auch: *Lernrate*) $\eta \in \mathbb{R}$ eingeführt. Gleichung (3.43) zeigt wie die Iterierte $\boldsymbol{\theta}^{(t)}$ in jeder Iteration des Gradientenabstiegs aktualisiert wird. Der Pseudocode ist in Alg. 4 gelistet. Dort bezeichnet ϵ eine kleine, vorher festgelegte Konstante die einer Mindestverbesserung entspricht. Wird diese nicht mehr erreicht, so terminiert das Verfahren. Obwohl der Gradientenaufstieg in dieser Form nicht implementiert wurde, wird er hier gezeigt, um die grundsätzliche Vorgehensweise iterativer Optimierungsverfahren zu verdeutlichen.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla f\left(\boldsymbol{\theta}^{(t)}\right) \tag{3.43}$$

$$\eta^* = \min_{\eta \in \mathbb{R}} f\left(\boldsymbol{\theta} - \eta \nabla f(\boldsymbol{\theta})\right) \tag{3.44}$$

Durch Lösung des eindimensionalen Optimierungsproblems (3.44) kann die optimale Schrittweite η^* ermittelt werden. Diese wird anschließend als Schrittweite $\eta^{(t)}$ der aktuellen Iteration übernommen und anschließend der nächste Suchpunkt $\boldsymbol{\theta}^{(t+1)}$ ermittelt. Unter gewissen Anforderungen an

die Schrittweite $\eta^{(t)}$ konvergiert das Verfahren sicher zum nächsten lokalen Optimum [76]. Allerdings ist die Konvergenzgeschwindigkeit stark von der konkreten Zielfunktion abhängig. Da die Log-Likelihood nicht strikt konvex ist, existieren möglicherweise mehrere gleichwertige globale Optima. Wird anstatt einer ML- eine MAP-Schätzung durchgeführt (Abs. 2.4), so ist die Zielfunktion strikt konvex [67]. Da es aber keine wesentlichen Unterschiede in der Herleitung der Zielfunktion sowie den partiellen Ableitungen gibt, wird die MAP-Schätzung hier nicht explizit berücksichtigt.

Die Bestimmung der Schrittweite bringt mehrere Probleme mit sich. Soll tatsächlich die im Sinne von Gleichung (3.44) optimale Schrittweite η^* ermittelt werden, sind dazu sehr viele Zielfunktionsauswertungen notwendig, was in praktischen Anwendungen zu unbrauchbar hohen Laufzeiten führt. Als Ausweg wird die Schrittweite oft approximativ mit Hilfe einer *Line-Search* bestimmt. Dazu wird die GD Iteration (3.43) für eine Menge $\mathcal{K}$ von Schrittweitenkandidaten "simuliert" und die Schrittweite, die die größte Verbesserung des Zielfunktionswertes brachte, ausgewählt. Enthält $\mathcal{K}$ allerdings viele Kandidaten, so werden erneut viele Zielfunktionsauswertungen benötigt. Sind dagegen nur wenige mögliche Schrittweiten in $\mathcal{K}$ enthalten, besteht die Gefahr, eine schlechte Schrittweite zu wählen. Ist die gewählte Schrittweite zu groß, zirkuliert das Verfahren um das Optimum und kann in ungünstigen Fällen divergieren. Ist sie zu klein gewählt, können sehr viele Iterationen notwendig sein, um das lokale Optimum zu erreichen.

Der "normale" Gradientenabstieg ist daher zum Training von CRFs eher ungeeignet. Sutton und McCallum [67] berichten von sehr langsamer Konvergenz zum Optimum und der damit verbundenen langen Laufzeit. Bereits Sha und Pereira [62] verwendeten daher das (Quasi-)Newton-Verfahren L-BFGS zur Maximierung der Likelihood. Verfahren dieser Klasse basieren auf der Annahme, dass sich die Zielfunktion in der Nähe des Optimums wie ein Polynom zweiten Grades verhält. In diesem Fall lässt sich zeigen, dass das in Alg. 4 dargestellte Verfahren schneller[3] konvergiert, wenn anstatt dem Gradienten die Suchrichtung $d^{(t)} := \left(\nabla^2 f\left(\boldsymbol{\theta}\right)\right)^{-1} \nabla f\left(\boldsymbol{\theta}\right)$ verwendet wird. Hierbei bezeichnet $\nabla^2 f\left(\boldsymbol{\theta}\right)$ die Matrix aller zweiten Ableitungen, auch *Hesse-Matrix* genannt. Da CRFs mehrere Millionen von Parametern besitzen können, ist die Bestimmung der exakten Hesse-Matrix kaum möglich. Die Hesse-Matrix eines Modells mit 10^6 Parametern würde bei der Verwendung von 32 Bit Fließkommazahlen bereits $3,8$ GiB Speicher benötigen. Bei L-BFGS wird diese daher mit Hilfe der m letzten Gradienten approximiert [49], was wiederum mit einer geringeren[4] Konvergenzordnung einhergeht.

Obwohl auch heute noch in einigen CRF-Implementierungen der L-BFGS Algorithmus zur Optimierung eingesetzt wird (s. Abs. 5.1), ist dieser in Bezug auf die Laufzeit im Vergleich mit den im Folgenden vorgestellten Verfahren nicht konkurrenzfähig.

3.8.2 Stochastische Optimierung

Ein Optimierungsproblem heißt *stochastisch*, falls die Zielfunktion f einem stochastischen Prozess unterliegt und damit bei wiederholter Auswertung an der Stelle $\boldsymbol{\theta}$ unterschiedliche Zielfunktionswerte liefert. Die Optimalstelle $\boldsymbol{\theta}^*$ eines solchen Problems "bewegt" sich innerhalb einer unbekannten Teilmenge des Suchraums. Durch die Optimierung solcher Probleme kann also lediglich die erwartete Optimalstelle $\mathbb{E}\left(\boldsymbol{\theta}^*\right)$ berechnet werden.

Die Idee besteht nun darin, die Trainigsmenge eines CRFs als stochastischen Prozess aufzufassen, der in jeder Iteration $b \leq N = |\mathcal{T}|$ Beispiele erzeugt. Grundsätzlich ist b frei wählbar, beispielsweise wird von Bottou [7] der Fall $b = 1$ betrachtet, wohingegen in Vishwanathan et al. [72] Resultate für $b \in \{1, 3, 6, 8\}$ zu finden sind. Diese Beispiele werden zu einem sog. *Batch* $\mathcal{B} \subseteq \mathcal{T}$ zusammengefasst. Die Trainingsmenge wird also in $J = \left\lceil \frac{N}{b} \right\rceil$ Batches $\mathcal{B}^{(j)}$, $1 \leq j \leq J$ partitioniert. Auf jedem dieser

[3] Das Newton-Verfahren besitzt auf streng konvexen Funktionen eine quadratische Konvergenzordnung.
[4] L-BFGS besitzt auf streng konvexen Funktionen eine superlineare Konvergenzordnung.

Batches wird dann nacheinander eine Iteration des gewöhnlichen Gradientenabstiegs durchgeführt. Dies induziert ein stochastisches Optimierungsproblem, da die Optimalstelle für jeden Batch eine andere ist. Dieses Verfahren wird als *Stochastischer Gradientenabstieg* (engl: *Stochastic Gradient Descent*, *SGD*) [7, 8, 72] bezeichnet.

Augenscheinlich hat man das zu lösende Problem "noch schwerer" gemacht, da das Optimum nun nicht mehr exakt bestimmt werden kann. Bemerkt man jedoch, dass der Parametervektor $\boldsymbol{\theta}$ eines CRFs allein durch die Traingsmenge induziert wird, also eigentlich $\boldsymbol{\theta} := \boldsymbol{\theta}_{\mathcal{T}}$ gilt, wird klar, dass auch jeder Batch seinen eigenen Parametersatz $\boldsymbol{\theta}_{\mathcal{B}}$ induziert, welcher nur die Gewichte für die in $\mathcal{B}$ enthaltenen Beobachtungen beinhaltet. In Abhängigkeit von b resultiert dieses Vorgehen in einer dramatischen Komplexitätsreduktion des Optimierungsproblems, da $\boldsymbol{\theta}_{\mathcal{B}}$ nur einen sehr kleinen Anteil der Parameter $\boldsymbol{\theta}_{\mathcal{T}}$ enthält falls $b \ll N$. Wenn also die Realisation x im aktuellen Batch nicht vorkommt, so ist die Erwartung des Modells bezüglich aller Realisationen die x beinhalten 0 und stimmt somit mit der empirischen Erwartung überein. Daher werden die Parameter $\boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},x}$ für alle Realisationen $\mathbf{y}_{\Delta(f)}$ schlichtweg nicht benötigt.

Mit dieser Modifikation können nicht nur die Zielfunktion und ihr Gradient wesentlich schneller berechnet werden, da in jeder Iteration über weniger Beispiele iteriert werden muss, auch das Optimum ist aufgrund der viel geringeren Dimension von $\boldsymbol{\theta}_{\mathcal{B}}$ leichter aufzufinden. Gleichungen (3.45) und (3.46) zeigen die Zielfunktion und die Berechnungsvorschrift der partiellen Ableitungen für die stochastische Optimierung von CRFs. Hier heißen die Faktorknoten einmalig g, um eine Verwechslung mit dem Bezeichner f generischer Funktionen zu vermeiden.

$$l_{CRF}\left(\boldsymbol{\theta}_{\mathcal{B}^{(j)}}; \mathcal{B}^{(j)}\right) = \sum_{i=1}^{|\mathcal{B}^{(j)}|} \sum_{g \in \mathcal{F}_h} \sum_{k=1}^{|\boldsymbol{\theta}_{\mathcal{B}^{(j)}}|} \mathbf{f}_k\left(\mathbf{y}^{(i)}_{\Delta(g)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(g)}\right) \cdot \boldsymbol{\theta}_k - \sum_{i=1}^{|\mathcal{B}^{(j)}|} \ln Z\left(\mathbf{x}^{(i)}\right) \tag{3.45}$$

$$\frac{\partial l_{CRF}\left(\boldsymbol{\theta}_{\mathcal{B}^{(j)}}; \mathcal{B}^{(j)}\right)}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h}, \mathbf{x}_v}} = \sum_{g \in \mathcal{F}_h} \left(\sum_{i=1}^{|\mathcal{B}^{(j)}|} \mathbf{f}_{\mathbf{y}_{\Delta(g)}, \mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(g)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(g)}\right) - \sum_{i=1}^{|\mathcal{B}^{(j)}|} \mathbf{f}_{\mathbf{y}_{\Delta(g)}, \mathbf{x}_v}\left(\mathbf{y}_{\Delta(g)}, \mathbf{x}^{(i)}_{\tilde{\Delta}(g)}\right) \cdot p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(g)} \,|\, \mathbf{x}^{(i)}\right) \right) \tag{3.46}$$

Auch die Wahl der Schrittweite wird einfacher. Nach Murata [46] konvergiert SGD für die in jeder Iteration fallende Schrittweite $\eta^{(t)} = t^{-1}$ mit optimaler Konvergenzgeschwindigkeit. Ein stochastischer Gradientenabstieg, dessen Schrittweite proportional zur Anzahl absolvierter Iterationen fällt, wird auch *Annealed Stochastic Gradient* (*ASGD*) genannt. Der erwartete Vorhersagefehler fällt in diesem Fall in jeder Iteration um t^{-1} und konvergiert fast sicher zum erwarteten Maximum-Likelihood Schätzer. Die fallende Schrittweite ist zwar für den Konvergenzbeweis essenziell, in praktischen Anwendungen von SGD werden aber auch für kleine konstante Schrittweiten (z.B. $\eta \approx 10^{-2}$) gute Ergebnisse erzielt [72]. Dabei sind die Laufzeiten wesentlich geringerer und die erzielten F_1-Scores sind mit denen von L-BFGS vergleichbar. Wie auch die Evaluation in Kapitel 6 zeigen wird, befinden sich die mit SGD geschätzten Parameter schon nach einer einzigen Iteration sehr nahe am globalen Optimum.

Eine Beobachtung, die in diesem Zusammenhang von Bottou und LeCun [8] sowie von Vishwanathan et al. [72] gemacht wurde, ist, dass sich SGD zum Training auf beliebig großen Datenmengen und daher auch zur Online-Optimierung von CRFs eignet. Verfahren wie L-BFGS oder Conjugate-Gradient sind nicht für die stochastische Optimierung der Likelihood geeignet [60], da sich der Parametervektor $\boldsymbol{\theta}_{\mathcal{B}}$ in jeder Iteration ändert, wodurch eine sinnvolle, auf den letzten m Gradienten basierende Schätzung der Hesse-Matrix erschwert wird. Stochastische Optimierungsverfahren, die die inverse Hesse-Matrix zur Korrektur der Suchrichtung verwenden, werden auch als *2SGD-Verfahren* bezeichnet [7]. Bordes [6] zeigte allerdings, dass sich die Konvergenz-

raten von 2SGD-Verfahren nur um eine Konstante im Vergleich zum gewöhnlichen stochastischen Gradientenabstieg unterscheiden.

Der von Vishwanathan und Schraudolph vorgestellte *Stochastische-Meta-Abstieg* (engl. *Stochastic-Meta-Descent*, SMD) [73] erlaubt die Berücksichtigung von Ableitungsinformationen zweiter Ordnung auf eine andere Art als bei 2SGD-Verfahren. Basierend auf dem Delta-Delta-Algorithmus [33] und dem Exponentiated-Gradient-Verfahren [37] postulierten sie die in (3.47)-(3.49) dargestellten update Regeln für den Parametervektor. Hierbei ist $\boldsymbol{\eta} \in \mathbb{R}^{|\boldsymbol{\theta}|}$ erstmalig ein Vektor, der für jeden Modellparameter eine eigene Schrittweite beinhaltet und $\mathbf{v}$ die sog. *Gradienten-Historie* (engl. *Gradient-Trace*) (3.49). Der Operator "$\cdot$" bezeichnet die komponentenweise Vektor-Multiplikation (Hadamard-Produkt).

$$\boldsymbol{\eta}^{(t)} = \boldsymbol{\eta}^{(t-1)} \cdot \max\left(\frac{1}{2}, 1 - \mu \nabla f\left(\boldsymbol{\theta}^{(t)}\right) \cdot \mathbf{v}^{(t)}\right) \tag{3.47}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \boldsymbol{\eta}^{(t)} \cdot \nabla f\left(\boldsymbol{\theta}^{(t)}\right) \tag{3.48}$$

$$\mathbf{v}^{(t+1)} = \lambda \mathbf{v}^{(t)} - \boldsymbol{\eta}^{(t)} \cdot \left(\nabla f\left(\boldsymbol{\theta}^{(t)}\right) \cdot \lambda \nabla^2 f\left(\boldsymbol{\theta}^{(t)}\right) \mathbf{v}^{(t)}\right) \approx \sum_{i=0}^{t} \lambda^i \frac{\partial \boldsymbol{\theta}^{(t+1)}}{\partial \ln \boldsymbol{\eta}^{(t-i)}} \tag{3.49}$$

Der skalare Parameter $\mu \in \mathbb{R}$ heißt *Meta-Schrittweite* und kann als Schrittweite der Schrittweitensteuerung interpretiert werden. Der ebenfalls skalare Parameter $\lambda \in [0; 1]$ gibt an, welcher Anteil der letzten Parameterupdates genutzt werden soll, um den Trace und damit die Schrittweiten zu korrigieren (3.49). In [72] werden die Standardparameter $\lambda = 1$ sowie $\boldsymbol{\eta}_k^{(0)} = \mu = 10^{-1}$, $\forall 1 \leq k \leq |\boldsymbol{\theta}|$ vorgeschlagen, wobei in den dortigen Experimenten auch kleinere Werte verwendet werden. Im Zuge der Versuche (Kapitel 6) führten diese Parameter allerdings häufig zur Divergenz des Verfahrens. Es erwiesen sich $\lambda = \frac{3}{4}$ sowie $\mu = 10^{-2}$ als stabilste Variante. Eine Herleitung der Updateregeln sowie Ergebnisse über die Anwendung von SMD auf anderen Zielfunktionen sind in [73] zu finden.

In den CRF-Experimenten aus [72] konnte sich SMD zwar von SGD absetzen, allerdings räumte Vishwanathan in einer privaten E-Mail Kommunikation Konfigurationsfehler in den SMD-Experimenten ein, die angeblich zu überproportional guten F_1-Scores führten. In Kapitel 5 werden beide Verfahren implementiert, da SMD auf SGD basiert und sich des Weiteren eine, aus algorithmischer Sicht, hochinteressante und sehr gut zu parallelisierende Technik zu Nutze macht, um das Hesse-Matrix-Vektor-Produkt $\nabla^2 f\left(\boldsymbol{\theta}^{(t)}\right) \mathbf{v}^{(t)}$ zu berechnen: Die *Automatische Differentiation.*

Pearlmutter's Operator & Automatische Differentiation. Automatische Differentiation (*AutoDiff*) bezeichnet die Automatische Berechnung von Ableitungsinformationen, ohne diese explizit zu implementieren. Die für SMD eingesetzte AutoDiff Technik basiert laut [72] auf Pearlmutter's Differentialoperator. Pearlmutter postulierte in [54] seinen Differentialoperator $\mathcal{R}_{\mathbf{v}}$ ursprünglich zur Verwendung von Ableitungsinformationen zweiter Ordnung beim Training Neuronaler Netze. In [72, 73] wurde diese Technik aufgegriffen um das für SMD benötigte Hesse-Matrix-Vektor-Produkt in Linearzeit zu berechnen. Auf die Theorie hinter Differentialoperatoren wird an dieser Stelle nicht weiter eingegangen, stattdessen wird nur das Resultat präsentiert.

Seien f, g zwei reellwertige, differenzierbare Funktionen, $c \in \mathbb{R}$ eine Konstante, $t \in \mathbb{R}$ eine Variable sowie $\mathbf{v}, \boldsymbol{\theta} \in \mathbb{R}^n$ zwei Vektoren gleicher Länge. Für die Anwendung des Differentialoperators $\mathcal{R}_{\mathbf{v}}$ gelten die Äquivalenzen (3.50) bis (3.54). Diese entsprechen, mit Ausnahme von (3.54), der gewöhnlichen Differentiation.

$$\mathcal{R}_{\mathbf{v}}\left[cf\left(\boldsymbol{\theta}\right)\right] = c\mathcal{R}_{\mathbf{v}}\left[f\left(\boldsymbol{\theta}\right)\right] \tag{3.50}$$

$$\mathcal{R}_{\mathbf{v}}\left[f\left(\boldsymbol{\theta}\right) + g\left(\boldsymbol{\theta}\right)\right] = \mathcal{R}_{\mathbf{v}}\left[f\left(\boldsymbol{\theta}\right)\right] + \mathcal{R}_{\mathbf{v}}\left[g\left(\boldsymbol{\theta}\right)\right] \tag{3.51}$$

$$\mathcal{R}_{\mathbf{v}}\left[f\left(\boldsymbol{\theta}\right) g\left(\boldsymbol{\theta}\right)\right] = \mathcal{R}_{\mathbf{v}}\left[f\left(\boldsymbol{\theta}\right)\right] g\left(\boldsymbol{\theta}\right) + \mathcal{R}_{\mathbf{v}}\left[g\left(\boldsymbol{\theta}\right)\right] f\left(\boldsymbol{\theta}\right) \tag{3.52}$$

$$\mathcal{R}_{\mathbf{v}}\left[f\left(g\left(\boldsymbol{\theta}\right)\right)\right] = \nabla f\left(g\left(\boldsymbol{\theta}\right)\right) \mathcal{R}_{\mathbf{v}}\left[g\left(\boldsymbol{\theta}\right)\right] \tag{3.53}$$

$$\mathcal{R}_{\mathbf{v}}\left[\boldsymbol{\theta}\right] = \mathbf{v} \tag{3.54}$$

Die Anwendung von (3.53) auf den Gradienten einer Funktion f liefert $\mathcal{R}_{\mathbf{v}}\left[\nabla f\left(\boldsymbol{\theta}\right)\right] = \nabla^2 f\left(\boldsymbol{\theta}\right)\mathbf{v}$, wobei g als Identität gewählt wurde. Dies ist grundsätzlich nicht überraschend, da $\mathcal{R}_{\mathbf{v}}$ gerade so definiert ist, dass diese Gleichheit gilt. Allerdings besteht die Berechnung des Gradienten einer Funktion aus der Berechnung einer Folge verschiedener Teilformeln, welche wiederum aus Folgen arithmetischer Operationen bestehen. Der Gradient der CRF Dichtefunktion wird bekanntlich als Differenz der empirischen sowie der modellierten Erwartung bestimmt. Die Erwartung des Modells wird aus den Randverteilungen berechnet und diese wiederum aus den Potentialfunktionen. Die Potentialfunktionen entsprechen einfachen Summen der Gewichte. Wird nun von Beginn der Berechnung an, auf jede dieser Operationen der $\mathcal{R}_{\mathbf{v}}$ Operator angewandt, so erhält man einen Algorithmus zur Berechnung von $\mathcal{R}_{\mathbf{v}}\left[\nabla f\left(\boldsymbol{\theta}\right)\right]$ und damit von $\nabla^2 f\left(\boldsymbol{\theta}\right)\mathbf{v}$ – und zwar ohne die Hesse-Matrix explizit zu berechnen. Es stellt sich jedoch heraus, dass die Formulierung eines eigenen Algorithmus zur Berechnung von $\mathcal{R}_{\mathbf{v}}\left[\nabla f\left(\boldsymbol{\theta}\right)\right]$ bei Verwendung einer geeigneten Programmiersprache gar nicht notwendig ist. Einige Programmiersprachen (z.B. C++) erlauben die Überladung arithmetischer Operatoren. Wird nun ein Datentyp verwendet, der analog zu den komplexen Zahlen eine zweite Komponente besitzt, können die arithmetischen Operatoren dieses Datentyps so überladen werden, dass diese zusätzlich zu ihrem gewöhnlichen Resultat in der zweiten Komponente das Resultat der Anwendung von $\mathcal{R}_{\mathbf{v}}$ berechnen. Dazu müssen die Regeln (3.50) bis (3.54) implementiert werden. Im folgenden wird kurz die Funktionsweise eines solchen Datentyps erläutert.

Der Name des o.g. Datentyps sei `FAD`. Jede Variable vom Typ `FAD` enthält die beiden skalaren Komponenten a und b, wobei a stets den Wert enthält, den die Variable bei Verwendung eines gewöhnlichen Fließkommadatentyps (`float`, `double`) enthalten würde. Sei $\mathbf{x} \in \mathbb{R}^n \times \mathbb{R}^n$ vom Typ `FAD` die Variable nach der differenziert wird. Die b-Komponente vom jeweils i-ten Eintrag des Vektors $\mathbf{x}$ wird mit $\mathbf{v}_i$ initialisiert. Konstanten bzw. Variablen nach denen nicht differenziert wird, seien ebenfalls vom Typ `FAD` und enthalten in der b-Komponente den Wert 0. Enthält ein Algorithmus nun das Statement `y=x[0]*x[1]` ($y = x_0 x_1$), so wird tatsächlich der Code `y.a=x[0].a*x[1].a` sowie `y.b=(x[0].b*x[1].a)+(x[0].a*x[1].b)` ausgeführt, was der Anwendung von Regel (3.52) entspricht. Somit enthält `y.b` den Wert $\mathcal{R}_{\mathbf{v}}\left[\mathbf{x}_1\mathbf{x}_2\right]$.

Berechnet ein Algorithmus also eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$, so transformiert die Verwendung des Datentyps `FAD` diesen in einen Algorithmus für die Funktion $F : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$, die in der ersten Komponente des Ergebnisses mit $f\left(\mathbf{x}\right)$ übereinstimmt und in der zweiten den Wert $\mathcal{R}_{\mathbf{v}}\left[f\left(\mathbf{x}\right)\right]$ enthält. Ist ein Algorithmus zur Berechnung des Gradienten ∇f einer Funktion f bereits implementiert, so muss lediglich der Datentyp ausgetauscht werden um einen Algorithmus zu erhalten der ebenfalls $\mathcal{R}_{\mathbf{v}}\left[\nabla f\left(\boldsymbol{\theta}\right)\right] = \nabla^2 f\left(\boldsymbol{\theta}\right)\mathbf{v}$ berechnet. Dies entspricht genau dem für SMD benötigten Hesse-Matrix-Vektor-Produkt. Platz- und Laufzeitkomplexität verdoppeln sich dabei, wodurch der Ressourcenverbrauch asymptotisch gleich bleibt. Bei einer expliziten Berechnung der Hesse-Matrix würden Platz- und Laufzeitkomplexität dagegen quadratisch ansteigen. Für SMD bedeutet dies, dass zusätzlich zu den o.g. Updateregeln bei der Berechnung des Gradienten der Datentyp `FAD` verwendet werden muss. Da die stochastische Optimierung von CRFs dem Stand der Technik entspricht, werden SGD und SMD im Zuge der Parallelisierung in Kapitel 5 für das Training von CRFs verwendet. Zur Einordnung der Qualität der berechneten Parametervektoren wird in der Evaluation (Kapitel 6) eine Implementierung einbezogen, die L-BFGS verwendet.

4 Parallele Berechnung mit GPGPU

An dieser Stelle wird die parallele Berechnung mit Hilfe von *Grafikprozessoren* (*Graphics Processing Unit*, GPU) erläutert. Die Nutzung solcher GPUs zur Ausführung gewöhnlicher, nicht grafikbezogener Algorithmen wird als *General-Purpose computation on Graphics Processing Units* (GPGPU) bezeichnet. Aufgrund der inheränten Parallelität vieler Grafikanwendungen verfügen GPUs über eine höhere Anzahl an Recheneneinheiten (*Kerne*) im Vergleich zu aktuellen Prozessoren (*Central Processing Unit*, *CPU*). Heutige[5] CPUs besitzen zwischen 2 und 8 Kernen, die hier verwendeten GPUs besitzen mehr als 2^8. Daher werden GPUs auch als *hochparallele* Prozessoren bezeichnet. Als Hard- und Softwareplattform wurde NVidias CUDA ausgewählt. Ein Vergleich von CUDA mit anderen parallelen Architekturen (Abs. 4.2.7) wird zeigen, dass diese Wahl keine wesentliche Einschränkung der Allgemeingültigkeit der hier entwickelten Implementierung nach sich zieht. CUDA wird in Abschnitt 4.2 vorgestellt.

Alle Recheneinheiten einer GPU teilen sich einen gemeinsamen Hauptspeicher. In Abgrenzung dazu bestehen sog. *parallele Verteilte Systeme* (*Rechencluster*) aus einer Vielzahl vernetzter Rechner, von denen jeder seinen eigenen Hauptprozessor und Hauptspeicher besitzt. Obgleich einige der hier getätigten Aussagen für beide Architekturen gelten mögen, werden Rechencluster nicht explizit behandelt, da die unterschiedlichen Speichermodelle beim Algorithmenentwurf zu vollkommen anderen Designentscheidungen führen.

In Abs. 4.1 wird die PRAM als theoretisches Modell paralleler Berechnung vorgestellt und die Verwendung des Work-Time-Paradigmas als PRAM äquivalentes, einfacheres Modell motiviert. Im zweiten Abschnitt 4.2 wird CUDA als konkrete parallele Berechnungsarchitektur eingeführt und die Details der entsprechenden Hard- und Software erläutert. Zur Konkretisierung der dort vorgestellten abstrakten Konzepte der parallelen Datenverarbeitung mit GPUs enthält dieser Abschnitt eine Einführung in die Programmiersprache CUDA C. Als Beispiel und möglicher Baustein der Implementierung in Kapitel 5 wird eine CUDA-Variante eines häufig eingesetzten parallelen Algorithmus – die parallele Reduktion – vorgestellt.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Tabelle 4.1: Die Flynn'sche Taxonomie.

Zur groben Einordnung von Rechnerarchitekturen wird hier die Flynn'sche Taxonomie verwendet. Dazu wird nach der Anzahl gleichzeitig ausführbarer Anweisungen (engl. *Instructions*) sowie Speicherzugriffe kategorisiert [20]. Die Taxonomie ist in Tabelle 4.1 dargestellt. Klassische Hauptprozessoren mit einem Kern gehören nach Flynn also zu den SISD-Architekturen, da sie alle Instruktionen und Speicherzugriffe sequenziell ausführen. SIMD-Maschinen führen ihre Instruktionen sequenziell aus, können diese aber auf mehrere Operanden gleichzeitig anwenden. Können verschiedene Instruktionen parallel auf mehrere Daten angewendet werden, so spricht man von MIMD-Rechnern. Die MISD-Rechner, die verschiedene Instruktionen parallel auf ein und denselben Operanden anwenden sind ein theoretisches Konstrukt und kommen in der Praxis nicht vor.

4.1 Parallele Registermaschinen und das Work-Time Paradigma

Die *Parallele Registermaschine* (engl. *Parallel Random Access Machine*, PRAM) oder ist das grundlegende Berechnungsmodell zur Analyse paralleler Algorithmen. Formal ist eine PRAM eine

[5] Stand: März 2011.

Turingmaschine, die für eine gegebene Eingabe mit logarithmischem Speicherplatzbedarf die Anzahl P benötigter Prozessoren sowie eine Folge von Registermaschinen-Programmen berechnet und diese parallel auf die Eingabedaten anwendet [34, 35]. Da sich die Programme der einzelnen Prozessoren theoretisch vollständig voneinander unterscheiden können, gehören PRAMs in die Kategorie der MIMD-Rechner. Abhängig vom Speicherzugriffsverhalten wird zwischen verschiedenen Typen von PRAMs unterschieden. In *Exclusive-Read-Exclusive-Write* (*EREW*) PRAMs können die Daten an einer Speicheradresse von jeweils einem Prozessor gleichzeitig gelesen oder geschrieben werden. Bei *Concurrent-Read-Exclusive-Write* (*CREW*) PRAMs können diese zwar von allen Prozessoren gleichzeitig gelesen, aber von nur einem geschrieben werden. Können zusätzlich beliebig viele Prozessoren gleichzeitig an dieselbe Adresse schreiben heißt das entsprechende Berechnungsmodell *Concurrent-Read-Concurrent-Write* (CRCW) PRAM. Die in dieser Diplomarbeit betrachtete, parallele Architektur entspricht mit leichten Einschränkungen (s. Abs. 4.2.1) einer CRCW PRAM Architektur.

Die formal korrekte Notation zur Formulierung und Analyse nicht-trivialer PRAM-Algorithmen ist vergleichsweise aufwendig. Das *Work-Time-* (WT) *Paradigma* stellt eine gleichwertige, komfortablere Alternative dar. Im Grunde sind WT-Algorithmen mit gewöhnlichen, sequenziellen Registermaschinenprogrammen identisch. Jedoch kann mit einem speziellen Statement (`forall`) angezeigt werden, dass die folgende Instruktion bzw. Methode für eine beliebige, aber feste Anzahl an Instanzen, parallel ausgeführt werden soll und zwar ungeachtet einer konkreten Anzahl P vorhandener Prozessoren.

Das WT-Paradigma verwendet zwei von der Eingabelänge n abhängige Komplexitätsmaße. Zum einen die *Arbeitskomplexität* $W(n)$, sie entspricht der Gesamtanzahl ausgeführter Operationen. Zum anderen die *Schrittkomplexität* $S(n)$, sie entspricht der Anzahl von Schritten, wobei jeder Schritt aus einer oder mehr simultan durchgeführter Operationen besteht. Die Schrittkomplexität entspricht der klassischen Laufzeitkomplexität eines sequenziellen Algorithmus, da ein WT-Algorithmus nach $S(n)$ Schritten terminiert. Definiert man $W_i(n)$ als die Anzahl parallel ausgeführter Operationen in Schritt i, ergibt sich die Arbeitskomplexität als Summe der $W_i(n)$ (s. Gleichung 4.1).

$$W(n) = \sum_{i=1}^{S(n)} W_i(n) \tag{4.1}$$

Das die Vernachlässigung der Anzahl tatsächlich benötigter Prozessoren keine Auswirkung auf die Berechnungskraft des darunterliegenden Berechnungsmodells hat, zeigt das folgende Theorem.

Theorem 4.1. *Brent's Theorem. Ein Work-Time Algorithmus mit Schrittkomplexität $S(n)$ und Arbeitskomplexität $W(n)$ kann durch eine P-Prozessor PRAM in $\mathcal{O}\left(S(n) + \left\lfloor \frac{W(n)}{P} \right\rfloor\right)$ Schritten simuliert werden. [34, S.27]*

Beweis. Die P Prozessoren der PRAM arbeiten den i-ten Schritt eines WT-Algorithmus in $\left\lceil \frac{W_i(n)}{P} \right\rceil$ Berechnungsschritten ab. Gleichung 4.2 zeigt, wie dieser Fakt ausgenutzt werden kann, um eine obere Schranke für die Gesamtlaufzeit herzuleiten.

$$\sum_{i=1}^{S(n)} \left\lceil \frac{W_i(n)}{P} \right\rceil \leq \sum_{i=1}^{S(n)} \left(\left\lfloor \frac{W_i(n)}{P} \right\rfloor + 1 \right) \leq S(n) + \left\lfloor \frac{W(n)}{P} \right\rfloor \tag{4.2}$$

■

Da die Anzahl der Prozessoren eines konkreten Parallelrechners stets fest ist und die im nächsten Abschnitt vorgestellte parallele Architektur der Work-Time-Notation sehr ähnlich ist, wird letztere zur Analyse der Komplexität der in Kapitel 5 entwickelten parallelen Algorithmen verwendet.

Die Work-Time-Komplexität ist ebenso für sequenzielle Algorithmen geeignet. In diesem Fall wird in jedem Schritt genau eine Operation ausgeführt und es gilt $W_i(n) = 1$. Damit folgt aus Gleichung 4.1 direkt $W(n) = S(n)$ und die Work-Time Komplexität ist mit der gewöhnlichen Laufzeitkomplexität identisch.

4.2 CUDA

CUDA (*Compute Unified Device Architecture*) bezeichnet eine GPGPU Architektur von NVidia. Sie ermöglicht die Verwendung von Grafikprozessoren in nicht-Grafik-Anwendungen und erlaubt die Implementierung hochparalleler Algorithmen mit gängigen Sprachen wie C, C++ oder Fortran. Der CUDA-Toolkit, bestehend aus Compiler, Linker, Assembler und Debugger ist frei verfügbar[6] und auf den Betriebssystemen Windows, Linux und MacOS lauffähig. Zur Übersetzung von CUDA-Programmen wird keine besondere Hardware benötigt. Diese ist nur zur Ausführung von Programmen auf der GPU erforderlich. Ist eine solche nicht verfügbar, so kann ein Emulationsmodus verwendet werden, um eine sequenzielle Ausführung auf der CPU zu erzwingen. Die CUDA-Hardware kann als Co-Prozessor mit eigenem Hauptspeicher betrachtet werden. Hard- und Software wurden laut NVidia [50] unter Berücksichtigung der Faktoren *Datenparallelität, Datenlokalität* sowie *Berechnungsintensität* entworfen.

Datenparallelität bedeutet hier, dass eine Funktion (oder ein Algorithmus) für verschiedene Argumente parallel ausgewertet werden kann, ohne dass eine Datenabhängigkeit zwischen den einzelnen Funktionsaufrufen besteht. Eine Funktionen f heißt *datenabhängig* von einer Funktion g, falls sich $f(x)$ als $h(g(x))$ darstellen lässt, für eine beliebige Funktion h. In diesem Fall kann f erst dann ausgewertet werden, wenn das Ergebnis von g vorliegt. Maximale Datenparallelität liegt bei einer Schrittkomplexität von $S(n) = 1$ vor, da in diesem Fall alle Operationen des Algorithmus parallel ausgeführt werden können (Beispiel: Vektoraddition). Die Analyse der Datenparallelität eines Algorithmus besteht in dieser Diplomarbeit aus der Identifikation von Datenabhängigkeiten.

Datenlokalität bedeutet, dass bereits verarbeitete Daten nicht gespeichert werden müssen. Der verwendete Speicher kann also nach der Bearbeitung eines Datensatzes wieder freigegeben werden. Dies ist bei gewöhnlichen iterativen Optimierungsverfahren beispielsweise nicht der Fall, da die Traingsmenge $\mathcal{T}$ mehrfach verarbeitet wird.

Berechnungsintensität entspricht der Anzahl arithmetischer Operationen eines Algorithmus pro Speicherzugriff.

CUDA erweitert die oben genannten Programmiersprachen um drei Abstraktionsmechanismen zur Implementierung paralleler Algorithmen, nämlich *Thread-Blöcke*, *gemeinsamen Speicher (*SMem*)* sowie *Barrierefunktionen*. Zur Organisation einer Vielzahl von Threads werden Thread-Blöcke eingeführt. Diese erlauben eine Ausnutzung feingranularer Daten- und Thread-Parallelität. Dazu muss ein algorithmisches Problem in grobe Teilprobleme partitioniert werden, welche unabhängig und parallel von einzelnen Thread-Blöcken gelöst werden können. Durch eine geschickte Partitionierung ist es so möglich, zusätzlich verfügbare Recheneinheiten zu verwenden, ohne den Algorithmus ändern zu müssen. Ein Teilproblem wird innerhalb eines Thread-Blocks von mehreren Threads gemeinsam bearbeitet. Die Kommunikation innerhalb eines Blocks erfolgt über den gemeinsamen Speicher und ermöglicht das kooperative Lösen der Teilprobleme durch die Threads. Durch den

[6] Ein Download der CUDA-Software ist auf der Internetpräsenz von NVidia möglich.
URL: http://developer.nvidia.com/page/home.html (Stand: 29.12.2010)

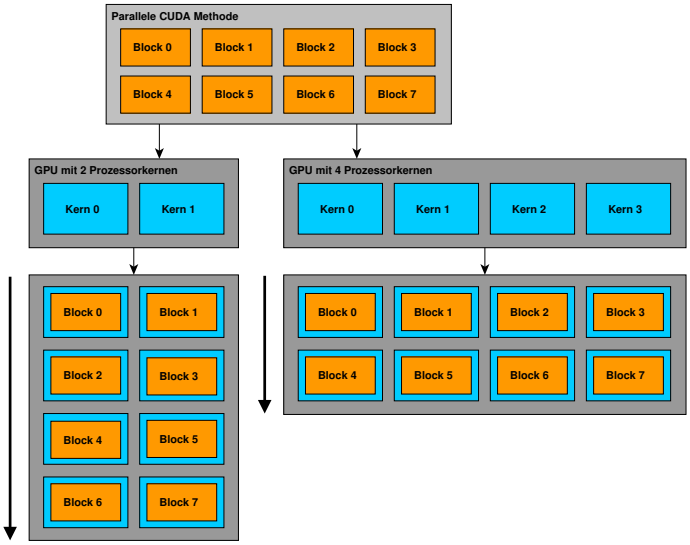


Abbildung 4.1: Schematische Darstellung der Ausführung einer CUDA-Methode mit acht Blöcken auf einer GPU mit zwei und einer mit vier Kernen. Da die Reihenfolge der Ausführung beliebig ist, können in beiden Fällen alle verfügbaren Kerne automatisch genutzt werden, ohne dass dies bei der Implementierung explizit berücksichtigt werden muss. In der dargestellten Situation wurde aus Gründen der Anschaulichkeit angenommen, dass ein Prozessorkern genau einen Block gleichzeitig bearbeiten kann. Genauere Informationen dazu finden sich in Abschnitt 4.2.5.

Aufruf von Barrierefunktionen kann die Ausführung aller Threads innerhalb eines Blocks synchronisiert werden. Somit wird sichergestellt, dass Änderungen des Speichers in allen Threads sichtbar sind.

Die Thread-Blöcke können von jedem verfügbaren Parallelprozessor, in beliebiger Reihenfolge, parallel oder sequenziell abgearbeitet werden, so dass CUDA-Programme grundsätzlich auf einer beliebigen Anzahl von Prozessorkernen ausgeführt werden können. Die tatsächliche Anzahl physikalisch vorhandener Prozessoren sowie auszuführender Thread-Blöcke muss daher erst zur Ausführungszeit bekannt sein. Der Scheduler der GPU weist allen verfügbaren Prozessorkernen Blöcke zu. Wie viele Blöcke dabei parallel bearbeitet werden, hängt von der konkreten Hardware sowie dem Ressourcenverbrauch eines Blocks ab (s. Abs. 4.2.5). Sobald ein Block terminiert, kann der Nächste bearbeitet werden. Die Ausführung der Blöcke erfolgt out-of-order, d.h. dass die Blöcke in einer nicht festgelegten Reihenfolge bearbeitet werden. Abbildung 4.1 soll diesen Zusammenhang veranschaulichen.

Da bereits mehrere Generationen von NVidia GPUs die CUDA-Architektur unterstützen, existieren verschiedene Hardware-Versionen. Die Unterschiede zwischen den einzelnen Versionen werden in Abschnitt 4.2.5 erläutert.

4.2.1 Architektur der GPU

Der größte Unterschied zwischen einer konventionellen CPU und einer GPU liegt in der Aufteilung der verfügbaren Transistoren. Ein Großteil der Transistoren einer CPU wird für den schnellen On-Chip-Speicher, den sog. *Cache*, verwendet – der entsprechende Speicher einer GPU ist im Vergleich dazu sehr klein. Der Cache aktueller CPUs beträgt bis zu 12 MiB, bei aktuellen GPUs liegt dieser bei max. 48 KiB. Stattdessen werden die "freien" Transistoren für Berechnungseinheiten, die sog. ALUs (*Arithmetic-Logical-Unit*) verwendet, wodurch eine schnelle Ausführung komplexer Berechnungen ermöglicht wird. Die schematische Darstellung in Abbildung 4.2 visualisiert diesen Unterschied. Hierbei ist allerdings zu beachten, dass aufgrund der hochparallelen Bauweise GPUs ($\approx 1,2$ GHz) wesentlich langsamer getaktet sind als CPUs (≈ 3 GHz) und die Geschwindigkeit

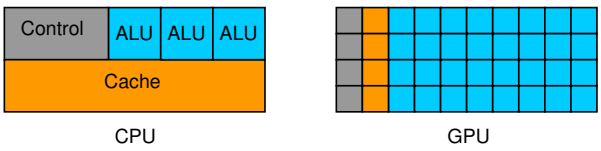


Abbildung 4.2: Schematische Darstellung der Transistornutzung bei CPU und GPU. Die Anzahl der ALUs einer GPU übersteigt die einer CPU bei Weitem zum Preis eines sehr kleinen on-Chip Speichers.

einer einzelnen GPU-ALU daher nicht mit der einer CPU vergleichbar ist. Damit ein Problem auf der parallelen GPU schneller als auf einer seriellen CPU gelöst werden kann, müssen daher zu jedem Zeitpunkt soviele GPU-ALUs wie möglich genutzt werden. Dies motiviert die Betrachtung der Datenparallelität des zu lösenden Problems bzw. der damit verbundenen Berechnungen.

CUDA-GPUs folgen der sog. *SIMT-* (*Single-Instruction-Multiple-Thread*) Architektur. Jede GPU besteht aus einem oder mehreren *Multiprozessoren* (MP). Jeder Multiprozessor besitzt eine feste Anzahl synchron arbeitender Recheneinheiten (ALUs) und erzeugt und verwaltet Threads in Gruppen der Größe 32. Diese Gruppen werden *Warps* genannt. Die Threads eines Warps starten zusammen an derselben Instruktionsadresse, besitzen aber einen eigenen Instruktionszähler und eigene Register, wodurch die Ausführung jedes Threads unabhängig von den anderen verzweigen kann. Ein *halber-Warp* entspricht der ersten oder zweiten Hälfte eines Warps. Wann immer ein Multiprozessor einen oder mehrere Thread-Blöcke zur Ausführung zugewiesen bekommt, partitioniert er diese selbstständig in Warps und reiht sie in seine Warteschlange ein. Dabei wird jeder Block auf dieselbe Art und Weise in Warps zerlegt; die Threads eines Blocks werden forlaufend durchnummeriert und in aufsteigender Reihenfolge in Warps eingeteilt. Die jeweilige Nummer eines Threads heißt auch *ThreadID*. Alle Threads eines jeden Warps können nur ein und dieselbe Instruktion gleichzeitig ausführen, so dass eine volle Ausnutzung eines Multiprozessors nur dann gewährleistet ist, wenn die Ausführungspfade aller 32 Threads eines Warps übereinstimmen. Kommt es bei einigen Threads innerhalb eines Warps zu einem bedingten Sprung und damit zu einer Divergenz der Ausführungspfade, wird die Ausführung jedes Pfades serialisiert. Dabei bleiben die Threads des jeweils anderen Pfades inaktiv, bis alle Threads auf denselben Ausführungspfad zurückkehren. Dies gilt nur für Threads innerhalb eines Warps; verschiedene Warps werden nebenläufig ausgeführt, unabhängig davon, ob sie denselben oder verschiedene Ausführungspfade einschlagen.

Die SIMT- ähnelt stark der SIMD-Architektur, bei welcher eine Instruktion gleichzeitig auf eine Vielzahl von Daten angewendet wird. Der Unterschied besteht darin, dass alle Recheneinheiten einer SIMD-Architektur von derselben Instruktion gesteuert werden, wohingegen SIMT-Instruktionen das Ausführungs- und Verzweigungsverhalten einzelner Threads steuern. Die Ausführungssemantik der beiden Architekturen ist laut NVidia [50, S.82] dieselbe, das heißt, dass die Unterschiede zwischen SIMD und SIMT im Hinblick auf die Korrektheit von Algorithmen ignoriert werden können. Für eine effiziente Ausführung des Codes sollte allerdings berücksichtigt werden, dass die Ausführungspfade der Threads innerhalb eines Warps, sofern dies möglich ist, nicht divergieren sollten. Daher handelt es sich um keine echte MIMD-Architektur, da dort alle Recheneinheiten vollkommen unabhängig voneinander sind.

4.2.2 Paralleler Methodenaufruf

In CUDA-Programmen können Methoden deklariert werden, bei deren Aufruf eine nahezu beliebige Anzahl an Instanzen der Methode nebenläufig auf der GPU ausgeführt werden. Solche Methoden werden mit Hilfe des `__global__`-Modifikators deklariert und werden *Kernel-Methoden*, *GPU-Kernel* oder einfach nur *Kernel* genannt. Ein CUDA-Thread entspricht einer Instanz des aufgerufenen Kernels. Die Anzahl zu startender Threads wird beim Methodenaufruf mit Hilfe der Syntax

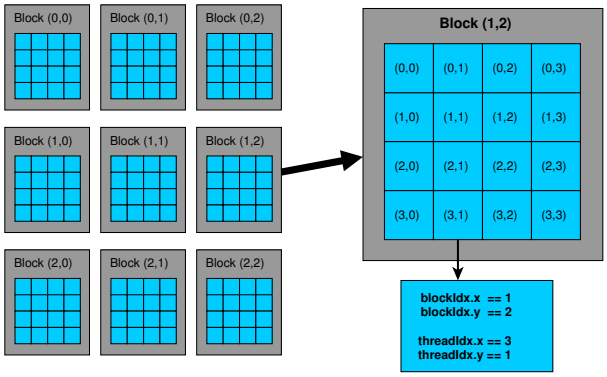


Abbildung 4.3: Visualisierung einer Ausführungskonfiguration mit 3x3 Blöcken (grau) und jeweils 4x4 Threads (blau). Die vergrößerte Ansicht von Block (1,2) zeigt die Indizierung der einzelnen Threads innerhalb eines Blocks. Die Position eines Threads im Grid kann im Programmcode mit Hilfe der eingebauten Variablen `blockIdx` und `threadIdx` abgefragt werden.

$$\texttt{Methodenname <<< Grid, Threads, SMem >>> (Parameter)} \tag{4.3}$$

spezifiziert. Dabei bezeichnet `Grid` die Anzahl der Thread-Blöcke, `Threads` die Anzahl der Threads pro Block und `SMem` die Größe des gemeinsamen Speichers pro Block. Der Ausdruck in den spitzen Klammern in (4.3) heißt *Ausführungskonfiguration* oder kurz *Konfiguration*. Die Angabe der Größe des gemeinsamen Speichers kann entfallen, sofern kein gemeinsamer Speicher benötigt wird (s. Abs. 4.2.4).

Mit Ausnahme dieser parallelen Methodenaufrufe ist die Ausführungssemantik von CUDA C mit derjenigen von gewöhnlichem C-Code identisch. Die Ausführungskonfiguration nimmt also die Rolle des in Abschnitt 4.1 erwähnten `forall`-Statements ein, was die Betrachtung von CUDA-Programmen als Work-Time-Algorithmen ermöglicht.

Im folgenden Code-Ausschnitt soll beispielhaft die Deklaration eines Gitters aus 1280 Blöcken mit jeweils 256 Threads und anschließendem Kernelaufruf gezeigt werden. Insgesamt werden $128 \cdot 10 \cdot 256 = 327680$ Instanzen der Methode `kernelMethod()` ausgeführt.

```
dim3 Gitter(128,10,1);           // 128 x 10 x 1 = 1280 Blöcke
dim3 InstanzenProBlock(16,8,2);  //  16 x  8 x 2 = 256 Threads
kernelMethod<<<Gitter, InstanzenProBlock>>>();
```

Gitter- und Blockgröße werden mit Hilfe des CUDA-Typs `dim3` deklariert. Dieser entspricht einer C-Struktur bestehend aus drei Variablen, `x`, `y` und `z`, vom Typ `int`. Ein Gitter darf höchstens zweidimensional sein. Das heißt, dass die `z`-Variable von Grids stets 1 sein muss. Die Gittergröße ist abhängig von der Größe der zu verarbeitenden Daten sowie der Anzahl tatsächlich vorhandener Recheneinheiten zu wählen. Blöcke können ein-, zwei- oder auch dreidimensional deklariert werden. Dabei dient die mehrdimensionale Indizierung der reinen Anschaulichkeit und ermöglicht eine natürliche Formulierung von Vektor-, Matrix- oder Volumenberechnungen. Für den Scheduler der GPU ist es dabei irrelevant ob 16x16 oder 256x1 Blöcke deklariert werden, gleiches gilt für die Threads. Die maximale Anzahl an Threads eines Blocks ist abhängig von der Hardware-Version entweder 512 oder 1024 – unabhängig davon wie viele Kerne die GPU tatsächlich besitzt.

Insgesamt stellt CUDA drei Modifikatoren zur Verfügung, mit deren Hilfe spezifiziert werden kann, ob eine Funktion auf der CPU oder der GPU ausgeführt werden soll und von wo aus sie aufgerufen werden kann. Der `__global__`-Modifikator deklariert eine Funktion als Kernel. Diese

werden auf der GPU ausgeführt und können nur von der CPU aufgerufen werden. Jeder Aufruf einer `__global__`-Funktion benötigt eine eigene Konfiguration.

Mit dem `__device__`-Modifikator deklarierte Funktionen repräsentieren klassische Subroutinen, wobei rekursive Aufrufe nur in den neusten Hardware-Versionen möglich sind. Diese werden auf der GPU ausgeführt und können nur von GPU-Methoden aus aufgerufen werden. Sie werden nicht konfiguriert, sondern übernehmen die Ausführungskonfiguration der aufrufenden Funktion. Werden, wie im folgenden Codebeispiel, zwei Kernel mit derselben Konfiguration aufgerufen,

```
kernelMethode1<<<Gitter, InstanzenProBlock>>>(..);
kernelMethode2<<<Gitter, InstanzenProBlock>>>(..);
```

so können diese mit Hilfe von `__device__`-Funktionen zu einem Kernelaufruf zusammengefasst werden.

```
__global__ void kernelMethode(..){
  deviceMethode1(..);
  deviceMethode2(..);
}
..
kernelMethode<<<Gitter, InstanzenProBlock>>>(..);
```

Innerhalb einer jeden CUDA-Methode (egal ob Kernel oder `__device__`-Methode) kann mit Hilfe der eingebauten Variablen `blockIdx` und `threadIdx` vom Typ `dim3`, auf den Index eines Blocks innerhalb des Grids bzw. eines Threads innerhalb seines Blocks zugegriffen werden. Diese sind der einzige Weg, um Blöcke und Threads innerhalb des Codes zu unterscheiden. Mit ihnen ist es möglich, den Kontrollfluss innerhalb eines Kernels divergieren zu lassen. Beispielsweise kann mit Hilfe einer bedingten Anweisung erreicht werden, dass ein bestimmter Teil des Codes nur vom Block mit dem Index 3 (`blockIdx.x==3`) oder von allen Threads, deren Index kleiner als 16 (`threadIdx.x<16`) ist, ausgeführt wird. Darüber hinaus kann die Dimension des Gitters, also die Anzahl parallel ausgeführter Blöcke, zur Ausführungszeit im Code über die Variable `gridDim` und die Anzahl ausgeführter Threads über die Variable `blockDim` abgefragt werden. Beide sind ebenfalls vom Typ `dim3`.

Zur Kennzeichnung einer Funktion die auf der CPU ausgeführt werden soll wird der `__host__`-Modifikator verwendet. Diese Funktionen können nur von der CPU aufgerufen werden. Es besteht also keine Möglichkeit CPU-Funktionen vom GPU aus aufzurufen. Wird keiner der drei Modifikatoren verwendet, so wird automatisch eine `__host__`-Funktion erzeugt. Eine Funktion kann sowohl den `__device__`- als auch den `__host__`-Modifikator haben und ist damit sowohl auf der CPU als auch auf der GPU ausführbar. Es ist jedoch nicht möglich die `__global__`- und `__host__`-Modifikatoren zusammen zu verwenden.

Im Folgenden wird die Programmierung mit CUDA C anhand eines Beispiels erläutert.

4.2.3 Beispiel: Vektoraddition

Hier werden zwei Vektoren `a` und `b` addiert und das Ergebnis im Vektor `c` gespeichert. Damit sollen die grundlegenden Konzepte der Programmierung mit CUDA C verdeutlicht werden. Der folgende Codeausschnitt ist ein Kernel zur parallelen Addition zweier Vektoren.

```
// Kernel Definition
__global__ void VektorAdditionKernel1(float* a, float* b, float* c) {
```

```
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
..
VektorAdditionKernel<<<1, d>>>(a, b, c);
```

An der Ausführungskonfiguration ist zu erkennen, dass ein Block mit `d` Threads verwendet wird. Jeder Thread addiert jeweils eine Komponente der beiden Vektoren und schreibt seine Summe in den Ergebnisvektor. Die Länge der als Argumente übergebenen Vektoren ist innerhalb des Kernels unbekannt. In diesem Fall muss in der aufrufenden Methode sichergestellt werden, dass die Anzahl gestarteter Threads gleich der Länge der Vektoren `a`, `b` und `c` ist, da es ansonsten zu Speicherzugriffsfehlern kommen kann. Ist P die Anzahl verfügbarer Prozessoren und d die Dimension der Vektoren, so beträgt die Schrittkomplexität $S(n) = 1$, da jeder Thread nur eine Addition ausführt. Da insgesamt d Additionen ausgeführt werden müssen, beträgt die Arbeitskomplexität $W(n) = d$.

Im Hinblick auf Kapitel 5 muss hierbei betont werden, dass die Anzahl der Threads eines Kernelaufrufs für jeden Block gleich ist, was wiederum Konsequenzen für den Algorithmenentwurf haben kann. Sollen beispielsweise N Vektoradditionen $\mathbf{c}^{(i)} := \mathbf{a}^{(i)} + \mathbf{b}^{(i)}$, $1 \leq i \leq N$ nebenläufig ausgeführt werden, so kann der oben gezeigte Kernel leicht auf N Blöcke erweitert und die Vektoren mit Hilfe zweidimensionaler Arrays repräsentiert werden. Sind die Dimensionen unterschiedlicher Vektorpaare jedoch ungleich, also $\mathbf{a}^{(i)}, \mathbf{b}^{(i)} \in \mathbb{R}^{d^{(i)}}$, $d^{(i)} \in \mathbb{N}$, so hat jeder Block eine unterschiedliche Anzahl Summationen auszuführen. Im folgenden Codeausschnitt führen die Threads eines Blocks nur dann eine Summation aus, wenn ihr Index kleiner als die Länge des zu addierenden Vektors ist. Dazu müssen $d = \max_{i \leq N} d^{(i)}$ Threads verwendet werden.

```
__global__ void VektorAdditionKernel2(float** a, float** b, float** c,
                                      int* d) {
  int i = gridIdx.x;
  int j = threadIdx.x;

  if(i < d[i]){
    c[i][j] = a[i][i] + b[j][i];
  }
}
```

Die Schrittkomplexität beträgt $S(n) = \mathcal{O}(1)$ und die Arbeitskomplexität $W(n) = \mathcal{O}(dN)$. Allerdings werden in diesem Fall Threads *verschwendet*. Threads, deren Index `j` größer als die Dimension der zu addierenden Vektoren ist, führen keine Addition aus und hätten daher gar nicht erst gestartet werden sollen. Eine häufig angewendete Alternative bietet die Verwendung sog. *Worker-Threads*. In diesem Fall entspricht die Anzahl der Threads pro Block einer vorher festgelegten Konstanten W. Das folgende Codebeispiel zeigt die Vektoraddition mit Worker-Threads, wobei jeder Thread des i-ten Blocks $\left\lceil \frac{d^{(i)}}{W} \right\rceil$ Additionen ausführt.

```
__global__ void VektorAdditionKernel3(float** A, float** B, float** C,
                                      int* d) {
  int i = gridIdx.x;
  int j = threadIdx.x;

  while(j < d[i]){
    C[i][j] = A[i][j] + B[i][j];
    j += blockDim.x;
```

```
  }
}
```

Diese Verwendung von Worker-Threads entspricht der expliziten Anwendung von Brent's Theorem. Falls es die Problemstellung erfordert, können analog dazu *Worker-Blöcke* verwendet werden. Die Anzahl der Worker-Threads bzw -Blöcke ist zwar grundsätzlich beliebig, sollte aber unter Berücksichtigung der Hardware gewählt werden. Details dazu sind in Abschnitt 4.2.5 zu finden.

Das letzte Code-Beispiel zeigt exemplarisch den Aufruf des oben definierten Kernels `VektorAdditionKernel()`. Der `__host__`-Modifikator wird hier nur zu Anschauungszwecken verwendet.

```
__host__ float** VektorAddition(float** a, float** b, int* d, int N) {
  int W_M = 32;

  dim3 Grid(1,1,1);
  dim3 Threads(W_M,1,1);

  float **c;
  // Initialisiere GPU-Speicher..

  VektorAdditionKernel3<<<Grid, Threads>>>(a, b, c, d, N);
  // Kopiere Daten vom GPU-Speicher in den RAM der CPU..
  return c;
}
```

4.2.4 Speicher

Im diesem Abschnitt wird die Speicherhierarchie von CUDA betrachtet. Jede CUDA-Hardware verfügt über einen eigenen Hauptspeicher, den sog. *globalen Speicher*, mit Größen zwischen 32 MiB und 6 GiB. Darüber hinaus besitzt jeder Multiprozessor bis zu 48 KiB gemeinsamen Speicher, 64 KiB *Konstanten-Speicher* sowie 8192 bis 32768 *Register*. Diese Ressourcen werden auf alle Blöcke und Threads, die nebenläufig von einem Multiprozessor ausgeführt werden, aufgeteilt. Die genaue Anzahl an Multiprozessoren und verfügbarem Speicher ist abhängig von der CUDA-Hardware-Version. Eine Übersicht über die Spezifikationen der verschiedenen Hardware-Versionen findet sich in Abschnitt 4.2.5.

Globaler Speicher. Die Daten von CUDA-Threads können in verschiedenen Speicherbereichen abgelegt werden. Der globale Speicher ist mit bis zu 6 GiB – meistens zwischen 512 MiB und 1 GiB – der größte verfügbare Speicherbereich. Er ist mit einem bis zu 512 Bit breiten Bus an die GPU angebunden und in 32, 64 sowie 128 Byte große Segmente partitioniert. Trotz interner Übertragungsraten von bis zu 192 GiB/s sind Zugriffe auf den globalen Speicher als "langsam" einzustufen, da die Latenz eines Zugriffs auf den globalen Speicher zwischen 400 und 800 Taktzyklen beträgt. *Latenz* bezeichnet die Anzahl an Taktzyklen die vergehen, bis ein Warp seine nächste Instruktion ausführen kann. Um diese Wartezeit zu verkürzen, ist die GPU in der Lage globale Speicherzugriffe innerhalb eines Warps zu einer oder mehreren Transaktionen zu verschmelzen. Ob die Zugriffe eines Warps verschmolzen werden können, hängt von der Länge der angefragten Binärworte sowie der Verteilung der Speicheradressen ab. Grundsätzlich reduziert jede Transaktion, deren Resultat nicht vollständig verwendet wird, den Speicherdurchsatz der GPU. Abbildung 4.4 verdeutlicht dies anhand zweier Speicherzugriffsszenarien. Die exakten Zugriffsstrategien der einzelnen CUDA-Hardware-Versionen sollen hier nicht weiter betrachtet werden, diese sind in [50]

zu finden. Globale Speicherzugriffe sind seit der CUDA-Hardware-Version 2.0 gecached. Dies ermöglicht eine Verringerung der Latenz durch Ausnutzung lokaler Zugriffsmuster. Trotzdem sollen die in Abschnitt 5.3 entworfenen Datenstrukturen die Vorgaben für die Verschmelzung von Speicherzugriffen berücksichtigen, um auch auf älteren Hardware-Versionen eine effiziente Ausführung zu ermöglichen. Sollen zwei oder mehr Threads eines Warps einen Schreibzugriff auf dieselbe Speicheradresse ausführen, so wird dieser nur von einem tatsächlich ausgeführt. Welcher Thread letztendlich auf die angeforderte Adresse schreibt ist undefiniert. Auf neueren Hardware-Versionen kann dieses Problem mit Hilfe von automatisch serialisierten, sog. *atomaren Schreibzugriffen* gelöst werden. Sind diese nicht verfügbar, so muss innerhalb des jeweiligen Algorithmus darauf geachtet werden, dass nie mehr als ein Thread gleichzeitig an dieselbe Adresse schreibt. Der im Vergleich zu CPUs geringe Hauptspeicher sowie die hohen Latenzen globaler Speicherzugriffe motivieren die Betrachtung von Datenlokalität und Berechnungsintensität beim Entwurf paralleler Algorithmen.

Kernel müssen den Rückgabetyp `void` haben, sie liefern also kein Ergebnis zurück. Daher muss ihr Ergebnis vom globalen Speicher in den Hauptspeicher der CPU kopiert werden. GPU-Kernel werden asynchron zum Kontrollfluss der CPU ausgeführt, das heißt, dass der Kontrollfluß an die CPU übergeben wird, bevor die GPU ihre Ausführung beendet hat. Die nebenläufige Ausführung von CPU- und GPU-Code findet außerdem beim Kopieren von Daten innerhalb des globalen Speichers der GPU sowie beim Kopieren von weniger als 64 KiB Daten vom Hauptspeicher der CPU in den globalen Speicher der GPU statt.

Das Reservieren von globalem Speicher erfolgt durch den Aufruf der Methode

```
cudaMalloc(void** target, size_t size),
```

wobei nach erfolgreicher Allokation der Zeiger `*target` auf einen `size` Byte großen Abschnitt des globalen Speichers zeigt [51].

Zur Organisation des Hauptspeichers der CPU bietet CUDA die Methode

```
cudaHostAlloc(void** target, size_t size, unsigned int flags),
```

an. Im Gegensatz zu `malloc(size_t size)` kann mit `cudaHostAlloc(..)` nicht-auslagerungsfähiger Hauptspeicher reserviert, sowie Abschnitte des CPU Hauptspeichers in den Speicherbereich der GPU abgebildet werden. Ersteres erhöht den Speicherdurchsatz, führte aber während der Entwicklung der in Kapitel 5 vorgestellten Implementierung zu Systeminstabilitäten und wird nicht von allen Hardware-Versionen unterstützt. Letzteres ist nur sinnvoll, falls der globale Speicher sehr klein (<128 MiB) ist, da ein CPU Hauptspeicherzugriff der GPU eine verhältnismäßig geringe Bandbreite von 2 bis 6 GiB/s aufweist. Diese geringe Bandbreite der Übertragung von Daten vom Hauptspeicher der CPU in den globalen Speicher der GPU stellt den Flaschenhals eines jeden GPU Programms dar. Trotzdem ist ein initiales Kopieren der Eingabedaten unerlässlich. Ein solcher Kopiervorgang erfolgt durch den Aufruf der Methode

```
cudaMemcpy(void* target, const void* source, size_t size, enum cudaMemcpyKind
transferModus).
```

Dabei werden `size` Byte an Daten von der Quelladresse `source` an die Zieladresse `target` kopiert. Der `transferModus` gibt dabei die Richtung des Kopiervorgangs an, das heißt falls der `transferModus`

- `cudaMemcpyHostToDevice` ist, wird `source` als Zeiger in den Hauptspeicher der CPU und `target` als Zeiger in den globalen Speicher der GPU interpretiert,

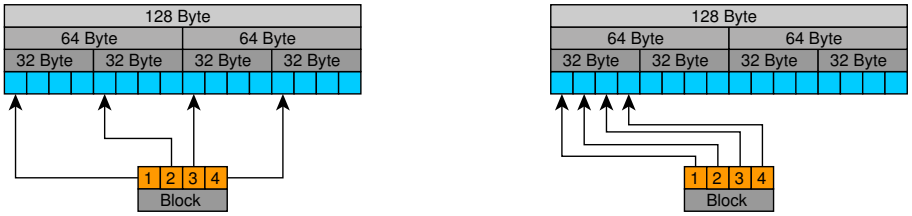


Abbildung 4.4: Schematische Dargestellung von Zugriffsmustern auf 64 Bit Worte. Links liegen die Adressen genau 32 Byte auseinander. Unabhängig davon ob die vier Anfragen zu vier 32 Byte, zwei 64 Byte oder einer 128 Byte Transaktion verschmolzen werden, bleiben die übrigen 348 Byte der ausgeführten Transaktionen stets ungenutzt. Rechts liegen die angefragten Adressen der vier Threads im gleichen 32 Byte Segment und können daher zu einer 32 Byte Transaktion zusammengefasst werden.

- `cudaMemcpyDeviceToHost` ist, wird `source` als Zeiger in den globalen Speicher der GPU und `target` als Zeiger in den Hauptspeicher der CPU interpretiert,
- `cudaMemcpyDeviceToDevice` ist, werden sowohl `source` als auch `target` als Zeiger in den globalen Speicher der GPU interpretiert.

Ist der Speicher voll, oder liegen Quell- oder Zieladresse beim Kopieren außerhalb des addressierbaren Bereichs, so schlägt die aufgerufene Methode fehl. Der zuletzt aufgetretene CUDA-Fehlercode kann zu jeder Zeit von der CPU über die Methode `cudaGetLastError()` abgefragt werden. Die Fehlercodes sind vom Typ `cudaError_t`. Wird der Code `cudaSuccess` zurückgegeben, so ist kein Fehler aufgetreten. Eine lesbare Form des Fehlercodes kann mit Hilfe der Methode `cudaGetErrorString(cudaError_t error)` erzeugt werden.

Lokaler-, Textur- und Konstantenspeicher. Jeder Thread besitzt 16 bis 512 KiB *lokalen Speicher* für statische Arrays und Variablen, die nicht in die Register des Multiprozessors passen. Die Bezeichnung "lokaler Speicher" ist insofern irreführend, als dass damit lediglich ein reservierter Bereich des globalen Speichers gemeint ist. Der Name entsteht aus der Tatsache, das im lokalen Speicher ausschließlich lokale Variablen eines Kernels abgelegt werden. Dieser Speicherbereich kann nicht explizit verwaltet werden und wird ausschließlich vom Compiler verwendet. Des Weiteren besteht die Möglichkeit, einige Bereiche des globalen Speichers als *Textur-Speicherbereich* zu deklarieren. Diese Bereiche besitzen zwar auf allen Hardware-Versionen einen 6 bis 8 KiB kleinen Cache, können von der GPU aber nicht beschrieben werden. Der Einsatz von Textur-Speicher ist auf die Filterung von Pixeldaten ausgelegt und unterstüzt beispielsweise die automatische Umwandlung von 8 oder 16 Bit Integer in 32 Bit Gleitkommadarstellung. Da solche Funktionen für allgemeine Algorithmen in der Regel nicht relevant sind, wird der Textur-Speicherbereich hier nur der Vollständigkeit halber erwähnt.

Der Konstanten-Speicher ist ein 64 KiB kleiner gecachter Bereich des globalen Speichers. Er kann nur von der CPU beschrieben werden und ist für Konstanten vorgesehen. Ab Hardware-Version 2.0 werden die Argumente von Kernelmethoden automatisch im Konstantenspeicher abgelegt. Variablen die explizit im Konstantenspeicher abgelegt werden sollen, müssen mit dem Modifikator `__constant__` deklariert werden.

Gemeinsamer Speicher. Jeder Multiprozessore einer GPU besitzt bis zu 48 KiB gemeinsamen Speicher. Da dies nur für die neusten Hardware-Versionen zutrifft, wird in dieser Diplomarbeit stets von 16 KiB ausgegangen, da dieser in allen Versionen zur Verfügung steht. Der SMem dient der Kommunikation der Threads und weist einige Besonderheiten auf. Daten im globalen, Konstanten- und Texturspeicher bleiben solange persistent bis sie überschrieben werden, der betroffene Speicherbereich freigegeben wird oder das Programm terminiert. Dies gilt jedoch nicht

Algorithmus 5 Ein GPU-Kernel, der zwei Arrays im gemeinsamen Speicher verwendet. Dazu wird ein Array für den gesamten, in der Ausführungskonfiguration angeforderten, SMem deklariert und innerhalb des Kernels zwei Zeiger auf dieses Array erzeugt und mit unterschiedlichen Adressen initialisiert. Hätte der angeforderte Speicher beispielsweise die Größe `16*sizeof(int)==64` Byte, so wären `g1` und `g2` Zeiger auf jeweils 32 Byte lange Speicherbereiche, also 8 Integer. Ändert ein Thread nun eines der Arrays, so ist diese Änderung in allen Threads des Warps sofort und nach dem Aufruf der Funktion `__syncthreads()` in allen Threads des Blocks verfügbar.

```
 1: extern __shared__ int gemeinsamerSpeicher[];
 2:
 3: __global__ void dieGPUMethode(float* dieWerte, int k){
 4:
 5:   volatile int* g1 = gemeinsamerSpeicher;
 6:   volatile int* g2 = gemeinsamerSpeicher + 8;
 7:   gemeinsameBerechnung(g1, g2);
 8:   __syncthreads();
 9:   ..
10: }
```

für den gemeinsamen Speicher. Dieser kann ausschließlich von den Threads innerhalb eines Blocks gelesen und beschrieben werden. Es besteht also keine Möglichkeit von einer `__host__`-Methode Daten vom Hauptspeicher der CPU direkt in den gemeinsamen Speicher eines Thread-Blocks zu kopieren. Eine Änderung des gemeinsamen Speichers ist für alle Threads innerhalb eines Warps sichtbar. Um das Resultat eines Schreibzugriffs für alle Threads eines Blocks sichtbar zu machen, muss im Anschluss an den Schreibvorgang die eingebaute Methode `__syncthreads()` aufgerufen werden. Innerhalb eines Warps ist dabei keine Synchronisation notwendig. Das heißt alle Schreibzugriffe auf den gemeinsamen Speicher sind sofort für alle Threads des jeweiligen Warps sichtbar.

Der SMem ist wesentlich schneller als der globale Speicher, da er direkt in der GPU verbaut ist. Er ist in 16 Speicherbänke unterteilt, so dass aufeinanderfolgende Worte in aufeinanderfolgenden Bänken liegen. Jeder Multiprozessor ist in der Lage, auf alle Bänke seines gemeinsamen Speichers simultan zuzugreifen, wobei jede Speicherbank eine Bandbreite von 32 Bit pro zwei Taktzyklen besitzt. Um einen maximalen Datendurchsatz zu erzielen, muss sichergestellt werden, dass alle simultan angefragten Speicheradressen in verschiedenen Bänken liegen. Fallen zwei oder mehr Zugriffe auf dieselbe Bank, so werden die Zugriffe auf diese Bank serialisiert. Ist n die Anzahl der Zugriffe auf dieselbe Bank, so spricht man von einem n-Bank-Konflikt. Jede Anfrage eines Warps an den SMem wird in eine Anfrage pro halb-Warp aufgeteilt und unabhängig ausgeführt. Infolgedessen kann es keinen Bank-Konflikt zwischen Speicherzugriffen des ersten und des zweiten halb-Warps geben. Der Fall, in dem mehrere Threads eines Warps dasselbe 32 Bit Wort lesen, stellt eine Ausnahme dar. Solche Zugriffe werden mit Hilfe eines sog. *Broadcasts* ausgeführt und verursachen keinen Bank-Konflikt. Müssen zwei oder mehr Threads eines Warps einen Schreibzugriff auf dieselbe Speicheradresse ausführen, so wird dieser nur von einem tatsächlich ausgeführt. Welcher Thread letztendlich auf die angeforderte Adresse schreibt ist undefiniert. Der gemeinsame Speicher eines Blocks wird bei der Terminierung des ausgeführten Kernels automatisch freigegeben. Die Latenz eines Zugriffs auf den gemeinsamen Speicher ist laut NVidia vergleichbar mit der eines Registerzugriffs.

Das Reservieren von gemeinsamen Speicher erfolgt durch die Deklaration eines externen Arrays mit dem Schlüsselwort `__shared__`. Algorithmus 5 zeigt wie der gemeinsame Speicher innerhalb eines Kernels genutzt werden kann. Das Schlüsselwort `volatile` verhindert dabei, dass der Com-

piler die Variablen in Registern ablegt. Damit wird sichergestellt, dass die Änderung eines so deklarierten Speicherbereichs auch wirklich in den gemeinsamen Speicher geschrieben wird.

Die geringe Größe des gemeinsamen Speichers stellt eine nicht zu vernachlässigende Restriktion dar. So stehen jedem Thread-Block bei Verwendung des Datentyps `double` (64 Bit Fließkommazahl) und einem 16 KiB kleinen gemeinsamen Speicher maximal 2048 Speicherplätze zur Verfügung.

	Hardware-Version					
Funktion	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**	**2.1**
Anz. MPs pro GPU	12, 16	1, 2, 4, 6, 8, 12, 14, 16	2, 6, 9, 12	24, 30	8, 10, 11, 14, 15, 16	1, 2, 3, 4, 6, 7
Anz. Kerne pro MP	8				32	48
Max. Anzahl Threads pro Block	512				1024	
Anz. Threads pro Warp	32					
Max. Anz. residenter Warps pro MP	24		32		48	
Anz. 32 Bit Register pro MP	8192		16384		32768	
Gemeinsamer Speicher pro MP	16 KiB				48 KiB	
Min. Anz. allokierter Blöcke	2				1	
Min. Anz. allokierter Register pro Block	256		512		64	
Min. Anz. allokierter Register pro Thread	8		16		2	
Min. allokierter gemeinsamer Speicher pro Block	512 Byte				128 Byte	
Anzahl parallel ausführbarer Kernel	1				16	
64 Bit Arithmetik	-			+		
Rekursive Aufrufe	-				+	
Cache für globalen Speicher	-				+	
Atomare Addition von Gleitkommawerten	-				+	

Tabelle 4.2: Unterschiede der CUDA-Hardware-Versionen.

4.2.5 Hardware-Versionen

Wie bereits deutlich geworden ist, unterliegen CUDA Soft- und Hardware einem ständigen Entwicklungsprozess, allerdings sichert NVidia eine Abwärtskompatibilität aktueller und zukünftiger Versionen zu. Diese Diplomarbeit bezieht sich auf das CUDA SDK Version 3.2. Die im Rahmen der Diplomarbeit entwickelte Implementierung wurde allerdings darauf ausgelegt auf allen CUDA-Hardware-Versionen ausführbar zu sein. Sie wurde auf den Versionen 1.3 und 2.0 getestet. Tabelle 4.2 zeigt eine Übersicht der Unterschiede aller bisher veröffentlichten Versionen. Einige dieser Werte können zur Ausführungszeit über die eingebaute Variable `cudaDeviceProp` abgefragt

werden. Die genauen Spezifikationen sind in [50] zu finden. Hierbei sind vor allem die Schranken interessant. Durch die Warpgröße wird implizit festgelegt, dass die Anzahl der in einer Ausführungskonfiguration angeforderten Threads pro Block ein vielfaches von 32 sein muss. Ist die gewählte Blockgröße kein Vielfaches der Warpgröße, so bleiben laut [50, S.88] Ressourcen ungenutzt. Dort wird ebenfalls erwähnt, dass die Anzahl residenter Warps und damit die Anzahl residenter Blöcke pro MP so hoch wie möglich sein sollte. Dies wird damit begründet, dass sich Warps, die auf einen globalen Speicherzugriff warten, im Leerlauf befinden. Ist die Anzahl residenter Warps pro MP allerdings hoch genug, so können während dieser Wartezeit die Instruktionen anderer Warps verarbeitet werden. Die maximale Anzahl residenter Blöcke pro MP ist von der Anzahl benötigter Register und der Größe des angeforderten gemeinsamen Speichers pro Block abhängig. Werden die in Tab. 4.2 angegebenen minimalen Größen für Register- und Speicherverbrauch eines Blocks nicht überschritten, so ergibt sich beispielsweise für die Hardware-Versionen 1.x eine maximale Anzahl von 32 Blöcken pro MP. Diese kann entweder über die minimale Anzahl allokierter Register pro Block oder dem minimalen Bedarf an SMem berechnet werden. Die hier entwickelte Implementierung wird sich an diesen Werten orientieren, damit sie in vielen Hardware-Versionen sinnvoll einsetzbar ist. Da die Anzahl der Multiprozessoren zwischen den einzelnen Versionen stark schwankt, wird in der hier entwickelten Implementierung die Möglichkeit bestehen, die Anzahl der verwendeten Blöcke anzupassen.

Zusätzlich sind bei der Implementierung weitere Einschränkungen zu beachten: Die Angabe von 48 KiB SMem bei den Versionen 2.x ist insofern abzuschwächen, als dass dieser teilweise für den Cache des globalen Speichers verwendet wird. Sind alle Caching-Funktionen der Hardware-Version 2.x aktiviert, so verfügen auch diese Multiprozessoren nur über 16 KiB SMem pro Multiprozessor. Daher wird in der Implementierung von 16 KiB gemeinsamen Speicher pro MP ausgegangen. Die atomaren Addition von Gleitkommawerten kann zwar in einigen Situationen eine Synchronisation der Threads ersparen, da diese allerdings nur von den neusten Versionen unterstützt wird, wurde auf die Verwendung in Kapitel 5 verzichtet. Selbiges gilt für die nebenläufige Ausführung verschiedener Kernel. Diese ist nach Aussagen von NVidia ohnehin nur für sehr kleine Kernel mit geringem Ressourcenverbrauch sinnvoll.

4.2.6 Beispiel: Parallele Reduktion

Parallele Reduktion ist die generische Parallelisierung assoziativer Funktionen. Sie wird immer dann verwendet, wenn eine assoziative Funktion auf eine Menge von Werten angewendet werden soll. Die Assoziativität ist dabei wichtig, da in diesem Fall die Reihenfolge der Auswertungen der Funktion keine Rolle spielt. Bei der Summenreduktion nimmt die Addition den Platz der assoziativen Funktion ein. Die Aufgabe ist es, ein Array von Zahlen aufzuaddieren. Ein SISD-Rechner würde in einer Schleife über das gesamte Array iterieren und die Werte sukzessive aufaddieren. Die Idee der parallelen Reduktion besteht nun darin, parallel mehrere Partialsummen zu bilden und diese abschließend zu addieren, um die Gesamtsumme zu erhalten. Algorithmus 6 zeigt eine CUDA-Implementierung der Summenreduktion von Mark Harris [28]. Die hier gezeigte Version verwendet 64 Blöcke mit jeweils 128 Threads und ist für Arrays der Länge $n \geq 16384$ ausgelegt. Man stelle sich vor, das zu summierende Array sei in Teile der Größe 256 partitioniert. Jeder Block reduziert dann einige dieser Teilarrays auf eine Partialsumme. Die so entstandenen 64 Werte können dann entweder sequenziell von der CPU oder einer erneuten parallelen Reduktion mit einem Block und 32 Threads aufsummiert werden. Der CUDA Code sowie die genaue Vorgehensweise werden im Folgenden erläutert.

In den Zeilen 4-8 werden die Variablen initialisiert. Das Array `smem` liegt im gemeinsamen Speicher, dessen Größe in der hier nicht dargestellten Ausführungskonfiguration auf die Anzahl der Threads pro Block gesetzt wird. Die Variable `tid` enthält die Nummer des jeweiligen Threads und `i` den Index des als nächstes zu addierenden Eintrags des Teilarrays des Blocks.

Algorithmus 6 CUDA Code der parallelen Summenreduktion.

```
 1: __global__ void reduce(float *g_idata, float *g_odata, unsigned int n){
 2:
 3:   float *smem = SharedMemory<float>();
 4:   unsigned int tid = threadIdx.x;
 5:   unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
 6:   unsigned int gridSize = blockSize*2*gridDim.x;
 7:   float mySum = 0;
 8:
 9:   for(; i < n; i += gridSize){
10:     mySum += g_idata[i];
11:     if (i + blockSize < n) mySum += g_idata[i+blockSize];
12:   }
13:
14:   smem[tid] = mySum;
15:   __syncthreads();
16:
17:    if (tid <  64) { smem[tid] = mySum = mySum + smem[tid + 64]; }
18:    __syncthreads();
19:
20:   if (tid < 32){
21:     smem[tid] = mySum = mySum + smem[tid + 32];
22:     smem[tid] = mySum = mySum + smem[tid + 16];
23:     smem[tid] = mySum = mySum + smem[tid +  8];
24:     smem[tid] = mySum = mySum + smem[tid +  4];
25:     smem[tid] = mySum = mySum + smem[tid +  2];
26:     smem[tid] = mySum = mySum + smem[tid +  1];
27:   }
28:   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
29: }
```

Die `gridSize` entspricht der Anzahl der Werte, die aufaddiert werden können, wenn jeder Thread genau zwei Zahlen des Eingabearrays addiert – in diesem Fall $64 \cdot 128 \cdot 2 = 16384$. Die Variable `mySum` fungiert als Akkumulator, in den jeder Thread das Ergebnis seiner Additionen ablegt. Die Schleife in den Zeilen 10-13 entspricht einer Worker-Thread Parallelisierung. Im Fall $n = 16384$ ist die Anzahl der Teilarrays gleich der Anzahl der Blöcke und die Schleife besteht aus genau einer Iteration, da die Anzahl zu summierender Werte mit der `gridSize` übereinstimmt. Innerhalb der Schleife addiert jeder Thread mindestens zwei Werte auf seinen Akkumulator. Ist $n > 16384$, so wird die `gridSize` auf den Index des nächsten zu summierenden Arrayeintrags addiert. Dies führt dazu, dass in der zweiten Iteration die Teilarrays aufsummiert werden, für die in der ersten Iteration nicht genug Blöcke vorhanden waren. Beispielsweise addieren die Threads des ersten Blocks in der ersten Iteration die Werte des ersten Teilarrays. In der zweiten Iteration addieren dieselben Threads die Werte des 65. Teilarrays, in der dritten die des 129. Teilarrays usw. solange, bis das komplette Eingabearray mit Blöcken überdeckt wurde. Nach der Terminierung der Schleife legen alle Threads eines Blocks den Wert ihres Akkumulators im gemeinsamen Speicher ab (Zeile 15). In Zeile 16 erfolgt eine Synchronisation der Threads, damit sichergestellt wird, dass die Änderungen des gemeinsamen Speichers im ganzen Block sichtbar sind. Zu diesem Zeitpunkt ist das komplette Eingabearray aus dem globalen Speicher eingelesen und jeder Block hat noch

128 Partialsummen im gemeinsamen Speicher zu addieren. In Zeile 18 führen nur die ersten 64 Threads eine Addition aus und schreiben ihre Summe zurück in den gemeinsamen Speicher. Dabei addiert der Thread mit der Nummer `tid` die Werte `smem[tid]` (`mySum`) sowie `smem[tid+64]`. Die übrigen 64 Threads bleiben von da an inaktiv. Da die Threads von den Multiprozessoren in Warps der Größe 32 verwaltet werden, verbrauchen diese inaktiven Threads keine Ressourcen mehr. Sie werden ab jetzt gar nicht mehr verwendet und die Ausführung ihrer Warps terminiert. Die doppelte Zuweisung an `SMem` und `mySum` dient dazu, dass das aktuelle Zwischenergebnis in der jeweils nächsten Addition nicht mehr aus dem gemeinsamen Speicher gelesen werden muss.

Nach der Synchronisation in Zeile 19 bleiben letztendlich 64 zu addierende Partialsummen übrig. Von nun an (Zeile 21) wird auch der zweite Warp dessen Thread-IDs größer als 32 sind nicht mehr benötigt. Da nur noch 32 Threads – also ein Warp – übrig sind, ist für die verbleibenden Additionen keine Synchronisation mehr notwendig, da ein schreibender Zugriff auf den gemeinsamen Speicher sofort für alle Threads eines Warps sichtbar ist.

In den Zeilen 22-27 halbiert sich die Anzahl zu addierender Werte mit jeder Zeile, bis schlussendlich ein einziger Wert übrig ist. Dabei werden offensichtlich mehr Additionen ausgeführt als notwendig sind, da beispielsweise Zeile 23 von allen 32 Threads des Warps ausgeführt wird, obwohl 16 Threads ausreichen würden. Die unnötigen Additionen verändern allerdings nicht die Semantik, so dass die berechnete Summe korrekt bleibt. Ein Ausschließen der überschüssigen Threads ist hier nicht mehr zweckmäßig, da ein Warp immer aus 32 Threads besteht. Es würden also lediglich unnötige arithmetischen Operationen gegen unnötige bedingte Anweisungen getauscht. Die Laufzeit wäre sogar schlechter, da die Bedingung (z.B. `if(tid<16)`) von allen 32 Threads ausgewertet werden müsste.

Abschließend wird die berechnete Partialsumme des Blocks zurück in den globalen Speicher geschrieben. Unabhängig von der Länge n des Eingabearrays sind nun noch die 64 Ergebnisse der Blöcke aufzuaddieren. Harris schlägt in einem Quellcode-Kommentar vor, diese letzten Additionen von der CPU ausführen zu lassen, da sich die Verwendung der GPU für die Verarbeitung so geringer Datenmengen nicht lohne und das Ergebnis der Summation in der Regel ohnehin von der CPU weiterverarbeitet wird.

Diese Implementierung erreicht auf allen CUDA-Hardware-Plattformen einen nahezu optimalen Datendurchsatz. Jeder Wert des Eingabearrays wird nur ein einziges Mal gelesen und die Worker-Threads greifen stets auf nebeneinanderliegende Speicheradressen zu. Darüberhinaus führt jeder Block nur einen Schreibzugriff auf den globalen Speicher aus. Insgesamt beträgt die Anzahl globaler Speicherzugriffe $n + 64$, was in Hinblick auf die hohe Latenz des globalen Speichers einen Großteil der Effizienz dieser Implementierung ausmacht. Da die Anzahl zu summierender Werte in jedem Schritt halbiert wird, ergibt sich eine Schrittkomplexität von $S(n) = \mathcal{O}(\log(n))$. Die Arbeitskomplexität beträgt $W(n) = \mathcal{O}(n)$, da insgesamt n Werte addiert werden.

Die hier gezeigte Reduktion ist als Musterbeispiel für einen inhärent parallelen Algorithmus zu verstehen. Die Aufgaben jedes Threads sind gleichförmig und die Adressen der zu verarbeitenden Daten vollständig aus den Indizes der Threads und Blöcke bestimmbar. Das folgende Kapitel wird zeigen, das dies schon bei unwesentlich komplizierteren Problemstellungen nicht mehr möglich ist. In der Regel sind bereits eine Vielzahl globaler Speicherzugriffe notwendig, um festzustellen, welche Daten überhaupt verarbeitet werden müssen.

4.2.7 Vergleich mit anderen Architekturen

Neben CUDA existieren weitere Frameworks für die parallele Programmierung sowohl für GPUs als auch für Mehrkern CPUs und Rechencluster. Als wichtigste Alternative zu CUDA sei hier die *Open Computing Language* (*OpenCL*) erwähnt. OpenCL ist ein von der Khronos Gruppe verwalteter, offener Standard für die Programmierung paralleler Prozessoren. In OpenCL C geschriebene

Programme sollen laut Spezifikation [26] auf einer Reihe von CPUs, GPUs und *Digitaler Signalprozessoren* (*DSP*) parallel ausführbar sein. Zur Zeit existieren viele OpenCL Implementierungen u.a. von NVidia, Apple, Intel, AMD, IBM, ARM, S3 und VIA. Strukturell sind sich OpenCL und CUDA sehr ähnlich. Alle in den obigen Abschnitten vorgestellten Konzepte existieren ebenfalls in OpenCL, wenn auch die Bezeichner leicht vonaneinder abweichen. Thread-Blöcke heißen beispielsweise *Work-Groups* und Threads heißen *Work-Items*. Auch das Speichermodell stimmt überein. Die Bezeichnungen des globalen sowie Konstantenspeichers sind identisch. Der gemeinsame Speicher heißt *lokaler Speicher* und darf nicht mit dem impliziten lokalen Speicher von CUDA verwechselt werden. Dieser heißt in OpenCL *privater Speicher*. Eine Synchonisation der Work-Items einer Work-Group ist ebenfalls möglich. Damit verfügt OpenCL über die drei zentralen Abstraktionsmechanismen (vgl. Abs. 4.2) von CUDA. Die Unterschiede sind also im wesentlichen syntaktischer Natur. Es existieren Ansätze um CUDA und OpenCL Programme ineinander zu konvertieren [29]. Aufgrund der breiteren Auswahl unterstützter Plattformen wäre OpenCL für die Implementierung von Lernverfahren gut geeignet. Allerdings befanden sich zu Beginn dieser Diplomarbeit die OpenCL Compiler der meisten o.g. Anbieter in einem wesentlich früheren Entwicklungsstadium als CUDA und waren instabiler und/oder schlechter dokumentiert.

Als weitere GPGPU Lösungen seien *DirectCompute* von Microsoft sowie AMD's *FireStream* genannt. Letzteres basiert auf *BrookGPU*, einer von Buck et al. [9] entwickelten Erweiterung der Programmiersprache C. Auf der Internetpräsenz[7] von AMD wird allerdings OpenCL als Standard für die Programmierung ihrer GPUs benannt. DirectCompute ist ein Bestandteil von Microsofts Multimedia Programmierschnittstelle DirectX und daher ausschließlich auf Windows Systemen ausführbar. Wie auch OpenCL ist DirectCompute konzeptionell sehr ähnlich zu CUDA. Die o.g. Abstraktionsmechanismen zur parallelen Programmierung existieren auch dort. Da CUDA nach Meinung des Autors die zur Zeit stabilste und am besten dokumentierte GPGPU Lösung bietet, wurde diese OpenCL und DirectCompute vorgezogen. Sobald die OpenCL Implementierungen ausgereifter sind, sollte eine Portierung der in Kapitel 5 entwickelten CRF Implementierung in Betracht gezogen werden.

Abschließend werden zwei weit verbreitete Frameworks zur parallelen Programmierung von Multikernprozessoren und Rechenclustern vorgestellt. Das *Message Passing Interface* (MPI) [65] ist auf die parallele Berechnung mit Verteilten Systemen ausgelegt. Das Speichermodell von MPI beinhaltet im Unterschied zu den oben genannten GPGPU Frameworks keinen gemeinsamen Speicher. Die Ausnutzung feingranularer Datenparallelität ist somit nicht möglich, da die Daten zur Kommunikation der Rechner über ein Netzwerk versendet werden müssen. In MPI-Programmen führt ein festgelegter Rechner des Rechenclusters den sog. *Wurzelprozess* aus. Dieser liest die Daten ein und verteilt diese zur Berechnung an die übrigen Rechner des Netzwerks. Nach Abschluss der Berechnungen senden diese ihre Ergebnisse zurück zum Wurzelprozess, an dem diese aggregiert werden. *Open Multi-Processing* (OpenMP) [12] ist auf Multikernprozessoren ausgelegt und erweitert die Programmiersprachen C, C++ und Fortran um Anweisungen zur Parallelisierung von Schleifen sowie zur Synchronisation. OpenMP bietet eine Unterstützung für gemeinsamen Speicher und kann zusammen mit MPI in Verteilten Systemen verwendet werden [13]. Die Kombination von MPI und OpenMP entspricht konzeptionell der Aufteilung einer Berechnung in Blöcke und Threads wie sie auch in CUDA, OpenCL und DirectCompute stattfindet. Der Vollständigkeit halber sei hier auch *POSIX Threads (Pthreads)* erwähnt. Pthreads bezeichnet alle Implementierungen des *Institute of Electrical and Electronics Engineers* (IEEE) Standards zur Erzeugung und Verwaltung von Threads [11]. Dieser ist in den meisten POSIX kompatiblen Betriebssystemen wie Linux, Mac OS X, FreeBSD oder Solaris integriert.

[7] AMD und OpenCL:
http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx, Stand: März 2011.

5 CRFs und GPGPU

Dieses Kapitel behandelt die parallele Implementierung von Conditional Random Fields für GPUs auf Basis der Konzepte und Algorithmen der Kapitel 3 und 4. Die Implementierung erfolgte mit CUDA C, wobei die parallelen Algorithmen in diesem Kapitel mit Hilfe eines Pseudocodes präsentiert werden, der ausschließlich Konzepte verwendet die in allen aus Kapitel 4 bekannten GPGPU Architekturen vorhanden sind. Zu Beginn der Bearbeitungszeit dieser Diplomarbeit wurden bestehende CRF Implementierung gesichtet, um festzustellen, ob dort die Integration paralleler CUDA-Algorithmen möglich ist. Es konnte allerdings keine geeignete Implementierung gefunden werden. Die spezifischen Eigenschaften der Implementierungen werden im ersten Abschnitt erläutert. Dabei werden zwei Referenzimplementierungen für die im nächsten Kapitel erläuterten Experimente ausgewählt. Eine die L-BFGS zur Optimierung verwendet, um Güte und Laufzeit der hier implementierten stochastischen Methoden mit denen eines Quasi-Newton-Verfahrens zu vergleichen, sowie eine, die SGD zur Optimierung verwendet. Zwei der im Folgenden untersuchten Implementierungen nutzen eine Parallelisierung zur Beschleunigung des Trainings von CRFs. In Abschnitt 5.2 wird das parallele Design der hier entworfenen Implementierung vorgestellt und mit denen den beiden eben genannten verglichen. Anschließend werden die entwickelten Datenstrukturen (Abs. 5.3) und Algorithmen (Abs. 5.4) vorgestellt und im letzten Abschnitt zu einer parallelen Implementierung von CRFs zusammengefügt.

5.1 Bestehende Implementierungen

CRF Implementierungen lassen sich in dedizierte Linear-Chain CRFs sowie allgemeine Frameworks für PGMs gliedern. Alle Linear-Chain Varianten verwenden den Forward-Backward Algorithmus zur Berechnung der Randverteilungen sowie den Viterbi-Algorithmus zur Klassifikation. Die PGM Frameworks bieten in der Regel mehrere Inferenzalgorithmen an. Es konnten acht Implementierungen ausgemacht werden, von denen zwei als teilparallelisiert zu bezeichnen sind. Keine der bestehenden Implementierung macht Gebrauch von GPGPU.

CRF++ ist eine in C++ geschriebene Linear-Chain CRF Implementierung. Diese wurde in [72] sowie in [23] als CRF Referenzimplementierung für die dortigen Experimente verwendet. CRF++ bietet zur Optimierung L-BFGS sowie den (MIRA) an. Da MIRA keine Maximum-Likelihood Schätzung durchführt, wird dieses Verfahren hier nicht weiter betrachtet. Eine Beschreibung des Algorithmus' ist in [16, 74] zu finden. Dem Programm kann eine Anzahl zu nutzender Threads P als Parameter übergeben werden, wobei keine Publikation ausgemacht werden konnte, in der die verwendete Parallelisierung beschrieben wird. Eine Sichtung des Programmcodes ergab, dass mit Hilfe von Pthreads (s. Abs. 4.2.7) P Instanzen des sequenziellen CRF Trainings bis zur Berechnung des Gradienten parallel ausgeführt werden. Jede Instanz bearbeitet eine Teilmenge der Trainingsdaten. Die einzelnen Gradienten werden abschließend von einem Thread sequenziell aufsummiert. Die Optimierung verläuft sequenziell. Versuche zeigten, dass P als Anzahl vorhandener CPU Kerne gewählt werden sollte, da das Training ansonsten langsamer ist als ohne Parallelisierung. CRF++ wird in Kapitel 6 als Referenzimplementierung für CRFs mit L-BFGS Optimierung verwendet, da für diese veröffentlichte Ergebnisse (s.o.) gefunden werden konnten. Die Laufzeiten von CRF++ sowie die Güte der Lernergebnisse werden mit denen der hier entwickelten Implementierung verglichen. Obwohl der Code von CRF++ gut strukturiert ist, konnte dieser nicht zu einer GPGPU Implementierung erweitert werden. Dort werden keine Datenstrukturen für dünnbesetzte Daten verwendet, wodurch bei einigen der Experimente in Kapitel 6 mehr als 20 GiB Hauptspeicher von CRF++ benötigt wurden. In Anbetracht der Größe des globalen Speichers einer GPU hätten alle Datenstrukturen ausgetauscht werden müssen, was einer Neuimplementierung gleichgekommen wäre. Der Quellcode von CRF++ ist unter folgender URL verfügbar:

`http://crfpp.sourceforge.net`

FlexCRFs/PCRFs zählen zu den in C++ geschriebenen Linear-Chain CRF Implementierungen, wobei PCRFs die parallele Variante von FlexCRFs bezeichnet. Die Parallelisierung wird in [55, 77] beschrieben. Im Wesentlichen stimmt diese mit der von CRF++ überein, wobei PCRFs auf Rechencluster ausgelegt ist und MPI verwendet. Nachdem die Daten von einem *Wurzelprozess* eingelesen wurden, werden diese partitioniert und an die Rechner des Netzwerks versandt. Jeder Einzelne berechnet, wie auch bei CRF++, die Randverteilungen und den Gradienten. Anschließend werden diese an den Rechner, der den Wurzelprozess ausführt, zurückgesandt. Dieser summiert die Einzelnen sog. *lokalen Gradienten* sequenziell auf und führt eine L-BFGS Iteration durch. Anschließend wird der neue Parametervektor an alle Rechner des Netzwerks verteilt und die verteilte Berechnung der lokalen Gradienten beginnt erneut. PCRFs konnte nicht für die Verwendung von GPUPU erweitert werden, da zum einen die MPI-Parallelisierung aufgrund des Speichermodells von Verteilten Systemen nicht ohne weiteres auf GPUs übertragbar ist und zum anderen die verwendeten C++ Datenstrukturen in CUDA C nicht verfügbar sind. Eine Umstellung auf GPGPU wäre also ebenfalls einer Neuimplementierung gleichgekommen. Des Weiteren stellte sich FlexCRF in Versuchen als langsamste Implementierung heraus. Daher fiel die Wahl der Referenzimplementierung auf CRF++. Der Quellcode von FlexCRFs und PCRFs ist unter folgender URL verfügbar:

`http://flexcrfs.sourceforge.net`

CRFSGD ist eine von Leon Bottou in C++ geschriebene Linear-Chain CRF Implementierung die zur Optimierung den stochastischen Gradientenabstieg verwendet. Der Programmcode ist monolithisch aufgebaut und auf eine sequenzielle Verarbeitung der Daten ausgelegt. Die Batchgröße ist konstant $b = 1$ und kann nicht geändert werden. Die Schrittweite wird mit Hilfe einer Linesearch bestimmt und kann nicht eingestellt werden. Von Bottou existieren [8, 7] verschiedene Publikationen über das online Training von CRFs und SVMs sowie das Maschinellem Lernen auf großen Datenmengen, wobei die dortigen Ergebnisse mit CRFSGD erzeugt wurden. Aufgrund der nicht modularen Implementierung sowie dem für GPUs zu hohen Speicherverbrauch von bis zu 6 GiB konnte auch diese Implementierung nicht für die Verwendung von GPGPU angepasst werden. Sie wird aber in Kapitel 6 als Referenzimplementierung für CRFs mit SGD Optimierung verwendet. Der Quellcode von CRFSGD ist unter folgender URL verfügbar:

`http://leon.bottou.org/projects/sgd`

CRFsuite ist die einzige reine C Implementierung von Linear-Chain CRFs die gefunden wurde. Diese kann wahlweise L-BFGS oder eine Mischung aus SGD sowie dem in [63] vorgestellten Verfahren zur Optimierung verwenden. Rückblickend wären zumindest Teile von CRFsuite verwendbar gewesen, allerdings hätte auch dort der Großteil der Datenstrukturen neu implemtiert werden müssen. Da keine Veröffentlichung ausgemacht werden konnte, in der CRFsuite für die Experimente benutzt wird, wurde stattdessen Bottou CRFSGD als Referenzimplementierung für SGD ausgewählt. Einzig das hier verwendete Dateiformat wurde an das von CRFsuite angelehnt. Der Quellcode von CRFsuite ist unter folgender URL verfügbar:

`http://www.chokkan.org/software/crfsuite`

MALLET ist eine in Java geschriebene Programmbibliothek für die Verarbeitung natürlicher Sprache. Zu den Funktionen zählt ebenfalls eine Linear-Chain CRF Implementierung mit L-BFGS Optimierung. Die Erweiterung GRMM (*Graphical Models in MALLET*) ermöglicht die Verwendung von CRFs mit beliebigen Faktorgraphen. Als Inferenzalgorithmen werden Junction Trees, die Belief Propagation sowie das Gibbs Sampling angeboten. Die Implementierung wurde nicht verwendet, da sich die CUDA Schnittstelle zu Java in anfänglichen Versuchen als instabil im Vergleich zur C Schnittstelle erwies. Der Quellcode von MALLET ist unter folgender URL verfügbar:

`http://mallet.cs.umass.edu`

FACTORIE ist eine Programmbibliothek für die Inferenz in beliebigen Faktorgraphen. Diese ist in der zu Java kompatiblem Programmiersprache Scala geschrieben. Primär erfolgt die Inferenz durch ein spezielles MCMC (*Markov Chain Monte Carlo*) Sampling [75]. Auf dem Internetauftritt wird von einer "vorläufigen Unterstützung von Belief-Propagation und Mean-Field Methoden" gesprochen. Als Optimierungsalgorithmen können der Gradientenabstieg sowie L-BFGS verwendet werden. Eine angemessene Einarbeitung in die Programmierung mit Scala sowie in die entsprechende GPGPU Schnittstelle wurde als zu zweitaufwendig eingestuft, um dort die in diesem Kapitel entworfenen Algorithmen zu implementieren. Factorie wurde zwar nicht als Referenz verwendet, jedoch wurde eines der Experimente ebenfalls auf einem HMM ähnlichen Linear-Chain CRF durchgeführt wurde. Factorie ist unter folgender URL verfügbar:

`http://code.google.com/p/factorie`

Infer.NET ist ein Framework von Microsoft Research für die Inferenz in PGMs geschrieben in C#. Als Inferenzalgorithmen werden Belief Propagation und Gibbs Sampling angeboten. Laut Internetauftritt wird ein sog. *Modell Compiler* verwendet, der aus einer Modelldefinition C# Programmcode erzeugt. Dort wird ebenfalls bemerkt, dass Infer-NET zur Zeit auf ungerichteten Modellen nur eingeschränkt nutzbar ist. Da das Framework auf Microsoft Windows ausgelegt ist und auf anderen Betriebssystemen nur eingeschränkt oder gar nicht verwendet werden kann, wurde es hier nicht eingesetzt. Das Framework ist unter folgender URL verfügbar:

`http://research.microsoft.com/en-us/um/cambridge/projects/infernet`

UGM ist ein Framework für ungerichtete Graphische Modelle. Auch hier werden beliebige Faktorgraphen sowie verschiedene Methoden zur Inferenz unterstützt. UGM ist in der kommerziellen Programmiersprache MATLAB[8] geschrieben. Die neuste[9] MATLAB Version besitzt eine GPGPU Schnittstelle, mit der es sowohl möglich ist CUDA C Methoden in MATLAB Code einzubetten, als auch in MATLAB eingebaute, parallelisierte Methoden zu nutzen. Allerdings stand zum einen keine GPU beschleunigte MATLAB Version zur Verfügung und zum anderen hätten alle Algorithmen von UGM, die nicht die vorparallelisierten Methoden nutzen, ohnehin neu implementiert werden müssen. Unter dem Gesichtspunkt, dass GPGPU Frameworks wie CUDA, OpenCL oder DirectCompute frei verfügbar sind, wurde es nicht als sinnvoll erachtet, die einzige kommerzielle Softwarelösung als Basis der Implementierung zu wählen. UGM ist unter folgender URL verfügbar:

`http://www.cs.ubc.ca/~schmidtm/Software/UGM`

Da keine geeignete CRF Implementierung ausgemacht werden konnte, wurde eine eigene entwickelt. In den folgenden Abschnitten werden Vorüberlegungen zum Design der Parallelisierung angestellt sowie die entworfenen Datenstrukturen und Algorithmen vorgestellt.

[8] http://www.mathworks.de/products/matlab
[9] Stand: MATLAB R2010b, Veröffentlicht am 3. September 2010.

5.2 Paralleles CRF Design

In diesem Abschnitt werden einige Vorüberlegungen zur Implementierung in Bezug auf Datenparallelität, Datenlokalität und Berechnungsintensität angestellt und daraus ein paralleles Design abgeleitet. Als Basis wird die Parallelisierung von PCRFs [55, 78] und CRF++ betrachtet. Bei der Betrachtung des Kontrollflusses beider Implementierungen (Abb. 5.1) wird klar, das beide Parallelisierungen im Grunde identisch sind. Die Verteilung der Daten im MPI Cluster entfällt bei CRF++, wobei jeder Thread von CRF++ eine eigene Kopie des Gradienten besitzt, da es bei der Berechnung ansonsten zu Schreibkonflikten unter den Threads kommen würde. Daher müssen die einzelnen lokalen Gradienten abschließend von einem Thread zusammengeführt werden. Bei der Parallelisierung für GPUs ist in dieser Hinsicht zu beachten, dass der globale Speicher der GPU im Vergleich zum Hauptspeicher der CPU klein ist und mehrfache Datenhaltung daher vermieden werden sollte.

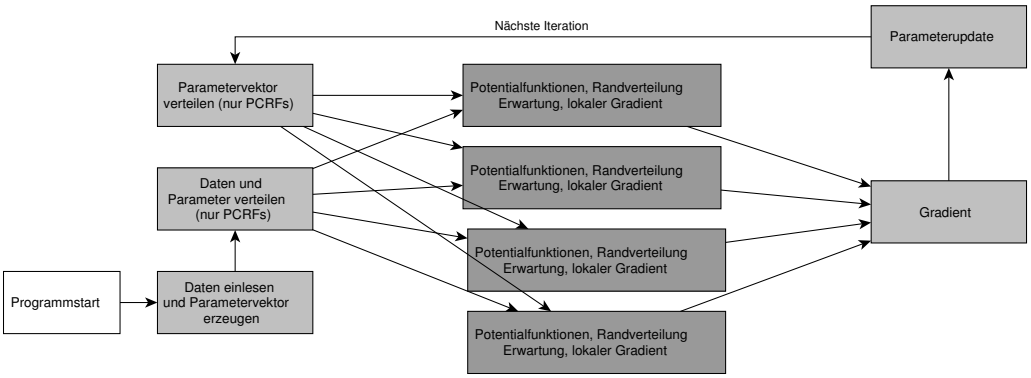


Abbildung 5.1: Schematische Darstellung des Kontrollflusses von PCRFs und CRF++. Die Schritte die nur für PCRFs gelten, können im Fall von CRF++ übersprungen werden.

Als Basis der hier entwickelten GPGPU Implementierung von CRFs wurde das stochastische Training gewählt, da es dem aktuellen Stand der Optimierung von CRFs entspricht. Die in Abschnitt 3.8 vorgestellten Optimierungsalgorithmen sind allesamt iterativ. Algorithmus 7 zeigt den Pseudocode der Hauptschleife eines jeden stochastischen CRF-Trainings. Man beachte, dass die Trainingsmenge $\mathcal{T}$ in dieser Formulierung nicht vorkommt. Wie bereits von Bottou und Vishwanathan [8, 72] bemerkt, spricht also nichts dagegen, dieses Verfahren auf einen potentiell endlosen Datenstrom anzuwenden. In diesem Fall ist eine hohe Datenlokalität gegeben, da ein einmal verarbeiteter Batch prinzipiell aus dem Speicher gelöscht werden kann. Zur Wahrung dieser Datenlokalität erfolgt das Training von CRFs auf GPUs also am besten so, dass dieses bereits beginnen kann, sobald ein Batch eingelesen ist. Im Vergleich dazu beginnt das Training bei den o.g. Implementierungen erst dann, wenn alle Daten eingelesen sind (s. Abb. 5.1). Um mehrere Iterationen auf denselben Daten ausführen zu können, sollte aber trotzdem die Möglichkeit bestehen, einmal eingelesene Daten im Hauptspeicher der CPU zu halten und bei Bedarf in den GPU Speicher zu kopieren.

Aus Abschnitt 4.2.5 ist bekannt, dass die verwendeten Multiprozessoren nur dann effektiv genutzt werden, wenn genug Warps vorhanden sind um die Latenz des globalen Speichers zu verstecken. Das zu lösende Problem muss also eine hohe Datenparallelität besitzen. Infolgedessen wird bei der Parallelisierung versucht, die Datenabhängigkeiten der aus Kapitel 3 bekannten Algorithmen aufzuspüren und alle nicht datenabhängigen Funktionsaufrufe zu parallelisieren. Werden dabei mehr Threads gestartet, als Recheneinheiten vorhanden sind, so greift Brent's Theorem und die "überschüssigen" Threads werden solange auf die verfügbaren MPs verteilt, bis alle bearbeitet wurden.

Da jede Iteration der Optimierung essentiell von den Ergebnissen der jeweils Vorangegangenen abhängt, können die einzelnen Iterationen nicht parallelisiert werden. Das Einlesen der Daten kann

Algorithmus 7 Pseudocode für das stochastische Training von Conditional Random Fields.

```
Eingabe: Faktorgraph G

 1: Lese nächsten Batch B
 2: Erzeuge Parametervektor θ_B
 3: Führe eine Iteration der Optimierung auf l_CRF(θ_B; B) aus
 4: Falls weitere Daten vorhanden sind: Gehe zu 1

Ausgabe: Erwartete optimale Modellparameter θ*
```

Algorithmus 8 Ablauf einer Iteration der Optimierung zur Maximierung der Likelihood.

```
1: Berechne Potentialfunktionen
2: Berechne Randverteilungen
3: Berechne Erwartung des Modells
4: Berechne Gradient
5: Update der Modellparameter
```

ebenfalls nicht parallelisiert werden, da eine Datenquelle wie Festplatte, Netzwerk, o.ä. nur sequenziell gelesen werden kann. Die Parallelisierung muss also innerhalb einer jeden Iteration der Optimierung (Alg. 7, Zeile 3) geschehen. Eine solche Iteration besteht aus den in Alg. 8 dargestellten Schritten. Dort ist an den aufeinanderfolgenden Zeilen eine inhärente Datenabhängigkeitsstruktur abzulesen. Das heißt, dass die Potentialfunktionen nicht parallel zu den Randverteilungen berechnet werden können, da hierzu die Werte der Potentiale bereits vorliegen müssen. Dasselbe gilt für die Erwartung des Modells, zu deren Berechnung die Werte der Randverteilungen erforderlich sind sowie für den Gradienten, der die Modellerwartung benötigt. Das Update der Modellparameter kann schließlich erfolgen, sobald der Gradient bekannt ist. Eine solche Iteration zerfällt also in eine Menge datenabhängiger Teilkomponenten.

In den Parallelisierungen von PCRFs und CRF++ werden die Schritte 1-4 von Alg. 8 für jede Teilmenge der Trainingsdaten von einem Thread bzw. einem Rechner des MPI Clusters berechnet. Dieses Vorgehen ist bei der Verwendung von GPGPU nicht geeignet, da ein einzelner Kern einer GPU nicht die Leistungsfähigkeit eines CPU Kerns besitzt. Wie aus Kapitel 4 bekannt, muss das zu lösende Problem in einzelne Teilprobleme zerlegt werden, um es auf das Thread-Block Konzept abzubilden. Aus diesem Grund wird in der hier entwickelten Parallelisierung für jede der o.g. Teilkomponenten ein eigener GPU Kernel entwickelt. Jeder Kernel ist für mehrere Teilkomponenten zu berechnen. Beispielsweise die Potentialfunktionen für jeden Faktorknoten jedes Beispiels oder die Erwartungen bezüglich jeder bekannten Realisation der versteckten Knoten. Diese werden auf die Blöcke abgebildet. Innerhalb der Blöcke sind wiederum kleinere Schritte auszuführen wie z.B. die Addition der Parameter zur Berechnung der Potentialfunktion. Solche Schritte werden auf die Threads abgebildet.

Auf Basis dieser Vorüberlegungen wurde für die hier entwickelte Implementierung der in Abb. 5.2 dargestellte Kontrollfluss entworfen. Dieser wird hier auch als *paralleles Design* bezeichnet. Die beiden wesentlichen Unterschiede zu den Parallelisierungen von CRF++ und PCRFs bestehen darin, dass die Verarbeitung der Daten in viele datenparallele Teilkomponenten zerlegt wurde und das Training beginnt, bevor die Trainingsdaten vollständig eingelesen wurden.

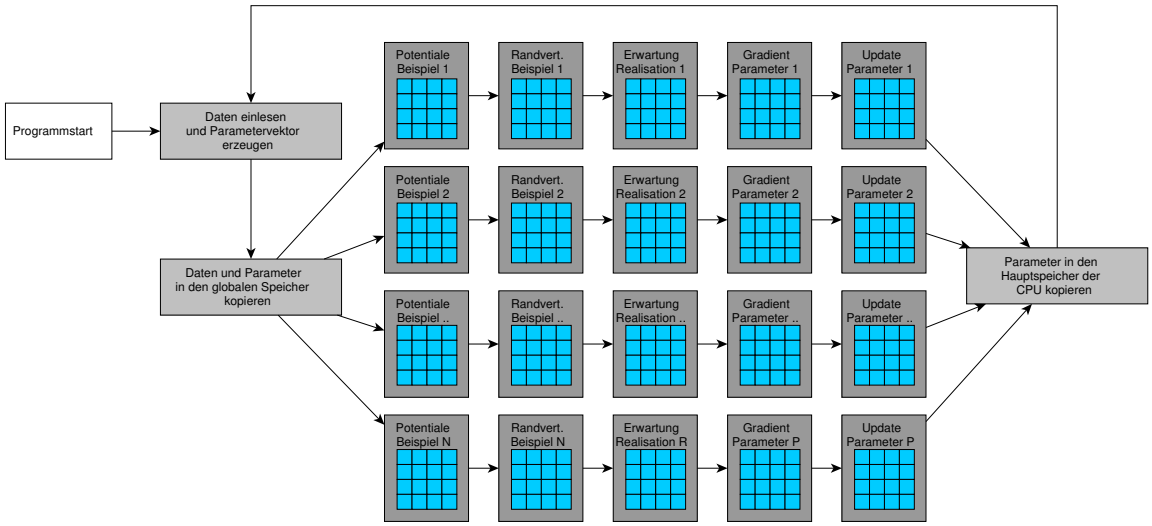


Abbildung 5.2: Schematische Darstellung des Kontrollflusses der hier entwickelten Implementierung. Jede Komponente des Trainingsprozesses wird in Teilprobleme zerlegt und unter Blöcken und Threads aufgeteilt. Lediglich das Einlesen der Daten wird von der CPU durchgeführt, wobei das Training beginnt, sobald der jeweils nächste Batch vollständig eingelesen ist.

In den folgenden Abschnitten werden nun die Datenstrukturen und parallelen Algorithmen vorgestellt und diskutiert, die zur Umsetzung dieses parallelen Designs entwickelt wurden. Abschließend werden diese zu einer parallelen Implementierung von CRFs zusammengefügt.

5.3 Datenstrukturen

Für die Implementierung von CRFs werden Datenstrukturen für Trainings- und Testdaten, Modellparameter sowie für Graphen benötigt. Aus Abschnitt 4.2 ist bekannt, dass parallele Speicherzugriffe der Threads eines Warps zusammengefasst werden können, sofern sie auf die gleiche Bank bzw. das gleiche Speichersegment zugreifen. Dazu muss sichergestellt werden, dass die von den Threads angeforderten Daten im Speicher wenn möglich nebeneinander liegen. Da die in C++ eingebauten dynamischen Datenstrukturen zum einen nicht von CUDA C unterstützt werden und zum anderen aus vielen kleinen Speicherbereichen bestehen, die in der Regel nicht zusammenhängend sind, werden hier ausschließlich einfache Arrays zur Haltung der Daten verwendet. Falls möglich, werden die Daten so in den Arrays abgelegt, dass Threads des gleichen Warps stets auf nebeneinanderliegende Daten zugreifen. Dieses Vorgehen erhöht die effektive Menge tatsächlich parallel verarbeiteter Daten und ermöglicht einen schnellen Datentransfer zwischen CPU und GPU Speicher, da die zu kopierenden Daten aus großen zusammenhängenden Speicherbereichen bestehen.

Da für das Erzeugen der meisten Datenstrukturen Konstanten erforderlich sind, die erst nach dem Einlesen der kompletten Trainingsdaten bekannt sind, müssen dem Programm obere Schranken für diese Werte übergeben werden. Näheres dazu findet sich im letzten Abschnitt dieses Kapitels.

Alle Daten werden initial im globalen Speicher abgelegt, da der gemeinsame Speicher der Multiprozessoren von der CPU weder gelesen noch beschrieben werden kann und nicht über mehrere Kernelaufrufe hinweg persistent ist. Selbst wenn die Berechnungen im gemeinsamen Speicher stattfinden, müssen die Ein- und Ausgabedaten jedes Kernelaufrufs im globalen Speicher liegen.

Im Folgenden wird stets davon ausgegangen, dass alle Mengen beliebig, aber fest numeriert sind. Die Datenstrukturen werden so entworfen, dass jedem hinzugefügten Objekt (Beispiel, Parameter, Faktorknoten, usw.) ein eindeutiger ganzzahliger *Schlüssel* bzw. *Index* zugewiesen wird, über den die Objekte zur Ausführungszeit in den entsprechenden Datenstrukturen adressiert werden.

i	u_i	v_i	i	u_i	v_i
1:	Das	O	12:	Auch	O
2:	Lama	B-TIER	13:	gepunktete	O
3:	glama	I-TIER	14:	Lamas	B-TIER
4:	ist	O	15:	kommen	O
5:	in	O	16:	vor	O
6:	den	O	17:	.	O
7:	südamerikanischen	B-ORT	18:	[Leerzeile]	
8:	Anden	I-ORT	...		
9:	beheimatet	O			
10:	.	O			
11:	[Leerzeile]				

Tabelle 5.1: Format der Eingabedaten von CRFs. Das Beispiel ist an die Named Entity Recognition angelehnt, allerdings frei erfunden.

Alle dichtbesetzten Datenstrukturen wie Faktor- und Variablennachrichten werden als einfaches Array gespeichert. Die Arrayindizes lassen sich dann aus den Indizes der einzelnen Objekte berechnen. Soll beispielsweise die Nachricht des j-ten Faktorknotens des i-ten Beispiels an seinen k-ten Nachbarn über das Label l im Speicher abgelegt werden, so wird diese an die Arrayposition $i\,|F|\,|\Delta_{max}|\,|\mathcal{Y}| + j\,|\Delta_{max}|\,|\mathcal{Y}| + k\,|\mathcal{Y}| + l$ des Arrays (Beispiel: `factor_messages`) geschrieben. Dabei wird der Index k aus der p-adischen Darstellung (s.u.) der versteckten Nachbarschaft des Faktorknotens berechnet.

5.3.1 Label, Beobachtungen und Trainingsmengen

Die Wertebereiche der in CRFs verwendeten Zufallsvariablen sind nominal. Die Trainingsdaten werden in allen frei verfügbaren CRF Implementierungen als Zeichenketten (Strings) repräsentiert (s. Abs. 5.1). Das übliche Dateiformat ist in Tab. 5.1 exemplarisch dargestellt. Jede Zeile enthält die Realisationen eines beobachteten- (linke Spalte) und eines versteckten Knotens (rechte Spalte). Wobei die k-te Zeile eines Beispiels die Realisationen der Knoten $u_k \in X$ sowie $v_k \in Y$ enthält. Einzelne Beispiele sind durch eine Leerzeile getrennt. In Tab. 5.1 wird das aus Abschnitt 2.2 bekannte IOB-Tagging verwendet. Dies bedeutet, dass das *Lama glama* (Tab. 5.1, Zeile 2-3) nur dann korrekt als `TIER` klassifiziert wurde, wenn das Label des Wortes "Lama" als Anfang eines Tiernamens (`B-TIER`), das Label des Wortes "glama" als einem Tiernamen zugehörig (`I-TIER`) und das darauf folgende Wort als nicht zu einem Tiernamen gehörig (jedes Label außer `I-TIER`) klassifiziert wurde. Gleiches gilt für die *südamerikanischen Anden*.

Vorverarbeitung. In Kapitel 3 wurde bereits angemerkt, dass gerade CRFs viele beobachtete Knoten verwenden, was bei den Daten aus Tab. 5.1 nicht der Fall ist. Das heißt, dass die eigentlich verwendeten Realisationen vor Beginn des Trainings durch eine Vorverarbeitung erzeugt werden müssen. Dazu werden Funktionen der gegebenen Realisationen verwendet. Diese können im Fall einer Textsegmentierung zur Berücksichtigung von Kontextinformationen aus einer Konkatenation mehrerer Worte (`‘‘Das/Lama/glama’’`) bestehen. Ebenso kann die Zugehörigkeit einzelner Worte zu Äquivalenzklassen (z.B. `‘‘enthältZiffer`") mittels regulärer Ausdrücke als beobachtete Realisation verwendet werden (s. Abs. 6.3.2). Dabei wird implizit ausgenutzt, dass jede Funktion einer Zufallsvariablen wieder eine Zufallsvariable ist. Zwischen den so erzeugten Zufallsvariablen existieren

```
 i   Label        Beobachtete Knoten gleicher Nachbarschaft

 1: O           u1=Das               u2=/Das
 2: B-TIER      u1=Lama              u2=Das/Lama
 3: I-TIER      u1=glama             u2=Lama/glama
 4: O           u1=ist               u2=glama/ist
 5: O           u1=in                u2=ist/in
 6: O           u1=den               u2=in/den
 7: B-ORT       u1=südamerikanischen u2=den/südamerikanischen
 8: I-ORT       u1=Anden             u2=südamerikanischen/Anden
 9: O           u1=beheimatet        u2=Anden/beheimatet
10: O           u1=.                 u2=beheimatet/.
11: [Leerzeile]
12: O           u1=Auch              u2=/Auch
13: O           u1=gepunktete        u2=Auch/gepunktete
14: B-TIER      u1=Lamas             u2=gepunktete/Lamas
15: ..
```

Tabelle 5.2: Eingabeformat der hier entwickelten Implementierung. Im dargestellten Beispiel hat jeder Faktorknoten dieselbe Anzahl beobachteter Nachbarn. Dies wird vom Eingabeformat allerdings nicht verlangt. Jede Zeile kann beliebig viele (mindestens eine) Beobachtungen enthalten.

per Konstruktion viele Abhängigkeiten. Da aber die Abhängigkeiten zwischen den beobachteten Knoten in CRFs nicht modelliert werden, können auf diese Weise nahezu beliebig viele weitere Zufallsvariablen durch Hinzufügen neuer Knoten erzeugt und zur Klassifikation verwendet werden, ohne dabei etwaige Unabhängigkeitsannahmen treffen zu müssen.

Betrachtet man die beispielhaften Realisationen aus Tab. 5.1, so hat der beobachtete Knoten u_2 die Realisation "`Lama`" und u_3 die Realisation "`glama`". Durch Konkatenation der beiden ergibt sich eine neue Zufallsvariable mit der Realisation "`Lama/glama`". Auf diese Weise wird dem korrespondierenden Faktorgraphen ein neuer beobachteter Knoten hinzugefügt. Diese werden auch als *Merkmal* bezeichnet. Wiederholt man diese Konkatenation für jeden beobachteten Knoten, so ergibt sich der in Abb. 5.3 (links) dargestellte Faktorgraph.

Das in Tab. 5.1 dargestellte Dateiformat impliziert, dass die Vorverarbeitung der Daten von der jeweiligen Implementierung ausgeführt wird. Der Autor wollte dies vermeiden, um die Vorverarbeitung der Daten mit anderen Programmen wie *RapidMiner* [44] zu erlauben. Die Vorverarbeitung der Daten für die Experimente in Kapitel 6 wurden beispielsweise mit Python-Skripten durchgeführt, welches die Daten aus Tabelle 5.1 mit Hilfe einfacher Zeichenkettenmanipulationen in das hier verwendete Datenformat (Tab. 5.2) überführt. Diese Skripte werden hier nicht weiter erläutert, da dazu eine Einführung in die Programmiersprache Python notwendig wäre. Darüberhinaus sind diese Skripte kein fester Bestandteil der hier entwickelten Implementierung, da die Vorverarbeitung mit jedem beliebigen Programm erfolgen kann.

Wie in Abschnitt 3 erwähnt, können zur vereinfachten Darstellung von Faktorgraphen alle beobachteten Knoten mit identischer Nachbarschaft zu einem aggregiert werden, so dass beide in Abb. 5.3 dargestellten Graphen äquivalent sind. Diese Aggregation motiviert das in Tab. 5.2 dargestellte Eingabeformat der hier entwickelten Implementierung. Dort enthält die jeweils i-te Zeile in der linken Spalte das i-te beobachtete Label, gefolgt von den Realisationen des i-ten

$\mathcal{X}$ (Beobachtungen)	Id	$\mathcal{X}$ (Beobachtung)	Id	$\mathcal{Y}$ (Label)	Id
‘‘u1=Das’’	0	‘‘u1=den’’	10	‘‘O’’	0
‘‘u2=Das/Lama’’	1	‘‘u2=den/südamerikanischen’’	11	‘‘B-TIER’’	1
‘‘u1=Lama’’	2	‘‘u1=südamerikanischen’’	12	‘‘I-TIER’’	2
‘‘u2=Lama/glama’’	3	‘‘u2=südamerikanischen/Anden’’	13	‘‘B-ORT’’	3
‘‘u1=glama’’	4	‘‘u1=Anden’’	14	‘‘I-ORT’’	4
‘‘u2=glama/ist’’	5	‘‘u2=Anden/beheimatet’’	15		
‘‘u1=ist’’	6	‘‘u1=beheimatet’’	16		
‘‘u2=ist/in’’	7	‘‘u2=beheimatet/.’’	17		
‘‘u1=in’’	8	‘‘u1=.’’	18		
‘‘u2=in/den’’	9	‘‘u2=./’’	19		

Tabelle 5.3: Darstellung der Mengen $\mathcal{X}$ und $\mathcal{Y}$ bei einer Textsegmentierung. Als beobachtete Knoten wurden die Worte des Satzes, sowie die Paare von aktuellem und nachfolgendem Wort gewählt.

Aggregats[10]. Der Präfix einer beobachteten Realisation kennzeichnet seinen Wertebereich. Die Reihenfolge sowie die Anzahl der Realisationen ist beliebig. Das heißt, dass theoretisch jedes beobachtete Aggregat andere Variablen mit eigenen Wertebereichen enthalten kann. Praktisch wird meist dasselbe Preprocessing auf alle Knoten des Graphen angewendet, so dass sich einzelne Beobachtungen denselben Wertebereich teilen (s. Tab. 5.2). Dieses Format ist an das von CRFsuite (s. Abs. 5.1) angelehnt, wobei die Präfixe dort fest vorgegeben sind und Informationen zur Adjazenz der Knoten beinhalten. In der hier entwickelten Implementierung sind die Präfixe dagegen frei wählbar, da das Dateiformat keinerlei Adjazenzinformationen bezüglich versteckter und beobachteter Knoten enthält. Die graphische Struktur ist also von den Eingabedaten abgekoppelt und wird separat repräsentiert (s.u.).

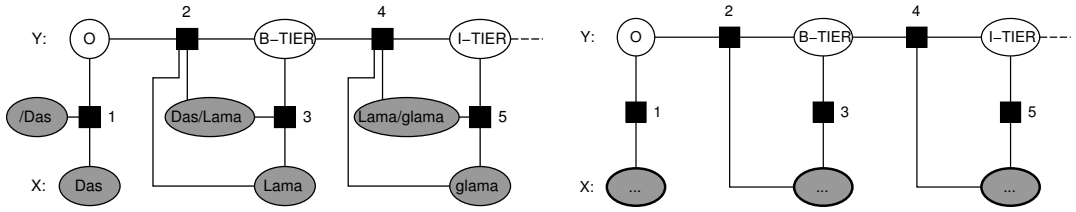


Abbildung 5.3: Zwei Faktorgraphen. Die Faktorknoten sind fortlaufend nummeriert. Alle beobachteten Knoten des linken Graphen mit gleicher Nachbarschaft wurden im rechten Graphen zu einem Knoten aggregiert.

Indizierung der Daten. Die eigentlichen Trainingsdaten bestehen also aus den Labeln der versteckten Knoten $v \in Y$ sowie den Realisationen der, durch die Vorverarbeitung erzeugten, aggregierten, beobachteten Knoten. Wird nun das erste Beispiel aus Tab. 5.1 mit der oben beschriebenen, exemplarischen Vorverarbeitung zum Training eingelesen, werden die in Tab. 5.3 dargestellten Mengen $\mathcal{X}$ und $\mathcal{Y}$ angelegt. Jeder eingefügten Realisation wird eine eindeutige ganzzahlige Identifikationsnummer (Id) zugewiesen. Auf diese Weise ist es möglich, die Mengen $\mathcal{X}$ und $\mathcal{Y}$ als Mengen ganzer Zahlen (den Identifikationsnummern) zu betrachten. So können die Beispiele als Folgen (Arrays) von Ids gespeichert werden. Dies ist in Tabelle 5.4 dargestellt. Um innerhalb der Implementierung effizient auf die Realisationen der einzelnen beobachteten Zufallsvariablen zugreifen zu können, wird für jedes Beispiel ein Hilfsarray $\mathbf{m}$ erzeugt, so dass $\mathbf{m}[k]$ auf den Index

[10] Existieren mehr Aggregate als versteckte Knoten, so kann den entsprechenden Zeilen das Label `NIL` zugewiesen werden. Dieses Label ist reserviert und signalisiert, dass die betroffene Zeile keine Informationen bezüglich versteckter Knoten enthält.

x	Das	Lama	glama	ist	in	den	süda.	Anden	beheimatet	.
Aggregat	0,1	2,3	4,5	6,7	8,9	10,11	12,13	14,15	16,17	18,19
m	0	2	4	6	8	10	12	14	16	18
y	O	B-TIER	I-TIER	O	O	O	B-ORT	I-ORT	O	O
	0	1	2	0	0	0	3	4	0	0

Tabelle 5.4: Repräsentation einer Beobachtung. **m** ist ein Hilfsarray, in dem die Startindizes der jeweiligen Aggregate gespeichert werden.

der ersten Realisationen des k-ten Aggregats verweist. Das Label des k-ten versteckten Knotens kann mit Hilfe des Arrays $\mathbf{y}$ an Position $\mathbf{y}[k]$ abgefragt werden.

Um nun die Trainingsmenge bzw. einen Batch darzustellen, werden die Arrays $\mathbf{x}^{(i)}$, $\mathbf{y}^{(i)}$ aller Beispiele hintereinanderkopiert, so dass die Felder $\mathbf{X} = \mathbf{x}^{(1)}\mathbf{x}^{(2)}\ldots$, $\mathbf{Y} = \mathbf{y}^{(1)}\mathbf{y}^{(2)}\ldots$ und $\mathbf{M} = \mathbf{m}^{(1)}\mathbf{m}^{(2)}\ldots$ entstehen. Bei der Konkatenation werden die Indizes in $\mathbf{M}$ so verschoben, das damit die Arrays $\mathbf{X}$ und $\mathbf{Y}$ indiziert werden können. Zusätzlich werden die Startindizes der einzelnen Beispiele im Array $\mathbf{S}$ abgelegt, so dass letztendlich ein wahlfreier Zugriff auf alle Realisationen, aller Knoten, aller Beispiele in $\mathcal{O}(1)$ möglich ist. Der Ausdruck $\mathbf{X}[\mathbf{M}[\mathbf{S}[i]+k]+j]$ liefert die Realisation des j-ten beobachteten Knotens des k-ten Aggregats des i-ten Beispiels zurück. Das Label des k-ten versteckten Knotens des i-ten Beispiels ist dementsprechend $\mathbf{Y}[\mathbf{S}[i]+k]$. Da jedes Beispiel aus einer unterschiedlichen Anzahl von Knoten bestehen kann (s. Abs. 3.5), wird ebenfalls die Anzahl der aktiven versteckten Knoten eines jeden Beispiels in einem Array gespeichert.

5.3.2 Versteckte Nachbarn und deren Realisationen

Die formale Darstellung von CRFs macht häufig den Gebrauch von Teilmengen der Knotenmenge V (z.B.: versteckte Nachbarschaft $\Delta(f) \subseteq V$) oder Realisationen von Knotenmengen wie z.B. $\mathbf{y}_{\Delta(f)}$. Auf die Darstellung mit Hilfe dynamischer Datenstrukturen wie Bäumen o.ä. wurde verzichtet, da die Speicherbereiche dynamischer Datenstrukturen in der Regel nicht zusammenhängend sondern durch Zeiger verbunden sind, wodurch in jeder Iteration sehr viele kleine Speicherabschnitte vom Hauptspeicher der CPU in den globalen Speicher der GPU kopiert werden müssten. Stattdessen werden alle Nachbarschaften oder sonstigen Knotenmengen, sowie gemeinsame Realisationen von Zufallsvariablen mit Hilfe der p-adische Darstellung kodiert. Mit dieser ist es möglich, eine Menge von ganzen Zahlen als eine ganze Zahl darzustellen. Dazu wird eine Menge absteigend sortiert[11] und als ganze Zahl zur Basis p interpretiert, wobei p eine Primzahl sowie eine obere Schranke für das größtmögliche Element der zu kodierenden Menge ist. Hat ein Abhängigkeits- bzw. Faktorgraph n versteckte Knoten, so sei $p_n := \texttt{nextPrime}(n)$ die nächst größere Primzahl. Theoretisch ist die Laufzeit von `nextPrime` polynomiell [18], praktisch wird dazu einmalig bei Initialisierung des Modells ein fest vorgegebenes Array der ersten 10^5 Primzahlen durchsucht. Dies beschränkt zwar die maximale Anzahl der Knoten des CRFs auf $\approx 10^5$, allerdings ist die Knotenzahl in praktischen Anwendungen wesentlich kleiner ($\approx 10^2$).

$$\text{encode}(\Delta(f)) = \sum_{i=1}^{|\Delta(f)|} \Delta_i(f)\, p_n^{i-1} = 65 \cdot p_n^0 + 12 \cdot p_n^1 + 3 \cdot p_n^2 = 21728 \tag{5.1}$$

$$\text{encode}\left(\tilde{\Delta}(f)\right) = \sum_{i=1}^{|\tilde{\Delta}(f)|} \tilde{\Delta}_i(f)\, p_n^{i-1} = 4 \cdot p_n^0 + 2 \cdot p_n^1 = 170 \tag{5.2}$$

[11] Da Mengen keine Ordnung besitzen, können diese beliebig sortiert werden, ohne die Semantik der Menge zu verändern.

Die obige Kodierung wird nun exemplarisch verwendet, um einen Faktorknoten $f \in F$ mit den versteckten Nachbarn $\Delta(f) = \{3, 65, 12\}$ sowie den beobachteten Aggregaten $\tilde{\Delta}(f) = \{2, 4\}$ darzustellen. Sei $n = 80$ sowie $p_n = 83$ die nächste Primzahl. Die Nachbarn werden absteigend sortiert $\Delta(f) = (65, 12, 3)$, $\tilde{\Delta}(f) = (4, 2)$ und wie in den Gleichungen (5.1, 5.2) als Zahl zur Basis p_n interpretiert, wobei Δ_i das i-te Element der sortierten Nachbarschaft bezeichnet. Diese Kodierung wird bereits beim Einlesen der Daten von der CPU vorgenommen und die Werte (hier: 21728 und 170) werden anschließend als 64 Bit `unsigned long` gespeichert. Haben zwei Faktorknoten dieselbe Nachbarschaft, so sind diese identisch. Ein Faktorknoten kann also durch eine p-adische Kodierung seiner Nachbarschaften eindeutig mit zwei natürlichen Zahlen dargestellt werden. Diese werden zu einem sog. *Faktor-Deskriptor* zusammengefasst und in der Implementierung benutzt, um Faktorknoten eindeutig zu repräsentieren. Eine alternative Darstellung mit Hilfe kleiner Arrays würde mindestens $|\Delta(f)| + |\tilde{\Delta}(f)|$ ganze Zahlen benötigen.

Da p_n eine Primzahl ist, kann die Kodierung durch Division und Modulo-Rechnung leicht invertiert werden. Der entsprechende CUDA Code ist in Alg. 9 zu sehen. Hierbei ist `a` ein Array im gemeinsamen Speicher, in welches die dekodierten Knotenindizes geschrieben werden. Das Argument `k` ist die Größe der zu dekodierenden Menge, wobei hier stets `k`$= \max\left(\Delta_{max}, \tilde{\Delta}_{max}\right)$ gewählt wird. Das Argument `p` ist die verwendete Primzahl sowie `m` der zu dekodierende Wert. Die Laufzeit beträgt offensichtlich $\mathcal{O}(k)$ wobei nur ein einziger globaler Speicherzugriff notwendig ist. Da die Latenz globaler Speicherzugriffe relativ hoch ist (vgl. Abs. 4.2.4), ist die p-adische Kodierung eine effiziente Methode zur Darstellung von Knotenmengen in CUDA-Programmen sowie allgemeinen GPGPU Programmen. Des Weiteren erhöht dieses Vorgehen die in Kapitel 4 erwähnte Berechnungsintensität, da hierbei $\mathcal{O}(k)$ globale Speicherzugriffe gegen $\mathcal{O}(k)$ Modulo- und Divisionsrechnungen "getauscht" werden. Hierbei sei darauf hingewiesen, dass das Array `a` im gemeinsamen Speicher liegt, dessen Latenz mit der eines Registerzugriffs vergleichbar ist (s. Abs. 4.2.4).

An dieser Stelle muss allerdings erwähnt werden, dass diese Kodierung die Größe der größten repräsentierbaren Nachbarschaft stark einschränkt, da der größte repräsentierbare Code 2^{64} ist. Die Darstellung einer Menge von 4 Knoten ist beispielsweise für $n = 8192$ nicht mehr uneingeschränkt möglich, da Mengen mit "großen" Knotennummern, wie z.B. $\{8192, 8191, 8190, 8189\}$, nicht mit 64 Bit Datentypen darstellbar sind. Dies stellt einen Trade-Off zwischen der Anzahl der versteckten Knoten eines Faktorgraphen und der Größe der größten Nachbarschaft eines Faktorknotens dar. Da die Knotenanzahl von CRFs – wie oben erwähnt – im Bereich von $\approx 10^2$ liegt, stellt diese Einschränkung kein Problem in der praktischen Anwendung dar. Sollen tatsächlich große Nachbarschaften ($\Delta_{max} > 10$) in großen Graphen ($n > 10^4$) verarbeitet werden, so ist es möglich, die Knoten einer Nachbarschaft aufzuteilen und mehrere Codes zu erzeugen. Dies wurde allerdings nicht implementiert. Die im Rahmen dieser Diplomarbeit entstandene Implementierung entscheidet beim Einlesen der Daten, ob das gewünschte Modell mit den verwendeten Datenstrukturen darstellbar ist und terminiert mit einer Fehlermeldung, falls dies nicht der Fall ist.

Da $\mathcal{Y}$ ebenfalls ganze Zahlen enthält (s.o.), können die Realisationen der Nachbarschaften auf dieselbe Weise als Zahl zur Basis `nextPrime(`$|\mathcal{Y}|$`)` kodiert werden. Da CRFs nur auf die Realisationen einzelner beobachteter Knoten konditioniert sind, ist die Kodierung gemeinsamer Realisationen beobachteter Knoten nicht notwendig, da diese schlichtweg nicht vorkommen. Zur Repräsentation einer einzelnen Beobachtung kann daher ihre ganzzahlige Identifikationsnummer (s. Tab. 5.3) verwendet werden.

Die in den Trainingsdaten enthaltenen Relisationen von Knotenmengen werden beim erstmaligen Einlesen ebenfalls fortlaufend nummeriert. Dabei ist es egal, an welcher Knotenmenge eine Realisation beobachtet wurde. Die im Folgenden verwendete Nummerierung der Faktoknoten entspricht derjenigen aus Abb. 5.3. In den Daten aus Tab. 5.2 wird am ersten Faktorknoten des ersten Beispiels die Realisation "`O`" eingelesen. Diese erhält daher die Nummer 1. Als nächstes wird am zweiten Faktorknoten die Realisation "`O,B-TIER`" eingelesen. Diese erhält die Nummer 2. Wird

Algorithmus 9 CUDA Code für die Dekodierung p-adischer Zahlen.

```
1: __device__ void baseDecode(volatile unsigned int *a, int k,
2:                            unsigned int p, unsigned long m){
3:   unsigned long c = m;
4:   for(unsigned int i=0; i<k; i++){
5:        a[i] = c%b;
6:        c    = c/b;
7:   }
8: }
```

die Realisation dann am vierten Faktorknoten des zweiten Beispiels erneut gelesen bekommt sie keine neue Nummer, sondern erhält erneut die Nummer 2. Um später darauf zurückgreifen zu können, wird ein `unsigned long` Array verwendet, dass an Position i die p-adische Kodierung der i-ten beobachteten Realisation enthält. Des Weiteren wird die Anzahl insgesamt beobachteter Realisation R für die Allokation verschiedener Arrays, beispielsweise für die berechneten Potentialfunktionen, verwendet.

5.3.3 Parametervektor

Nun wird die Darstellung von Parametern erläutert. Aus Kapitel 3 ist bekannt, dass die Parameter zur Reduktion der Platzkomplexität lediglich für diejenigen Realisationen erzeugt werden, die in den Trainingsdaten vorkommen. Dadruch, dass jeder durch einen Batch induzierte Parametervektor nur eine Auswahl aller bekannten Gewichte enthält, wird die Platzkomplexität weiter reduziert. Beim Einlesen eines Batches wird dazu eine Liste der Parameter erzeugt, die in den globalen Speicher der GPU kopiert werden müssen. Zur eindeutigen Identifikation der Parameter wurde ein sog. *Parameter-Deskriptor* auf Basis der p-adischen Kodierung entwickelt. Jeder Parameter $\boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}$ mit $v \in \tilde{\Delta}(f)$ wird durch die p-adische Kodierung der Nachbarschaft $\Delta(f)$, die p-adische Kodierung der Realisation $\mathbf{y}_{\Delta(f)}$ sowie der Id der beobachteten Realisation $\mathbf{x}_v$ repräsentiert. Da beim Einlesen der Daten mit Hilfe der Präfixe sichergestellt wird, dass zwei unterschiedliche beobachtete Knoten niemals dieselbe Realisation besitzen können (s. Abs. 5.3.1), ist diese Darstellung eindeutig. Zusätzlich enthält ein Parameter die Template-Id h, falls er ein Parameter des h-ten Faktor-Templates $\mathcal{F}_h$ ist. Zur schnellen Indizierung der anderen Datenstrukturen wird ebenfalls die fortlaufende Nummer der Realisation $\mathbf{y}_{\Delta(f)}$ (s. Abs. 5.3.2) sowie die Anzahl der versteckten Knoten $|\Delta(f)|$ gespeichert.

Auf der CPU Seite wird der Parametervektor des Modells $\boldsymbol{\theta}$ als assoziatives Array dargestellt, welches in der Lage ist Parameter-Deskriptoren in Laufzeit $\mathcal{O}(\log(n))$ auf den entsprechenden Gleitkommawert $\boldsymbol{\theta}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}$ abzubilden. Für die Darstellung des Parametervektors $\boldsymbol{\theta}_{\mathcal{B}}$ auf der GPU wird die oben erwähnte Liste nach der Id der Realisation $\mathbf{x}_v$ sortiert und die entsprechenden Parameter aus dem assoziativen Array in ein einfaches Gleitkomma Array $\boldsymbol{\theta}_{\mathcal{B}}$ kopiert. Dazu wird ein zweites gleichlanges Array angelegt, in welches die entsprechenden Parameter-Deskriptoren kopiert werden. Benötigt nun ein GPU Thread die Information, welche Realisation durch den i-ten Parameter gewichtet wird, so wird die i-te Stelle des Parameter-Deskriptor-Arrays ausgelesen. Die Parameter werden mit $0,05$ initialisiert, da dies in [67] vorgeschlagen wurde. Dieser Wert ist allerdings einstellbar.

Zur Berechnung der Potentiale einer Realisation $\mathbf{y}_{\Delta(f)}, \mathbf{x}_{\tilde{\Delta}(f)}$ werden alle Gewichte der unter $\mathbf{x}_v$, $v \in \tilde{\Delta}(v)$ bekannten Realisationen benötigt. Dazu wird ein Array der Länge $|\mathcal{X}|$ angelegt. Für jede mögliche Realisation $\mathbf{x}_v$ eines jeden beobachteten Knotens v wird in diesem Array vermerkt,

wo die entsprechenden Parameter im Array $\boldsymbol{\theta}_{\mathcal{B}}$ beginnen und enden. Dies motiviert die Sortierung von $\boldsymbol{\theta}_{\mathcal{B}}$ entsprechend der Id der Realisationen, da so alle Parameter für eine Realisation $\mathbf{x}_v$ im Speicher nebeneinander liegen.

Während der jeweils aktuelle Parametervektor erzeugt wird, werden die Häufigkeiten der Realisationen $\mathbf{y}_{\Delta(f)}, \mathbf{x}_v$ gezählt, da diese zur Berechnung des Gradienten benötigt werden. Zusätzlich wird für jede, der im Batch vorkommenden Realisationen von beobachteten Knoten in einer Liste vermerkt, in welchen Beispielen diese vorkommen.

Im Folgenden wird beschrieben, welche Datenstrukturen zu Darstellung der Faktorgraphen verwendet werden.

5.3.4 Faktorgraphen

Faktorgraphen werden in der Implementierung durch die Nachbarschaften der Knoten beschrieben. Dazu werden alle Knoten fortlaufend durchnummeriert. Die Menge der Faktorknoten F wird durch ein Array der entsprechenden Faktor-Deskriptoren (s.o.) repräsentiert, so dass das Array an Position i den Deskriptor des i-ten Faktorknotens enthält. Für den schnellen Zugriff auf die Variablennachrichten, werden zusätzlich die fortlaufende Nummer eines Faktorknoten bezüglich der Nachbarschaften seiner versteckten Nachbarn gespeichert. Ist also der versteckte Knoten v in der Nachbarschaft des Faktors f enthalten, so ist im Deskriptor von f gespeichert, der wievielte Nachbar f von v ist. Auf diese Weise kann ein Variablenknoten die ausgehenden Nachrichten an seine Nachbarn in einer festen Reihenfolge im Speicher abgelegen. Die Faktorknoten können somit zur Berechnung ihrer Faktorknachrichten direkt auf die eingehende Nachricht zugreifen. Für jeden versteckten Knoten werden lediglich die Nummern der Faktorknoten gespeichert, zu denen dieser adjazent ist. Da die beobachteten Nachbarschaften der Faktorknoten in den Faktordeskriptoren gespeichert sind, ist damit der Graph vollständig beschrieben.

Die explizite Repräsentation der Faktorgraphen ist bei der Verwendung des Forward-Backward Algorithmus nicht notwendig, da der Graph dort in den Algorithmus eincodiert ist. Bei den allgemeinen Varianten der Belief-Propagation wird der Graph im Konstanten-Speicher der GPU abgelegt, da unabhängig von der Batchgröße nur eine Instanz des Graphen erforderlich ist.

In diesem Abschnitt wurden alle wesentlichen Datenstrukturen der Implementierung vorgestellt. Die Laufzeit eines Zugriffs auf eine der Datenstrukturen beträgt $\mathcal{O}(\Delta_{max})$ falls dazu eine p-adische Id dekodiert werden muss und ansonsten $\mathcal{O}(1)$.

5.4 Algorithmen

In diesem Abschnitt werden die Algorithmen aus Kapitel 3 bezüglich eventueller Datenabhängigkeiten analysiert und anschließend parallele Algorithmen hergeleitet. Die Algorithmen werden hier in Pseudocode dargestellt. Falls nicht anders angegeben, liegen die verwendeten Datenstrukturen im globalen Speicher. Alle numerischen Variablen und Arrays seien mit 0 initialisiert. Parallel ausgeführte Instruktionen werden mit einem `forall`-Statement kenntlich gemacht. Der Ausruck

`forall(Block,`W_F`) Faktorknoten` $j \in F$`{ .. }`

entspricht einer Schleife über alle Faktornoten, wobei die einzelnen Iterationen der Schleife unabhängig voneinader sind und parallel von W_F Blöcken ausgeführt werden. Teilweise wird auf die Angabe einer Grundmenge (hier: F) verzichtet. Beispielsweise beschreibt der Ausdruck

`forall(Thread,` W_Y`) unter` x `bekannte Realisation` $\mathbf{y}$`{ .. },`

eine Schleife über alle unter x bekannten Realisationen $\mathbf{y}$ (vgl. Abs. 3.2), wobei die einzelnen Iterationen der Schleife parallel von W_Y Threads ausgeführt werden. Diese Notation ist an die in [34] verwendete angelehnt. Dort kennzeichnet ein `forall`-Statement eine Schleife deren Iterationen von "allen verfügbaren" Prozessoren ausgeführt werden soll. Hier wird zusätzlich die Anzahl der verwendeten Blöcke und Threads (hier W_F bzw. W_Y) angegeben sowie, ob es sich um eine parallele Ausführungen von Blöcken oder Threads handelt. Diese Anzahlen können als freier Parameter des Algorithmus verstanden werden, von denen die meisten in der CUDA-Implementierung über einen Kommandozeilenparameter einstellbar sind. Dies wird im letzten Abschnitt dieses Kapitels näher erläutert.

Da ausschließlich Konzepte verwendet werden, die auf allen aus Abschnitt 4.2.7 bekannten GPGPU Architekturen zur Verfügung stehen, sollten sich die im Folgenden vorgestellten Algorithmen leicht auf jede davon portieren lassen. Der vollständige CUDA Code der hier gezeigten Algorithmen ist auf der CD-ROM in Anhang D zu finden.

Obwohl in den hier gezeigten Algorithmen häufig Summationen oder Maximumsbildungen, also assoziative Funktionen, benötigt werden, wird dazu selten die Reduktion aus Kapitel 4 eingesetzt. Experimente während der Entwicklung zeigten zwar, dass sich diese bei der Verarbeitung großer Datenmengen ($\approx 10^8$ Werte) lohnt, in den Algorithmen jedoch häufig viele ($\approx 10^6$) kleine Summen mit wenigen ($< 10^2$) Summanden zu berechnen sind. In diesem Fall ist es vollkommen ausreichend, mehrere sequenzielle Summationen parallel durchzuführen, anstatt dazu $\approx 10^6$ Reduktionen zu verwenden.

Bevor nun die Algorithmen vorgestellt werden, wird noch ein technisches Detail erläutert, dass bei jeder CRF Implementierung zu beachten ist.

Logarithmierte Berechnungen. Sutton und McCallum [67] bemerkten, dass bereits kleine Instabilitäten in den Berechnungen zu ungenauen Ergebnissen und im schlimmsten Fall zur Divergenz der Verfahren führen können. Sie erklärten weiterhin, dass dieses Problem weitestgehend behoben werden könne, indem ausschließlich mit Logarithmen gerechnet werde. Im Fall von Multiplikationen stellt dies offensichtlich kein Problem dar, denn wenn zwei reelle Zahlen $a, b \in \mathbb{R}$ multipliziert werden sollen, gilt offensichtlich $a \cdot b = \exp(\ln(a) + \ln(b))$. Das heißt alle Multiplikationen innerhalb einer Berechnung können als $\ln(a) + \ln(b)$ dargestellt werden und erst das endgültige Ergebnis muss mit Hilfe der Exponentialfunktion in den Ursprungsraum überführt werden. Ist allerdings eine Summe $\sum_i a_i$ zu berechnen, so ist dies nicht ohne Weiteres möglich, falls anstelle der a_i lediglich die $\ln(a_i)$ vorliegen. Im Grunde bleibt nur die Umkehrung des Logarithmus übrig, wobei Sutton und McCallum [67] dazu die logarithmische Addition $\bigoplus$ vorschlugen (s. Gleichung 5.3).

$$\bigoplus(a, b) := \begin{cases} a + \ln(1 + \exp(b - a)) & ,\text{falls } b \leq a \\ b + \ln(1 + \exp(a - b)) & ,\text{sonst} \end{cases} \tag{5.3}$$

Dabei wird der Exponent zur Erhaltung der numerischen Stabilität stets so klein wie möglich gewählt und es gilt $\exp(\oplus_i \ln(a_i)) = \sum_i a_i$. Soll also beispielsweise die Forward-Nachricht (5.4) des FB Algorithmus berechnet werden, so wird stattdessen die Log-Forward-Nachricht (5.5) berechnet.

$$\alpha_t(y \mid \mathbf{x}) = \sum_{y' \in \mathcal{Y}} f_t(y \mid \mathbf{x}_t) \cdot f_{\{t-1,t\}}\left(y', y \mid \mathbf{x}_t\right) \cdot \alpha_{t-1}\left(y' \mid \mathbf{x}\right) \tag{5.4}$$

$$\ln \alpha_t(y \mid \mathbf{x}) = \bigoplus_{y' \in \mathcal{Y}} \ln f_t(y \mid \mathbf{x}_t) + \ln f_{\{t-1,t\}}\left(y', y \mid \mathbf{x}_t\right) + \ln \alpha_{t-1}\left(y' \mid \mathbf{x}\right) \tag{5.5}$$

5.4.1 Potentialfunktionen

Die zentrale Komponente der Dichtefunktion von CRFs sind die Potentialfunktionen bzw. die lokalen Funktionen des Faktorgraphen. Im Folgenden werden Komplexität und Datenparallelität der Potentialfunktionen analysiert und daraus ein paralleler Algorithmus hergeleitet. Die lokalen Funktionen müssen zur Berechnung der Randverteilung

- für jedes Beispiel im aktuellen Batch $\mathcal{B}$, also für alle $i \in \{1, \ldots, |\mathcal{B}|\}$,
- für alle aktiven Faktorknoten $j \in F$ eines jeden Beispiels,
- sowie für alle bekannten Realisationen $\mathbf{y}_{\Delta(j)}$ der versteckten Nachbarn von j

ausgewertet werden (s. Abs. 3.6). Ist eine mögliche Realisation der versteckten Knoten $\mathbf{y}'_{\Delta(j)}$ bei gegebener Realisation x eines observablen Knotens nicht in den Trainingsdaten enthalten, so existiert für diese Kombination auch kein Gewicht, welches innerhalb der Potentialfunktion aufaddiert werden könnte. Sowohl die Anzahl aktiver Faktorknoten eines Beispiels als auch die Anzahl der zu einer Beobachtung x bekannten Realisationen sind a-priori unbekannt. Daher werden diese mit $\mathcal{O}(|F|)$ bzw. $\mathcal{O}(|\mathcal{Y}|^{\Delta_{max}})$ nach oben abgeschätzt, wobei Δ_{max} die Größe der größten Nachbarschaft eines Faktorknotens bezeichnet. Damit ergibt sich $\mathcal{O}(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}})$ als obere Schranke für die Anzahl insgesamt zu berechnender Potentialfunktionen.

Aus Kapitel 3 ist bekannt, dass sich die Potentialfunktion von CRFs als Summe über den Potentialen aller beobachteten Nachbarknoten eines Faktorknotens darstellen lassen (5.6).

$$\ln f\left(\mathbf{y}_{\Delta(j)}, \mathbf{x}_{\tilde{\Delta}(j)}\right) = \sum_{v \in \tilde{\Delta}(j)} \ln f\left(\mathbf{y}_{\Delta(j)}, \mathbf{x}_v\right) = \sum_{v \in \tilde{\Delta}(j)} \left\langle \mathbf{f}_j\left(\mathbf{y}_{\Delta(j)}, \mathbf{x}_v\right), \boldsymbol{\theta}_j \right\rangle \tag{5.6}$$

Da die Nachbarschaft eines Faktorknotens von der konkreten graphischen Struktur abhängt, kann die Anzahl der zu summierenden Terme ebenfalls nur asymptotisch mit $\mathcal{O}(\tilde{\Delta}_{max})$ angegeben werden, wobei $\tilde{\Delta}_{max}$ die Größe der größten beobachteten Nachbarschaft aller Faktorknoten bezeichnet. Da zur Berechnung allein die Parameter verwendet werden, sind die einzelnen $\mathcal{O}(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}})$ Auswertungen der Potentialfunktion unabhängig voneinander und können parallel ausgeführt werden. Die Anzahl der Beispiele im Batch $|\mathcal{B}|$ ist eine Konstante, also können analog zu Abschnitt 4.2.3 $|\mathcal{B}|$ parallele Blöcke der Funktion instanziiert werden. Die Anzahl der Faktorknoten ist zwar fest, jedoch ist aus Kapitel 3 bekannt, dass die Anzahl der versteckten Knoten einzelner Beispiele variabel sein kann und damit einhergehend auch die Anzahl der Faktorknoten. Daher werden zur Parallelisierung W_F Worker-Blöcke verwendet. Dasselbe gilt für die Anzahl beobachteter Realisationen. Diese ist ebenfalls variabel, was die Verwendung von W_Y Worker-Threads motiviert.

Algorithmus 10 Berechung der Potentialfunktionen bei BP und LBP.

```
 1: berechnePotentiale(){
 2:   forall(Block,|B|) Beispiel i ∈ B{
 3:     forall(Block, W_F) Faktorknoten j ∈ F{
 4:       for Realisation x aller beobachteten Nachbarn von j{
 5:         forall(Thread, W_Y) unter x bekannte Realisation y_Δ(j){
 6:           Potential[i][j][y_Δ(j)] += Parameter[j,x,y_Δ(j)]
 7:         }
 8:       }
 9:     }
10:   }
11: }
```

Der parallele Pseudocode ist in Alg. 10 zu sehen. Die Addition der Gewichte auf das Potential der Realisation $\mathbf{y}_{\Delta(j)}$ (Zeile 6) wird parallel von $|\mathcal{B}|\,W_F W_Y$ Threads, aufgeteilt in $|\mathcal{B}|\,W_F$ Blöcke zu jeweils W_Y Threads, ausgeführt. Dabei sind die Speicherzugriffe so organisiert, dass Threads, mit nebeneinanderliegender ThreadID, auf im Speicher nebeneinanderliegende Parameter zugreifen. Die Threads des Blocks (i, j) berechnen also die Potentiale aller bekannten Realisationen des j-ten Faktorknotens des i-ten Beispiels. Die berechneten Potentiale werden im globalen Speicher abgelegt. Falls ein Parameter-Tying eingesetzt wird, wird in Zeile 6 anstatt der Nummer des Faktorknotens j die Nummer des Templates verwendet, welche aus dem Deskriptor des Faktorknotens ausgelesen werden kann (s. Abs. 5.3.4).

Die Berechnung der $|F|$ lokalen Funktionen wird unter W_F Worker-Blöcken aufgeteilt. Selbiges gilt für die $\mathcal{O}(|\mathcal{Y}|^{\Delta_{max}})$ Parameter, die von den W_Y Worker-Threads geladen und aufsummiert werden. Allein die Schleife über die $\mathcal{O}(\tilde{\Delta}_{max})$ beobachteten Nachbarknoten wurde nicht parallelisiert. Die Schrittkomplexität des Algorithmus beträgt damit $S_{Pot}(n) = \mathcal{O}(\lceil |F| / W_F \rceil \left\lceil |\mathcal{Y}|^{\Delta_{max}} / W_Y \right\rceil \tilde{\Delta}_{max})$ und die Arbeitskomplexität $W_{Pot}(n) = \mathcal{O}(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}}\,\tilde{\Delta}_{max})$. Der Faktor $|\mathcal{B}|$ lässt sich nur in der Arbeitskomplexität finden, da die Berechnungen über allen Beispielen im Batch vollständig parallelisiert wurde.

Der Versuch die Schleife über die Realisationen der beobachteten Nachbarn ebenfalls zu parallelisieren scheiterte. Die Verwendung weiterer Worker-Blöcke oder -Threads, führte in Zeile 6 zu Schreibkonflikten. Da die Reihenfolge der Ausführung einzelner Threads nicht beeinflussbar ist, kommt es immer dann zu einem Schreibkonflikt – und damit zu einem undefinierten Ergebnis –, wenn mehr als ein Thread des Blocks (i, j) sein Gewicht auf das Potential der Realisation $\mathbf{y}_{\Delta(j)}$ addieren muss. Zur Lösung dieses Problems wurde versucht, die Summe der Gewichte in Partialsummen zu zerlegen, d.h. jeder Thread besitzt ein eigenes `Potential` Array in welches er seine Gewichte aufaddiert. Die einzelnen Arrays sollten im Anschluss per paralleler Reduktion zu einem aggregiert werden (s. Abb. 5.4). Warum dieses Vorgehen fehl schlug wird im Folgenden erläutert.

Partialsummen im gemeinsamen Speicher. Die Reduktion aus Kapitel 4 verwendet den gemeinsamen Speicher zur Ablage der Zwischenergebnisse. Werden die `Potential` Arrays der Threads im gemeinsamen Speicher abgelegt, so ist die Anzahl maximal repräsentierbarer Realisationen versteckter Nachbarschaften durch die Größe des gemeinsamen Speichers (16 KiB) nach oben beschränkt. Die Anzahl der Realisationen die maximal beobachtet werden dürfen damit 16 KiB gemeinsamer Speicher ausreichen sei R_{max}. Gleichung (5.7) zeigt, wie die konkrete obere Schranke bei Verwendung eines Datentyps der Größe s berechnet werden kann. Bei Verwendung

des Datentyps `float` ($s = 4\,\text{Byte}$) sowie 16 Worker-Threads pro Block ($W_Y = 16$) wären maximal 256 Realisationen repräsentierbar.

$$\begin{aligned} R_{max} \cdot W_Y \cdot s &= 16 \cdot \text{KiB} \\ \Leftrightarrow R_{max} &= \frac{16 \cdot 1024}{W_Y \cdot s} \end{aligned} \tag{5.7}$$

Dies würde allerdings nicht einmal für die in Abschnitt 6.3.1 verwendeten Daten ausreichen. Diese enthalten 22 Klassen und als graphische Struktur wird ein Linear-Chain Graph verwendet. Die größten Cliquen des Graphen sind also die Kanten und es gilt $\Delta_{max} = 2$. Die Anzahl möglicher Realisationen beträgt demnach $|\mathcal{Y}|^{\Delta_{max}} = 22^2 = 484$ von denen maximal $R_{max}/\,|\mathcal{Y}|^{\Delta_{max}} = 52\%$ repräsentierbar wären. Im Fall von acht Worker-Threads wären zwar 100% der Realisationen repräsentierbar, trotzdem würde eine geringe Erhöhung von $|\mathcal{Y}|$ oder Δ_{max} ausreichen, um die Ausführung zu verhindern. Diese Einschränkung wurde bei der Entwicklung als untragbar eingestuft und daher die Verwendung des gemeinsamen Speichers zur Ablage der Partialsummen verworfen.

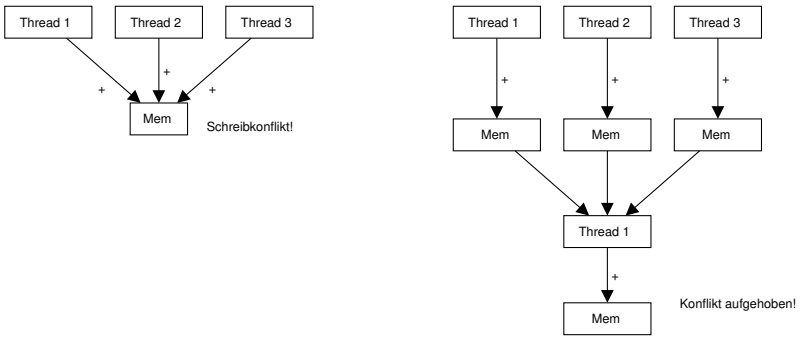


Abbildung 5.4: Partialsummenzerlegung zur Lösung von Schreibkonflikten. Schreiben mehrere Threads gleichzeitig an dieselbe Adresse, so ist das Ergebnis undefiniert (links). Erhält jeder Threads seinen eigenen Speicherbereich, so kann der Schreibkonflikt aufgelöst werden, allerdings impliziert dieses Vorgehen bei P Threads eine ver-P-fachung des Speicherbedarfs (rechts).

Partialsummen im globalen Speicher. Es ist ebenfalls denkbar, die `Potential` Arrays der Threads im globalen Speicher abzulegen. Dieses Vorgehen bringt keine praktisch relevanten Einschränkungen mit sich, da die Größe des globalen Speichers die des gemeinsamen Speichers um mehrere Größenordnungen übertrifft. Allerdings steigt die Anzahl lesender globaler Speicherzugriffe durch die Reduktion der W_Y `Potential` Arrays um $R \cdot W_Y$, wobei R die Anzahl bekannter Realisationen ist. Da jeder dieser Zugriffe eine sehr hohe Latenz (s. Abs. 4.2.4) aufweist, sollte ein solches Vorgehen vermieden werden. Des Weiteren spricht die Verringerung der Berechnungsintensität aufgrund der gesteigerten Anzahl globaler Speicherzugriffe ebenfalls gegen eine Verwendung des globalen Speichers. Ist A die Anzahl arithmetischer Operationen sowie G die Anzahl globaler Speicherzugriffe von Alg. 10, so beträgt dessen Berechnungsintensität A/G. Die Berechnungsintensität desselben Algorithmus' mit Partialsummenzerlegung im globalen Speicher liegt bei $A/\,(G + R \cdot W_Y)$ und ist damit geringer. Die Schleife wurde letztendlich nicht parallelisiert.

Potentiale für den FB Algorithmus. Aus Kapitel 3 ist bekannt, dass es nicht sinnvoll ist, die lokalen Funktionen, die zur Berechnung der Faktornachrichten notwendig sind, in jeder (L)BP Iteration neu zu berechnen. Stattdessen können diese vor dem Beginn des Nachrichtsnaustauschs berechnet und solange gespeichert werden, bis (L)BP terminiert.

Der Forward-Backward Algorithmus benötigt das Potential jeder Realisation jedes Faktorknotens genau zwei Mal (vgl. Abs. 3.6.1). In diesem Fall erweist es sich als praktikabel die berechneten

Potentiale im gemeinsamen Speicher abzulegen, obwohl dies bedeutet, dass die Potentiale mehrfach berechnet werden müssen. Allerdings ist dieses Vorgehen auch mit Einschränkungen verbunden. Die Anzahl maximal darstellbarer Label sei Y_{max}. Die maximale Anzahl versteckter Nachbarn eines Faktorknotens beträgt in einem Linear-Chain Graphen 2. Die Größe s des verwendeten Datentyps betrage 4 Byte.

$$\begin{aligned} & Y_{max}^2 \cdot s = 16 \cdot \text{KiB} \\ \Leftrightarrow\ & Y_{max} = \sqrt{s \cdot 1024} \end{aligned} \tag{5.8}$$

Mit Gleichung (5.8) ergibt sich $Y_{max} = 64$, was als obere Schranke für die Anzahl repräsentierbarer Label als ausreichend eingestuft wurde. Durch dieses Vorgehen steigt die Berechnungsintensität um die Anzahl benötigter Potentiale. Dies resultiert in einer erheblichen Beschleunigung des Forward-Backward Algorithmus (s. Kap. 6).

Da der gemeinsame Speicher zwischen zwei Kernelaufrufen ungültig wird, mussten die Potentialfunktionen des FB Algorithmus als `__device__`-Methoden implementiert werden. Algorithmus 10 wurde dahingehend verändert, dass die Berechnung nicht mehr an allen Faktorknoten parallel stattfindet, sondern der Index der zu berechnenden lokalen Funktion als Argument übergeben wird. So kann der aufrufende Kernel entscheiden, welches Potential tatsächlich berechnet wird. Die endgültige Methode zur Berechnung der lokalen Funktionen im FB Algorithmus ist in Alg. 11 zu sehen. Die Potentialfunktionen werden im folgenden Abschnitt verwendet, um damit die Randverteilungen zu berechnen.

Algorithmus 11 Berechung der Potentialfunktionen beim Forward-Backward Algorithmus.

```
1: berechnePotentiale(Beispiel i, Faktorknoten j){
2:   for Realisation x aller beobachteten Nachbarn von j{
3:     forall(Thread, W_Y) unter x bekannte Realisationen y_Δ(j){
4:       Potential[i][j][y_Δ(j)] += Parameter[j,x,y_Δ(j)]
5:     }
6:   }
7: }
```

5.4.2 Inferenz

Zur Berechnung der Randverteilung werden die aus Abs. 3.6 bekannten Inferenzalgorithmen FB, BP und LBP implementiert. Wie auch im vorangegangenen Abschnitt werden dazu zuerst eventuelle Datenabhängigkeiten identifiziert und daraus eine mögliche Parallelisierung abgeleitet. Im Anschluss wird der parallele Pseudocode vorgestellt und diskutiert. Dabei wird der FB Algorithmus getrennt von den beiden BP Varianten behandelt, da sich aus der festgelegten Linear-Chain Struktur beim FB Algorithmus erhebliche Unterschiede in der Implementierung ergeben haben.

Belief-Propagation. Die Anzahl zu berechnender Randverteilungen ist mit der Anzahl auszuwertender Potentialfunktionen identisch. Insgesamt müssen nach abgeschlossener Inferenz $\mathcal{O}(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}})$ Wahrscheinlichkeiten im globalen Speicher liegen. Eine für jede bekannte Realisation jedes Faktorknotens jedes Beispiels. Wie in Abschnitt 3.6 beschrieben, werden dazu die Faktor und Variablennachrichten berechnet, wobei hier zur Wahrung der numerischen Stabilität die logarithmischen Versionen (5.9) und (5.10) verwendet werden (vgl. Abs. 5.2). Aufgrund der

besseren Lesbarkeit wird im Folgenden von Faktor- und Variablennachrichten gesprochen, obwohl eigentlich Log-Faktor- und Log-Variablennachrichten gemeint sind.

$$\ln m_{f\to v}(y \mid \mathbf{x}) = \bigoplus_{\mathbf{y'}_{\Delta(f)-v} \in \mathcal{Y}^{|\Delta(f)|-1}} \ln f\left(v = y, \mathbf{y'}_{\Delta(f)-v} \mid \mathbf{x}_{\tilde{\Delta}(\mathbf{x})}\right) + \sum_{u \in \Delta(f)-v} \ln m_{u\to f}\left(\mathbf{y'}_u \mid \mathbf{x}\right) \tag{5.9}$$

$$\ln m_{v\to f}(y \mid \mathbf{x}) = \sum_{g \in \Delta(u)-f} \ln m_{g\to v}(y \mid \mathbf{x}) \tag{5.10}$$

Faktor- und Variablennachrichten sind datenabhängig voneinander, da zur Berechnung einer der beiden Nachrichten der Wert der jeweils anderen benötigt wird. Aus diesem Grund werden beide Typen von Nachrichten abwechselnd berechnet. Da zusätzlich nur die Potentiale benötigt werden, sind die einzelnen Instanzen der Nachrichten unabhängig voneinander. Das heißt, dass alle benötigten Faktor- und Variablennachrichten parallel berechnet werden können. Die genaue Parallelisierung hängt dabei vom gewählten Scheduling ab. Im Folgenden wird zuerst die Loopy-Belief-Propagation und anschließend die serielle Belief-Propagation betrachtet. Hier sei angemerkt, dass vor jeder Berechnung die Variablen- und Faktornachrichten mit 0 (also $\ln(1)$, s. Abs. 3.6) initialisiert werden. Die Initialisierung der Variablen wird aus Gründen der Übersichtlichkeit nicht dargestellt.

Algorithmus 12 Paralleler Pseudocode der Loopy-Belief-Propagation.

```
 1: loopyBeliefPropagation(Schranke I){
 2:   berechnePotentiale()
 3:   for t = 0 to I{
 4:     forall(Block, |B|) Beispiel i{
 5:       forall(Block, W_F) Faktorknoten j{
 6:         berechneLogFaktornachrichten(i, j)
 7:       }
 8:       forall(Block, W_V) versteckter Knoten v{
 9:         berechneLogVariablennachrichten(i, v)
10:       }
11:     }
12:   }
13:   forall(Block, |B|) Beispiel i{
14:     forall(Block, W_F) Faktorknoten j{
15:       berechneFaktorBeliefsUndZx(i, j)
16:     }
17:     forall(Block, W_V) versteckter Knoten v{
18:       berechneVariablenBeliefsUndZx(i, v)
19:     }
20:   }
21: }
```

Flooding-Schedule. Mit Hilfe des Flooding-Schedules können die Randverteilungen in beliebigen Faktorgraphen berechnet werden. Algorithmus 12 zeigt den parallelen Pseudocode der Belief-Propagation mit Flooding-Schedule. Dort bezeichnen W_F und W_V die Anzahlen der Worker-Blöcke eines parallelen Aufrufs. Der Algorithmus bekommt die Anzahl zu absolvierender Itera-

tionen I (vgl. Abs. 3.6) als Argument übergeben. Zu Beginn wird die im vorherigen Abschnitt beschriebene Methode zur Berechnung der Potentiale aufgerufen (Zeile 2). Anschließend liegen diese im globalen Speicher und die Berechnung der Nachrichten kann beginnen. Dazu werden innerhalb der Schleife (Zeile 3-12) zuerst die Nachrichten aller Faktorknoten für alle Beispiele parallel berechnet (Zeile 6) sowie anschließend die Nachrichten aller Variablenknoten (Zeile 9). Die entsprechenden Methoden werden weiter unten beschrieben. Nach Terminierung der Schleife können die Beliefs und die Normalisierungen an allen Knoten des Faktorgraphen parallel berechnet werden (Zeilen 13-20). Für diese Methoden wird kein Pseudocode angegeben, da er im wesentlichen mit den Methoden zur Berechnung der Nachrichten übereinstimmt. Der entsprechende CUDA Code ist der CD-ROM in Anhang D zu entnehmen. Die Berechnung der Beliefs erfolgt dabei mit Hilfe der Gleichungen (5.11) sowie (5.12). Die Normalisierung wird an jedem Knoten als Summe aller Beliefs berechnet (vgl. Abs. 3.6).

$$\ln b_f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}\right) = \ln f\left(\mathbf{y}_{\Delta(f)} \,|\, \mathbf{x}_{\tilde{\Delta}(f)}\right) + \sum_{v \in \Delta(f)} \ln m_{v \to f}\left(\mathbf{y}_v \,|\, \mathbf{x}\right) \tag{5.11}$$

$$\ln b_v\left(y \,|\, \mathbf{x}\right) = \ln m_{v \to f}\left(y \,|\, \mathbf{x}\right) + \ln m_{f \to v}\left(y \,|\, \mathbf{x}\right) \tag{5.12}$$

Die Berechnung der Beliefs der Variablenknoten folgt dabei einem Vorschlag aus [40]. Dort wurde bemerkt, dass eine ausgehende Nachricht eines versteckten Knotens v an seinen benachbarten Faktorknoten f dem Produkt der Nachrichten aller anderen benachbarten Faktorknoten entspricht. Es reicht also aus, die verbleibende Nachricht von f an v mit der Nachricht von v an f zu multiplizieren um den Belief des Knotens v zu berechnen. Aufgrund der logarithmierten Berechnungen werden hier entsprechende Summen an Stelle der Produkte gebildet. Somit wurde jeder wesentliche Schritt des Flooding-Schedules parallelisiert. Die Schrittkomplexität (5.13) ergibt sich aus der Anzahl der Iterationen I, den Anzahlen $|F|, n$ der zu berechnenden Nachrichten sowie den Schrittkomplexitäten der Nachrichtenberechnungen. Die Arbeitskomplexität (5.14) ergibt sich analog unter Berücksichtigung der Arbeitskomplexitäten der Nachrichtenberechnungen.

$$S_{LBP}(n) = \mathcal{O}\left(I\left(\left\lceil \frac{|F|}{W_F} \right\rceil S_{m_{f \to v}} + \left\lceil \frac{n}{W_V} \right\rceil S_{m_{v \to f}}\right)\right) \tag{5.13}$$

$$W_{LBP}(n) = \mathcal{O}\left(I\,|\mathcal{B}|\left(|F|\, W_{m_{f \to v}} + n W_{m_{v \to f}}\right)\right) \tag{5.14}$$

Hier bezeichnet n die Anzahl der versteckten Knoten, $S_{m_{f \to v}}$ und $S_{m_{v \to f}}$ die Schrittkomplexitäten der Faktor- und Variablennachrichten sowie $W_{m_{f \to v}}$ und $W_{m_{v \to f}}$ die entsprechenden Arbeitskomplexitäten. Bevor die Methoden zur Berechnung der Nachrichten diskutiert werden, folgt die Beschreibung des seriellen Schedulings.

Serielles Scheduling. Die serielle BP dient der Inferenz in Faktorbäumen. Dabei werden die Nachrichten der Knoten sequenziell berechnet. Der Psudocode ist in Alg. 13 dargestellt. Da die serielle BP konvergiert, sobald über jede Kante zwei Nachrichten versendet wurden, kann die Anzahl benötigter Iteration leicht berechnet werden (Zeile 2). Die Parallelisierung der Schleife (Zeile 3-10) erfolgt analog zu LBP über alle Beispiele im Batch. Allerdings werden in jeder Iteration nur die Nachrichten eines Faktor- sowie eines Variablenknotens berechnet. Dabei sei Faktorknoten 0 o.B.d.A. ein Faktorblatt. Ist dies nicht so, müssen die Knoten beim Einlesen der Daten umbenannt werden. Die jeweils nächsten Knoten werden in den Zeilen 7 und 9 bestimmt. Die entsprechenden Methoden wählen stets den Nachbarn aus, der bis jetzt am seltensten besucht wurde. Da ein Baum keine Kreise enthält, wird sich die BP in der letzten Iteration wieder an Knoten 0 befinden. Es wäre also ebenfalls denkbar, die `for`- durch eine `while`-Schleife zu ersetzen, welche terminiert, sobald zum zweiten Mal der Knoten 0 besucht wurde. Der Pseudocode zur Berechnung der Beliefs

und Normalisierungen wurde hier ausgelassen, da er zu dem der LBP identisch ist.

Algorithmus 13 Paralleler Pseudocode der seriellen Belief-Propagation.

```
 1: serielleBeliefPropagation(){
 2:   Schranke I = 2|E|, j = 0, v = 0
 3:   berechnePotentiale()
 4:   for t = 0 to I{
 5:     forall(Block, |B|) Beispiel i{
 6:       berechneFaktornachrichten(i, j)
 7:       v = nächsterVariablenknoten(j)
 8:       berechneVariablennachrichten(i, v)
 9:       j = nächsterFaktorknoten(v)
10:     }
11:   }
12:   // Beliefs und Normalisierung:  siehe LBP
13: }
```

Schritt- sowie Arbeitskomplexität ergeben sich analog zur denen des Flooding-Schedules als (5.15) und (5.16).

$$S_{BP}(n) = \mathcal{O}\left(|E|\left(S_{m_{f\to v}} + S_{m_{v\to f}}\right)\right) \tag{5.15}$$

$$W_{BP}(n) = \mathcal{O}\left(|E|\,|\mathcal{B}|\left(W_{m_{f\to v}} + W_{m_{v\to f}}\right)\right) \tag{5.16}$$

Nun werden die parallelen Algorithmen zur Berechnung der Faktor- und Variablennachrichten vorgestellt sowie deren Komplexität ermittelt.

Faktornachrichten. Die hier entwickelte Methode (Alg. 14) zur parallelen Berechnung der Faktornachrichten (5.9) berechnet die Nachricht $\ln m_{f\to v}\left(y \,|\, \mathbf{x}^{(i)}\right)$ für alle Label $y \in \mathcal{Y}$ sowie alle versteckten Nachbarn $v \in \Delta(f)$ und legt diese im globalen Speicher ab. Sie wurde als `__device__`-Methode implementiert um vom BP sowie vom LBP Kernel aufgerufen werden zu können. Die Nummer des Beispiels i sowie die des Faktorknotens j, der die Nachrichten versendet, werden dem Algorithmus als Argument übergeben. Die zentrale Idee bei der Parallelisierung ist, dass nicht für jeden versteckten Nachbarn über alle $|\mathcal{Y}|^{|\Delta(f)|-1}$ möglichen Realisationen der anderen versteckten Nachbarn iteriert werden muss, sondern nur ein Mal über alle $|\mathcal{Y}|^{|\Delta(f)|}$ Möglichkeiten.

Algorithmus 14 Paralleler Pseudocode für die Berechnung der Faktornachrichten.

```
 1: berechneFaktornachrichten(Beispiel i, Faktor j){
 2:   forall(Thread, W_Y) Realisationen y_Δ(j) ∈ 𝒴^|Δ(j)|{
 3:     forall(Thread, W_Δ) versteckter Nachbar v von j{
 4:       Summand = 0
 5:       for versteckter Nachbar u ≠ v von j{
 6:         Summand += Variablennachricht[i][u][j][y_Δ(j),u]
 7:       }
 8:       if(bekannt(y_Δ(j))) Summand += Potential[i][j][y_Δ(j)]
 9:       logAdd(Faktornachricht[i][j][v][y_Δ(j),v][ThreadID], Summand)
10:     }
11:   }
12:   forall(Thread, W_Y) Label y ∈ 𝒴{
13:     sum(Faktornachricht[i][j][v][y])
14:   }
15: }
```

Die Berechnung der Nachrichten wird mit Hilfe von Worker-Threads in W_Y Partialsummen aufgeteilt (Zeile 2), so dass jeder Thread $|\mathcal{Y}|^{|\Delta(f)|}/W_Y$ Iterationen ausführt. Des Weiteren wurde die Berechnung der Nachrichten an die $|\Delta(f)|$ versteckten Nachbarn unter W_Δ Worker-Threads aufgeteilt (Zeile 3). Damit besteht jeder Block aus $W_Y \cdot W_\Delta$ Threads. Zur Parallelisierung der Schleife (Zeilen 4-9) wurden W_Y Partialsummen für jeden versteckten Nachbarn und jedes Label angelegt. Die einzelnen Partialsummen werden am Ende der Methode (Zeile 12-14) zu jeweils einer Faktornachricht für jedes Label und jeden versteckten Nachbarn von den Threads sequenziell aufsummiert.

Innerhalb einer jeden Iteration (Zeilen 4-9) berechnet jeder Thread einen Summanden einer Nachricht die von j an v verschickt wird. Dazu summiert dieser die Log-Variablennachrichten der anderen versteckten Nachbarn auf (Zeile 6) und addiert ggf. das Potential der aktuellen Realisation $\mathbf{y}_{\Delta(f)}$, falls diese bekannt ist (Zeile 7). Die Notation $\mathbf{y}_{\Delta(j),u}$ bezeichnet dabei das Label des Knotens u in der gemeinsamen Realisation $\mathbf{y}_{\Delta(j)}$. Die Ablage eines berechneten Summanden erfolgt durch logarithmische Addition desselben auf die Partialsumme des Threads (Zeile 9). Hierbei ist zu beachten, dass die Realisationen $\mathbf{x}$ der beobachteten Knoten in dieser Formulierung nicht vorkommen. Diese werden ausschließlich zur Auswahl der Parameter bei der Berechnung der Potentialfunktionen verwendet.

Oben wurde erwähnt, dass die Partialsummen im globalen Speicher abgelegt werden. Im Abschnitt über die Potentialfunktionen wurde zwar argumentiert, das dies zu einer Verringerung der Berechnungsintensität führt, hier ist die Situation jedoch eine andere. Testläufe, in denen der gemeinsame Speicher zur Ablage der Partialsummen verwendet wurde, waren $\approx 25\%$ langsamer[12] als bei der Verwendung des globalen Speichers. Dies ist darauf zurückzuführen, dass jeder Multiprozessor bei Verwendung des gemeinsamen Speichers wesentlich weniger Blöcke verwalten kann. Es wird eine Partialsumme pro Worker-Thread für jeden Nachbarn und jedes Label benötigt. Da die Größe des gemeinsamen Speichers für jeden Block gleich sein muss, kann die Anzahl der Nachbarn nur mit Δ_{max} nach oben abgeschätzt werden. Bei der Verwendung eines Datentyps der Größe $s = 4\,\text{Byte}$ sowie der $W_Y = 16$ Threads ergibt sich bei 22 Klassen (CoNLL-2000 Datensatz, Kap. 6) ein Bedarf

[12] CONLL2000 Testdatensatz. 2012 Beispiele, $|\mathcal{B}| = 128$, $W_Y = 16$, $W_\Delta = 2$. Eine Iteration. Gemeinsamer Speicher: 15,17 Sekunden. Globaler Speicher: 12,17 Sekunden.

an gemeinsamen Speicher in Höhe von $W_Y \cdot \Delta_{max} \cdot |\mathcal{Y}| \cdot s \cdot (1/1024) = 2,75$ KiB. Somit kann ein Multiprozessor mit 16 KiB gemeinsamen Speicher $\lfloor 16/2,75 \rfloor = 5$ Blöcke gleichzeitig verwalten. Wird dagegen der globale Speicher verwendet, so kann die maximale Anzahl an Blöcken (vgl. Abs. 4.2.5) bearbeitet werden. Daher wurde hier der globale Speicher zur Ablage der Partialsummen gewählt.

Um die Multiprozessoren stets vollständig auszulasten wurden in der CUDA-Implementierungen $W_Y = 16$ sowie $W_\Delta = 2$ gewählt. Damit ist die Anzahl der Threads pro Block 32, was der Warpgröße der Multiprozessoren entspricht. Prinzipiell wäre es an dieser Stelle möglich die Hardwareinformationen der GPU abzufragen um daraus eine "bessere" Anzahl an Threads zu berechnen, allerdings wurde dieser Ansatz hier nicht weiter verfolgt, da zur Evaluation verschiedene Hardware Versionen notwendig gewesen wären. Auf den verfügbaren Versionen 1.3 und 2.0 ergaben sich für $W_Y = 16$, $W_Y = 32$ sowie $W_Y = 64$ keine wesentlichen Unterschiede, da dieses Vorgehen wiederum die Anzahl maximaler verwaltbarer Blöcke pro MP verringert. Daher wurden W_Y und W_Δ kleinstmöglich gewählt. Weniger als 32 Threads sind auf der hier betrachteten GPU Architektur nicht sinnvoll (vgl. Abs. 4.2.5). Im Unterschied zu den Potentialfunktionen erwies sich eine sequenzielle Summation als langsamer. Tests in denen die Schleifen der Zeilen 2 und 3 vollständig sequenziell ausgeführt wurden zeigten, dass die parallele Variante mit Partialsummen im globalen Speicher mehr als doppelt so schnell ist[13].

Die einzige Schleife, die in Alg. 14 nicht parallelisiert wurde, ist diejenige über die "anderen Nachbarn" (Zeile 5). Es wurde angedacht, die $\Delta_{max} \cdot |\mathcal{Y}|$ Summationen stattdessen mit Δ_{max} parallelen Summenreduktion zu berechnen. Dies wurde jedoch nicht implementiert, da lediglich $W_Y = 16$ Werte pro Nachbar zu addieren sind.

Damit ist die Parallelisierung der Berechnung der Faktornachrichten vollständig erläutert. Schritt- und Arbeitskomplexität ergeben sich zu (5.17) und (5.18). Bei einer ausreichenden Anzahl an Threads wird die Laufzeit also durch die maximale Anzahl versteckter Nachbarn eines Faktorknotens dominiert. Hierzu ist allerdings anzumerken, dass die Anzahl der Worker Threads zur Summation der $|\mathcal{Y}|^{\Delta_{max}}$ möglichen Realisationen exponentiell in der Größe der größten versteckten Nachbarschaft sein muss, um diesen Faktor in der asymptotischen Laufzeit zu "verstecken". Daher könnte man an dieser Stelle argumentieren, dass die Laufzeit trotz der Parallelisierung durch die $|\mathcal{Y}|^{\Delta_{max}}$ Summationen dominiert wird.

$$S_{m_{f\to v}}(n) = \mathcal{O}\left(\left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil \left\lceil \frac{\Delta_{max}}{W_\Delta} \right\rceil \Delta_{max}\right) \tag{5.17}$$

$$W_{m_{f\to v}}(n) = \mathcal{O}\left(|\mathcal{Y}|^{\Delta_{max}} \Delta_{max}^2\right) \tag{5.18}$$

Im Folgenden wird die Methode zur parallen Berechnung der Variablennachrichten vorgestellt.

[13] CONLL2000 Testdatensatz. 2012 Beispiele, $|\mathcal{B}| = 128$. Eine Iteration.
$W_Y = 16$, $W_\Delta = 2$: 11,96 Sekunden. $W_Y = W_\Delta = 1$: 28,45 Sekunden.

Algorithmus 15 Paralleler Pseudocode für die Berechnung der Variablennachrichten.

```
 1: berechneVariablennachrichten(Beispiel i, versteckter Knoten v){
 2:   forall(Thread, W_Δ) benachbarter Faktorknoten j von v{
 3:     for benachbarter Faktorknoten g ≠ j von v{
 4:       forall(Thread, W_Y) Label y ∈ 𝒴{
 5:         logAdd(SMemVariablennachricht[i][v][j][y],
                  Faktornachricht[i][g][v][y])
 6:       }
 7:     }
 8:     forall(Thread, W_Y) Label y ∈ 𝒴{
 9:       Variablennachricht[i][v][j][y] = SMemVariablennachricht[i][v][j][y]
10:     }
11:   }
12: }
```

Variablennachrichten. Die hier entwickelte Methode (Alg. 15) zur parallelen Berechnung der Variablennachrichten (5.9) berechnet die Nachricht $\ln m_{v\to f}\left(y \,|\, \mathbf{x}^{(i)}\right)$ für alle Label $y \in \mathcal{Y}$ sowie alle benachbarten Faktorknoten $f \in \Delta(v)$ und legt diese im globalen Speicher ab. Sie wurde als `__device__`-Methode implementiert um vom BP sowie vom LBP Kernel aufgerufen werden zu können. Die Nummer des Beispiels i sowie die des Variablenknotens v, der die Nachrichten versendet, werden dem Algorithmus als Argument übergeben. Wie auch bei den Faktornachrichten verschickt der Knoten v an jeden seiner Nachbarn $|\mathcal{Y}|$ Nachrichten, welche der Summe der eingehenden Nachrichten der jeweils anderen Nachbarn entsprechen. Die Parallelisierung verläuft über alle mit v benachbarten Faktorknoten. Dazu werden W_Δ Threads (Zeile 2), sowie zur Berechnung der $|\mathcal{Y}|$ Nachrichten, W_Y Threads (Zeile 4) verwendet. Jeder Block besteht also aus $W_\Delta \cdot W_Y$ Threads. Diese wurden analog zur Berechnung der Faktornachrichten so gewählt, dass ihr Produkt $W_\Delta \cdot W_Y$ der Warpgröße entspricht. Die Summation der eingehenden Faktornachrichten erfolgt im gemeinsamen Speicher (Zeile 5). Die fertig berechneten Nachrichten werden abschließend von den Threads in den globalen Speicher kopiert (Zeile 9). Der einzige nicht parallelisierte Teil des Codes ist auch hier (s.o.) die Schleife über die "anderen Nachbarn". Eine Parallelisierung der Schleife hätte zwar den Grad der Parallelität erhöhen, jedoch würde die Berechnungsintensiät pro Thread sinken. Daher wurde an dieser Stelle auf die Verwendung weiterer Threads verzichtet. Schritt- und Arbeitskomplexität des Algorithmus sind in (5.19) bzw. (5.20) angegeben, wobei $\hat{\Delta}_{max}$ die Größe der größten Nachbarschaft eines versteckten Knotens bezeichnet. Bei einer ausreichenden Anzahl an Threads wird also auch hier die Komplexität durch die Größe der Nachbarschaften dominiert.

$$S_{m_{v\to f}}(n) = \mathcal{O}\left(\left\lceil \frac{|\mathcal{Y}|}{W_Y} \right\rceil \left\lceil \frac{\hat{\Delta}_{max}}{W_\Delta} \right\rceil \hat{\Delta}_{max}\right) \tag{5.19}$$

$$W_{m_{v\to f}}(n) = \mathcal{O}\left(|\mathcal{Y}|\, \hat{\Delta}^2_{max}\right) \tag{5.20}$$

Komplexität der Belief-Propagation. Im Folgenden wird die Gesamtkomplexität von BP und LBP angegeben und verglichen. Da nun die Komplexitäten der parallelen Algorithmen zur Berechnung der Potentiale, der Nachrichten sowie der Schedulings bekannt sind, kann die Gesamtkomplexität der Belief-Propagation angegeben werden. Für die Loopy-Belief-Propagation

ergeben sich (5.21) sowie (5.22) als entsprechende Gesamtkomplexitäten.

$$S_{LBP}(n) = \mathcal{O}(\overbrace{\left\lceil \frac{|F|}{W_F} \right\rceil \left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil \tilde{\Delta}_{max}}^{\text{Potentiale}} + I(\overbrace{\left\lceil \frac{|F|}{W_F} \right\rceil \left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil \left\lceil \frac{\Delta_{max}}{W_\Delta} \right\rceil \Delta_{max}}^{\text{Faktornachrichten}} + \underbrace{\left\lceil \frac{n}{W_V} \right\rceil \left\lceil \frac{|\mathcal{Y}|}{W_Y} \right\rceil \left\lceil \frac{\hat{\Delta}_{max}}{W_\Delta} \right\rceil \hat{\Delta}_{max}}_{\text{Variablennachrichten}})) \tag{5.21}$$

$$W_{LBP}(n) = \mathcal{O}(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}}\,\tilde{\Delta}_{max} + I\,|\mathcal{B}|\,(\underbrace{|F|\,|\mathcal{Y}|^{\Delta_{max}}\,\Delta_{max}^2}_{\text{Faktornachrichten}} + n\,|\mathcal{Y}|\,\hat{\Delta}_{max}^2)) \tag{5.22}$$

Vor allem die Schrittkomplexität präsentiert sich in einer recht unübersichtlichen Form. Geht man allerdings davon aus, dass die verwendete Hardware tatsächlich über beliebig viele Recheneneinheiten verfügt, so kollabiert der Ausdruck für die Schrittkomplexität zu Gleichung (5.23). Aufgrund der Summation exponentiell vieler Terme ist allerdings davon auszugehen, dass die reale Laufzeit durch den Term für die Faktornachrichten in der Arbeitskomplexität (5.22) dominiert wird. Hierbei ist zu beachten, dass der Faktor $|\mathcal{Y}|^{\Delta_{max}}$ zwar auch im Term für die Potentialfunktionen auftaucht, jedoch ist dies eine sehr großzügige Abschätzung der Anzahl zu berechnender Potentiale, da aus Kapitel 3 bekannt ist, dass nur wenige dieser Realisationen tatsächlich in den Trainingsdaten vorkommen. Zur Berechnung der Faktornachrichten muss dagegen über alle $|\mathcal{Y}|^{\Delta_{max}}$ Möglichkeiten iteriert werden – unabhängig davon, ob sie jemals beobachtet wurden oder nicht.

$$S'_{LBP}(n) = \mathcal{O}\left(\tilde{\Delta}_{max} + I\left(\Delta_{max} + \hat{\Delta}_{max}\right)\right) \tag{5.23}$$

Für die serielle Belief-Propagation zeichnet sich ein ähnliches Bild ab. Die entsprechenden Komplexitäten sind in (5.24) und (5.25) dargestellt. Da dort allerdings nur die Nachrichten eines Faktorknotens pro Iteration berechnet werden, ist zu erwarten, dass die Laufzeit um den Faktor $\mathcal{O}(|F|)$ geringer ist als bei der Verwendung des Flooding-Schedules. Nimmt man wiederum an, dass beliebig viele Prozessoren zur Verfügung stehen, so sind die Schrittkomplexitäten von BP und LBP mit Ausnahme der Anzahl zu absolvierender Iterationen identisch. In der CUDA-Implementierungen wird die Anzahl der LBP Iterationen stets mit der Anzahl der BP Iterationen $2\,|E|$ approximiert.

$$S_{BP}(n) = \mathcal{O}(\overbrace{\left\lceil \frac{|F|}{W_F} \right\rceil \left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil \tilde{\Delta}_{max}}^{\text{Potentiale}} + I(\overbrace{\left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil \left\lceil \frac{\Delta_{max}}{W_\Delta} \right\rceil \Delta_{max}}^{\text{Faktornachrichten}} + \underbrace{\left\lceil \frac{|\mathcal{Y}|}{W_Y} \right\rceil \left\lceil \frac{\hat{\Delta}_{max}}{W_\Delta} \right\rceil \hat{\Delta}_{max}}_{\text{Variablennachrichten}})) \tag{5.24}$$

$$W_{BP}(n) = \mathcal{O}\left(|\mathcal{B}|\,|F|\,|\mathcal{Y}|^{\Delta_{max}}\,\tilde{\Delta}_{max} + |E|\,|\mathcal{B}|\left(\underbrace{|\mathcal{Y}|^{\Delta_{max}}\,\Delta_{max}^2}_{\text{Faktornachrichten}} + |\mathcal{Y}|\,\hat{\Delta}_{max}^2\right)\right) \tag{5.25}$$

Im Folgenden wird die parallele Variante der Belief-Propagation für Linear-Chain Graphen vorgestellt: der parallele Forward-Backward Algorithmus.

Forward-Backward Algorithmus. Der Forward-Backward Algorithmus ist auf die Inferenz in Linear-Chain Graphen ausgelegt. Grundsätzlich ist dieser mit der seriellen Belief-Propagation identisch, falls diese ebenfalls auf einem Linear-Chain Graphen (Abb. 5.5) durchgeführt wird. Der wesentliche Unterschied besteht jedoch darin, dass die Größen der größten Nachbarschaften Δ_{max} und $\hat{\Delta}_{max}$ beim Forward-Backward Algorithmus Konstanten sind. Die Anzahl der Faktorknoten eines Linear-Chain Graphen ist eine Funktion der Anzahl der versteckten Knoten.

Algorithmus 16 Pseudocode des parallelen Forward-Backward Algorithmus.

```
 1: forwardBackward(){
 2:   forall(Block, |B|) Beispiel i{
 3:     for t := 1 to n{
 4:       berechnePotential(i, t)
 5:       berechnePotential(i, t − 1,t)
 6:       forall(Thread, W_Y) Label y{
 7:         berechneForwardNachricht(i, t, y)
 8:       }
 9:     }
10:     berechneZx()
11:     resetSMem()
12:     for t := n to 1{
13:       berechnePotential(i, t)
14:       berechnePotential(i, t,t + 1)
15:       forall(Thread, W_Y) Label y{
16:         berechneBackwardNachricht(i, t, y)
17:         berechneRandverteilung(i,t,y)
18:         berechneRandverteilungen(t,t + 1)
19:       }
20:     }
21:   }
22: }
```

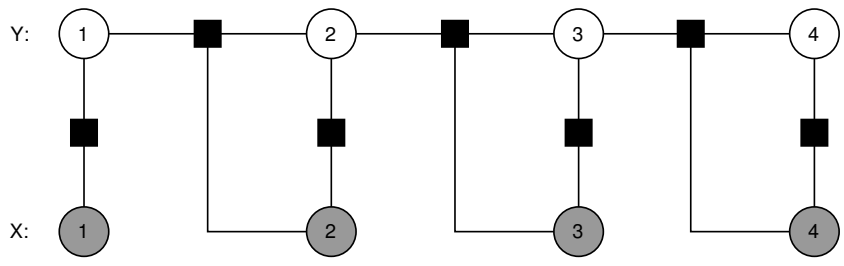


Abbildung 5.5: Der Faktorgraph eines Linear-Chain CRFs mit vier versteckten Knoten.

Da sowohl für jeden versteckten Knoten (1-Clique) als auch für jede Kante (2-Clique) des Abhängigkeitsgraphen ein Faktorknoten existiert, ergibt sich $|F| = 2n - 1$ als Anzahl der Faktorknoten. Verwendet man dazu die in Abs. 3.6.1 erwähnte Vorgängerrelation, lässt sich der Nach-

richtenfluss beim FB Algorithmus über die n versteckten Knoten anstatt über die Faktorknoten formulieren.

$$\ln \alpha_t (y \mid \mathbf{x}) = \bigoplus_{y' \in \mathcal{Y}} \ln f_t (y \mid \mathbf{x}_t) + \ln f_{\{t-1,t\}} \left(y', y \mid \mathbf{x}_t\right) + \ln \alpha_{t-1} \left(y' \mid \mathbf{x}\right) \tag{5.26}$$

$$\ln \beta_t (y \mid \mathbf{x}) = \bigoplus_{y' \in \mathcal{Y}} \ln f_t \left(y' \mid \mathbf{x}_t\right) + \ln f_{\{t,t+1\}} \left(y, y' \mid \mathbf{x}_t\right) + \ln \beta_{t+1} \left(y' \mid \mathbf{x}\right) \tag{5.27}$$

$$\ln Z (\mathbf{x}) = \bigoplus_{y \in \mathcal{Y}} \ln \alpha_T (y \mid \mathbf{x}) \tag{5.28}$$

Der parallele FB Algorithmus ist in Alg. 16 dargestellt. Die Parallelisierung erfolgt, wie auch bei PCRF und CRF++, primär über die Beispiele. Wobei analog zur allgemeinen Belief-Propagation ein Block pro Beispiel verwendet wird (Zeile 2). Zu Beginn wird sequenziell über alle versteckten Knoten in aufsteigender Reihenfolge iteriert (Zeile 3), wobei der versteckte Knoten am linken Rand des Graphen o.B.d.A. Knoten 1 sei. Innerhalb einer jeden Iteration werden zuerst die lokalen Funktionen des t-ten versteckten Knotens berechnet, also $\ln f_t (y \mid \mathbf{x}_t)$ sowie $\ln f_{\{t-1,t\}} \left(y', y \mid \mathbf{x}_t\right)$. Falls ein Knoten nicht existiert, so sind die Werte aller lokalen Funktionen die diesen verwenden 0. Anschließend werden die Forward-Nachrichten (5.26) von W_Y Threads parallel für alle Label ausgewertet. Die Berechnung ist in Alg. 17 dargestellt.

Algorithmus 17 Berechnung der Forward-Nachrichten.

```
1: berechneForwardNachricht(Beispiel i, versteckter Knoten t, Label y){
2:   for Label y' ∈ 𝒴 {
3:     Summand = potential[i][t][y] + potential[i][t − 1][t][y'][y]
                 + SMemAlpha[i][y']
4:     logAdd(NewSMemAlpha[i][y], Summand)
5:   }
6:   alpha[i][t][y] = NewSMemAlpha[i][y]
7: }
```

Wie im Abschnitt über die Potentialfunktionen bereits erwähnt wurde, werden die Potentiale beim FB Algorithmus im gemeinsamen Speicher abgelegt. Dasselbe gilt hier für die Forward-Nachrichten (Alg. 17, Zeilen 3 und 4). Jeder Operand der Berechnungen der Nachrichten wird entweder aus dem gemeinsamen Speicher gelesen oder in diesen geschrieben, wobei die Nachrichten der vorangegangenen Iteration stets überschrieben werden. Am Ende der Berechnung (Alg. 17, Zeile 6) werden diese in den globalen Speicher kopiert, da sie später für die Randverteilungen benötigt werden. Wurden die Nachrichten an allen versteckten Knoten berechnet, so wird mit diesen in Zeile 10 des FB Algorithmus die Normalisierung (5.28) bestimmt. Der entsprechende Pseudocode ist hier nicht dargestellt, da er in wesentlichen Punkten mit Alg. 17 übereinstimmt. Auch die Normalisierungen werden nach der Berechnung im globalen Speicher abgelegt. Die Methode in Zeile 11 soll andeuten, dass an dieser Stelle alle Werte im gemeinsamen Speicher auf 0 zurückgesetzt werden. Abschließend wird sequenziell über alle versteckten Knoten in absteigender Reihenfolge iteriert (Zeile 12). Innerhalb jeder Iteration (Zeilen 13-19) werden erneut die Potentiale berechnet. Dies ist notwendig, da die Potentiale im gemeinsamen Speicher abgelegt werden und dieser stets durch nachfolgende Berechnungen überschrieben wird. Da die Potentiale bei der Belief-Propagation im

globalen Speicher abgelegt wurden, war deren Berechnung nur einmalig notwendig.

$$p\left(v_t = y \,|\, \mathbf{x}\right) = \exp\left(\ln \alpha_t \left(y \,|\, \mathbf{x}\right) + \ln \beta_t \left(y \,|\, \mathbf{x}\right) - \ln Z\left(\mathbf{x}\right)\right) \tag{5.29}$$

$$\begin{aligned} p\left(v_{t-1} = y', v_t = y \,|\, \mathbf{x}\right) &= \exp(\ln \alpha_{t-1}\left(y' \,|\, \mathbf{x}\right) + \ln f_t\left(y \,|\, \mathbf{x}_t\right) + \ln f_{\{t-1,t\}}\left(y', y \,|\, \mathbf{x}_t\right) \\ &+ \ln \beta_t\left(y \,|\, \mathbf{x}\right) - \ln Z\left(\mathbf{x}\right)) \end{aligned} \tag{5.30}$$

Anschließend werden parallel mit W_Y Threads die Backward-Nachrichten (5.27) sowie die Randverteilungen (5.29, 5.30) bestimmt. Die Berechnung der Backward-Nachrichten (Alg. 16, Zeile 16) verläuft analog zu Alg. 17, wobei die Backward-Nachrichten nicht im globalen Speicher abgelegt werden. Stattdessen werden diese direkt im Anschluss verwendet um zusammen mit den Forward-Nachrichten und den Normalisierungen die Randverteilungen zu bestimmen. Dabei entspricht Zeile 17 der Berechnung von Gleichung (5.29) und Zeile 18 einer sequenziellen Schleife über alle möglichen Label des Vorgängers eines Knotens. So werden insgesamt $\mathcal{O}(|\mathcal{B}|\, n(|\mathcal{Y}| + |\mathcal{Y}|^2))$ Randverteilungen berechnet. Eine für jede mögliche Realisation, jedes Knotens und jeder Kante für jedes Beispiel.

$$S_{FB}\left(n\right) = \mathcal{O}\left(n \left\lceil \frac{|\mathcal{Y}|}{W_Y} \right\rceil |\mathcal{Y}| + nS_{Pot}\right) \tag{5.31}$$

$$W_{FB}\left(n\right) = \mathcal{O}\left(|\mathcal{B}|\, n\, |\mathcal{Y}|^2 + |\mathcal{B}|\, nW_{Pot}\right) \tag{5.32}$$

Der FB Algorithmus berechnet also mit der Schrittkomplexität (5.31) sowie der Arbeitskomplexität (5.32) alle Randverteilungen. Da die Anzahl der Label durch Verwendung des gemeinsamen Speichers ohnehin auf 64 beschränkt ist (s. Abs. 5.4.1), wurde in der CUDA-Implementierung stets $W_Y = |\mathcal{Y}|$ gewählt, so dass sich eine Schrittkomplexität von $S'_{FB}\left(n\right) = \mathcal{O}(n\,|\mathcal{Y}| + nS_{Pot})$ ergibt. Die Komplexitäten der Potentialberechnungen wurden hier nicht substituiert, da die Laufzeit des FB Algorithmus in der Literatur (z.B: [67]) oft ohne Berücksichtigung der Potentialfunktionen angegeben wird. Dies mag daran liegen, dass der Algorithmus ursprünglich von Rabiner für Hidden Markov Modelle vorgestellt wurde. Da diese generativ sind, müssen die Abhängigkeiten zwischen den Beobachtungen modelliert werden. Aus diesem Grund ist jeder Faktorknoten in HMMs mit nur einem beobachteten Knoten benachbart, wodurch die Summation der Parameter zur Berechnung der Potentialfunktion vernachlässigt werden kann. In CRFs ist die Situation bekanntlich eine andere. Da die Anzahl bekannter Parameter nur mit $\mathcal{O}(|\mathcal{Y}|^2)$ nach oben abgeschätzt werden kann und die Nachbarschaft höchstens $\tilde{\Delta}_{max}$ beobachtete Knoten enthält, ergibt sich eine obere Schranke für die Schrittkomplexität des hier gezeigten FB Algorithmus in Höhe von $S^U_{FB}\left(n\right) = \mathcal{O}(n\,|\mathcal{Y}|^2\,\tilde{\Delta}_{max})$. Schätzt man die Anzahl der beobachteten Nachbarn sowie die Anzahl der bekannten Parameter wie üblich mit $\Omega\left(1\right)$ ab, so ergibt sich $S^L_{FB}\left(n\right) = \Omega\left(n\,|\mathcal{Y}|\right)$ als untere Schranke für die Schrittkomplexität. Da diese der klassischen Laufzeitkomplexität entspricht (vgl. Abs. 4.1), ist – zumindest auf realen Parallelprozessoren – mit einem erheblichen Laufzeitunterschied im Vergleich mit BP und LBP zu rechnen. Allerdings auf Kosten der Allgemeingültigkeit.

Während der Entwicklung wurde versucht die Forward- und Backwardnachrichten parallel zu berechnen, d.h. jedes α_t wird parallel zu β_{T-t} berechnet. Allerdings stellte sich dieser Ansatz als ineffizient gegenüber der sequenziellen Berechnung der Nachrichten heraus, da in diesem Fall die Randverteilungen nicht mehr direkt nach den Backward-Nachrichten berechnet werden können. Stattdessen müssen die Backward-Nachrichten nach der Berechnung ebenfalls im globalen Speicher abgelegt und später zur Berechnung der Randverteilungen aus diesem gelesen werden, wodurch die Anzahl globaler Speicherzugriffe nahezu verdoppelt wird.

In diesem Abschnitt wurden drei Algorithmen zur Berechnung der Randverteilungen in allgemeinen Graphen, in kreisfreien Graphen sowie in Linear-Chain Graphen analysiert und parallelisiert. Die Komplexität der entsprechenden Algorithmen fällt mit der Komplexität der graphischen Struktur. Die realen Laufzeiten der Algorithmen werden in Kapitel 6 experimentell analysiert.

Im nächsten Abschnitt werden zwei Klassifikationsalgorithmen vorgestellt, die in Kombination mit den soeben erläuterten Inferenzalgorithmen eingesetzt werden.

5.4.3 Klassifikation

Um ein trainiertes CRF auf ungelabelte Daten anzuwenden, werden Klassifikationsalgorithmen benötigt. Mit diesen können die Realisationen mit maximaler Wahrscheinlichkeit berechnet werden. Zur Klassifikation werden die Beispiele ebenfalls in Batches partitioniert, wobei es formal keinen Unterschied macht, ob die Beispiele parallel oder in Reihe klassifiziert werden. Im Folgenden wird der Klassifikationalgorithmus vorgestellt, der in der hier entwickelten Implementierung zusammen mit BP und LBP eingesetzt wird. Anschließend wird der Viterbi-Algorithmus parallelisiert, der ausschließlich zur Klassifikation in Linear-Chain CRFs geeignet ist.

Algorithmus 18 Paralleler Pseudocode der Maximum-Belief Klassifikation.

```
 1: berechneMaxBelief(){
 2:   forall(Block, |B|) Beispiel i{
 3:     forall(Thread, W_n) Versteckter Knoten v
 4:       meinMaxBelief = -1;
 5:       for Label y ∈ Y{
 6:         if(maxBelief < belief[i][v][y]){
 7:           meinMaxBelief = belief[i][v][y]
 8:           meinMaxLabel = y
 9:         }
10:       }
11:       maxLabel[i][v] = y
12:     }
13:   }
14: }
```

Maximum-Belief Klassifikation. Bei beiden Varianten der Belief-Propagation geschieht die Klassifikation, indem an jedem versteckten Knoten das Label mit maximalem Belief vorhergesagt wird. Dies ist allerdings erst möglich, nachdem die Beliefs mit BP oder LBP berechnet wurden. Anschließend wird Algorithmus 18 verwendet um den versteckten Knoten die wahrscheinlichsten Label zuzuweisen. Die Parallelisierung erfolgt durch $|\mathcal{B}|$ Blöcke zu je W_n Threads (Zeilen 2 und 3). Jeder Thread iteriert über alle Label und bestimmt das Maximum der einzelnen Beliefs (Zeilen 5-10). Abschließend wird dieses im globalen Speicher abgelegt (Zeile 11). Die so erhaltenen Label entsprechen der wahrscheinlichsten Realisation der einzelnen Knoten. Die Schrittkomplexität des MBK Algorithmus beträgt $S(n) = \mathcal{O}(\lceil n/W_n \rceil |\mathcal{Y}|)$ und die Arbeitskomplexität $W(n) = \mathcal{O}(|\mathcal{B}| n) |\mathcal{Y}|$. Hier bezeichnet n die Anzahl versteckter Knoten.

Viterbi Algorithmus. Wie bereits in Kapitel 3 erwähnt, wird zur Klassifikation in Linear-Chain CRFs der Viterbi Algorithmus eingesetzt. Dieser ähnelt in vielen Punkten dem FB Algo-

rithmus. Allerdings werden hier die Max-Forward-Nachrichten verwendet (s. Alg. 19). Im Unterschied zu den gewöhnlichen Forward-Nachrichten, wird nicht über die Potentiale der Label eines Vorgängerknotens summiert, sondern deren Maximum gebildet (Zeilen 5-8). Auf diese Weise wird für jedes Label y des versteckten Knotens t während der Berechnung der Nachricht das wahrscheinlichste Label des Vorgängerknoten $t-1$ bestimmt. Die Nachricht wird im gemeinsamen Speicher (Zeile 10) und das Label des Vorgängers im globalen Speicher abgelegt (Zeile 11). Der gemeinsame Speicher konnte zur Ablage der Vorgängerlabel nicht verwendet weden, da ansonsten die Länge des längsten Beispiels durch die Größe des gemeinsamen Speichers nach oben beschränkt wäre. Im Hinblick darauf, dass aus demselben Grund bereits die maximale Anzahl darstellbarer Label auf 64 beschränkt ist, wurde hier von einer Verwendung des gemeinsamen Speichers abgesehen.

Algorithmus 19 Parallele Berechnung der Max-Forward-Nachrichten.

```
1: berechneMaxForwardNachricht(Beispiel i, versteckter Knoten t, Label y){
2:   maxForward = -1;
3:   for Label y' ∈ 𝒴 {
4:     forward = potential[i][t][y] + potential[i][t − 1][t][y'][y]
                 + SMemAlpha[i][t − 1][y']
5:     if(maxForward < forward){
6:       maxForward = forward
7:       maxLabel = y'
8:     }
9:   }
10:  SMemAlpha[i][t][y] = maxForward
11:  delta[i][t][y] = maxLabel
12: }
```

Der eigentliche Klassifikationsalgorithmus ist in Alg. 20 dargestellt. Dort werden alle Instruktionen parallel für alle Beispiele des Batches ausgeführt (Zeile 2). Initial werden für jeden Knoten und jedes Label die oben erläuterten Max-Forward Nachrichten inklusive der wahrscheinlichsten Vorgänger parallel von W_Y Threads berechnet (Zeilen 6-8). Anschließend (Zeilen 10-17) wird das wahrscheinlichste Label des letzten Knotens als Maximalstelle seiner Nachrichten bestimmt und im globalen Speicher abgelegt. Basierend auf diesem Label werden in der letzten Schleife (Zeilen 18-20) die Label der anderen Knoten rekursiv aus dem `delta`-Array (Zeile 19) abgelesen. Die einzelnen Iterationen der Schleife sind inhärent datenabhängig, da jede Iteration auf dem Ergebnis der jeweils vorangegangenen basiert. Die so erhaltenen Label entsprechen der wahrscheinlichsten Realisation der ganzen Struktur. Die Schrittkomplexität des Viterbi Algorithmus beträgt $S(n) = \mathcal{O}(n \lceil |\mathcal{Y}| / W_Y \rceil |\mathcal{Y}| S_{Pot})$, da W_Y Threads über n Knoten iterieren um dort $|\mathcal{Y}|$ Max-Forward-Nachrichten zu berechnen, welche eine Schrittkomplexität von $\mathcal{O}(|\mathcal{Y}|)$ besitzen. Da dies für $|\mathcal{B}|$ Beispiele parallel ausgeführt wird, ergibt sich eine Arbeitskomplexität von $W(n) = \mathcal{O}\left(|\mathcal{B}| n |\mathcal{Y}|^2 W_{Pot}\right)$. In der CUDA-Implementierung wurde, wie auch beim Forward-Backward Algorithmus, stets $W_Y = |\mathcal{Y}|$ gewählt.

Algorithmus 20 Der parallele Viterbi-Algorithmus.

```
 1: viterbi(){
 2:   forall(Block, |B|) Beispiel i{
 3:     for t := 1 to n{
 4:       berechnePotential(i, t)
 5:       berechnePotential(i, t − 1,t)
 6:       forall(Thread, W_Y) Label y {
 7:         berechneMaxForwardNachricht(i, t, y)
 8:       }
 9:     }
10:     maxForward = -1;
11:     for Label y ∈ Y{
12:       if(maxForward < alpha[i][T][y]){
13:         maxForward = alpha[i][T][y]
14:         mLabel = y
15:       }
16:     }
17:     maxPropLabel[i][T] = mLabel
18:     for t := n to 1{
19:       maxPropLabel[i][t] = delta[i][t + 1][maxPropLabel[i][t + 1]]
20:     }
21:   }
22: }
```

Beide hier vorgestellten Klassifikationsalgorithmen wurden mit CUDA C implementiert und in Kapitel 6 zur Klassifikation eingesetzt. Im folgenden Abschnitt wird ein paralleler Algorithmus die Berechnung der Erwartung eines CRFs präsentiert.

5.4.4 Erwartung, Gradient und Likelihood

Zur Berechnung des Gradienten der Zielfunktion ist es notwendig die Erwartung des CRFs bezüglich der Realisationen zu kennen (s. Abs. 3.8). Um die Erwartung einer Realisation zu berechnen, müssen die Randverteilungen der Realisation in allen Beispielen aufsummiert werden. Dazu wird in der Definition der partiellen Ableitung (5.33, rechts) über alle $|\mathcal{B}|$ Beispiele sowie über alle Faktorknoten des Templates $\mathcal{F}_h$ iteriert. Über welches Template iteriert werden muss, kann aus dem Parameter-Deskriptor abgelesen werden.

$$\frac{\partial l_{CRF}\left(\boldsymbol{\theta}_{\mathcal{B}};\mathcal{B}\right)}{\partial \boldsymbol{\theta}_{\mathbf{y}_{\mathcal{F}_h},\mathbf{x}_v}} = \sum_{f\in\mathcal{F}_h}\left(\sum_{i=1}^{|\mathcal{B}|}\mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}^{(i)}_{\Delta(f)},\mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right) - \sum_{i=1}^{|\mathcal{B}|}\mathbf{f}_{\mathbf{y}_{\Delta(f)},\mathbf{x}_v}\left(\mathbf{y}_{\Delta(f)},\mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right)\cdot p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(f)}\,|\,\mathbf{x}^{(i)}\right)\right) \tag{5.33}$$

Enthält der Merkmarlsvektors $\mathbf{f}\left(\mathbf{y}^{(i)}_{\Delta(f)},\mathbf{x}^{(i)}_{\tilde{\Delta}(f)}\right)$ am Eintrag $\mathbf{y}_{\Delta(f)},\mathbf{x}_v$ eine 1, so bedeutet dies, dass die Realisationen $\mathbf{y}_{\Delta(f)}$ und $\mathbf{x}_v$ im i-ten Beispiel an Faktorknoten f vorkommen. In diesem Fall wird die Wahrscheinlichkeit aufaddiert, die das Modell dieser Realisation zuweist, nämlich die Randverteilung $p_{\boldsymbol{\theta}}\left(\mathbf{y}_{\Delta(f)}\,|\,\mathbf{x}^{(i)}\right)$. Aus algorithmischer Sicht ist es höchst ineffizient, tatsächlich über alle Beispiele und alle Faktorknoten zu iterieren, um die Erwartung zu berechnen. Stattdessen wird in der hier entwickelten Implementierung bereits beim Einlesen des Batches für jede Realisation eines beobachteten Knotens v in einer Liste vermerkt, ob und in welchen Beispielen diese

vorkommt. Die Länge der längsten Liste sei L_{max}. Sollen nun die Erwartungen des Modells berechnet werden, so wird über die Einträge dieser Listen iteriert anstatt über alle Beispiele. Der parallele Algorithmus zur Berechnung der Erwartungen ist in Alg. 21 dargestellt. In Zeile 2 erfolgt eine Parallelisierung mit W_A Blöcken über alle im Batch $\mathcal{B}$ enthaltenen Realisationen beobachteter Knoten. In Zeile 3 wird sequenziell über die oben erwähnten Listen iteriert. In der 4. Zeile erfolgt eine Parallelisierung über alle unter x bekannten Realisationen mit W_Y Threads. Die Threads laden die entsprechenden Randverteilungen und addieren diese auf den Erwartungswert der Realisation (Zeile 5). Die Speicherzugriffe wurden dabei so organisiert, dass Threads mit nebeneinanderliegender ThreadID stets auf im Speicher nebeneinanderliegende Randverteilungen zugreifen. Ist A die Gesamtanzahl aller beobachteten Realisationen des aktuellen Batches, so beträgt die Schrittkomplexität $S_{\hat{\mathbb{E}}}(n) = \mathcal{O}\left(\left\lceil \frac{A}{W_A} \right\rceil \left\lceil \frac{|\mathcal{Y}|^{\Delta_{max}}}{W_Y} \right\rceil L_{max}\right)$ und dementsprechend die Arbeitskomplexität $W_{\hat{\mathbb{E}}}(n) = \mathcal{O}\left(A\,|\mathcal{Y}|^{\Delta_{max}} L_{max}\right)$. Hierbei wurde die Anzahl bekannter Realisationen erneut mit $|\mathcal{Y}|^{\Delta_{max}}$ nach oben abgeschätzt.

Algorithmus 21 Parallele Berechnung der Erwartung eines CRFs.

```
1: crfErwartung(){
2:   forall(Block, W_A) in B enthaltene Beobachtung x{
3:     for Vorkommen von x im Beispiel i an Faktorknoten j{
4:       forall(Thread, W_Y) unter x bekannte Realisation y_Δ(j){
5:         Erwartung[y_Δ(j), x] += Randverteilung[i][j][y_Δ(j)];
6:       }
7:     }
8:   }
9: }
```

Liegen die Erwartungen des Modells vor, so kann der Gradient berechnet werden. Neben den Modellerwartungen sind dafür die tatsächlichen Häufigkeiten der Realisationen im Batch – die empirischen Erwartungen – erforderlich. Diese werden direkt beim Einlesen eines Batches mitgezählt. Der Pseudocode zur Berechnung des Gradienten ist in Alg. 22 dargestellt. Die Notation `Block+Thread` soll andeuten, dass die gewählte Aufteilung in Blöcke und Threads keine problemabhängige Semantik besitzt. Grundsätzlich könnten die Berechnung auf $|\boldsymbol{\theta}_{\mathcal{B}}|$ Threads aufgeteilt werden, von denen jeder das Update eines Parameters übernimmt. Da die maximale Anzahl an Threads pro Block allerdings auf 512 bzw. 1024 beschränkt ist (s. Abs. 4.2.5), muss die Parallelisierung auf mehrere Blöcke aufgeteilt werden. Daher wurden analog zur Reduktion aus Abs. 4.2.6 64 Blöcke mit jeweils 128 Threads gewählt. Die optimalen Block- und Threadanzahlen sind laut NVidia stark von der Hardware abhängig und sollten durch Experimentieren [50, S.88] bestimmt werden. Die Schrittkomplexität beträgt $S_{Grad}(n) = \mathcal{O}\left(\left\lceil \frac{|\boldsymbol{\theta}_{\mathcal{B}}|}{W_A W_Y} \right\rceil\right)$ und die Arbeitskomplexität $W_{Grad}(n) = \mathcal{O}\left(|\boldsymbol{\theta}_{\mathcal{B}}|\right)$ entspricht der Länge des aktuellen Parametervektors.

Algorithmus 22 Parallele Berechnung des Gradienten der Log-Likelihood.

```
1: berechneGradient(){
2:   forall(Block+Thread, W_B, W_T) Parameter p ∈ θ_B{
3:     Gradient[p] = -(Häufigkeit[p] - Erwartung[p])
4:   }
5: }
```

Die parallele Berechnung des Zielfunktionswerts wurde ebenfalls implementiert. Der Wert der Log-Likelihood entspricht der Summe der logarithmierten Wahrscheinlichkeiten der Beispiele, welche per Summenreduktion gebildet werden kann. Allerdings hat sich während der Entwicklung gezeigt, dass ihr Wert weder für das Training gebraucht wird, noch eine verlässliche Aussage über den Testfehler macht. Daher wurde die Berechnung in den letzten Versionen deaktiviert. Stattdessen werden Accuracy, Precision, Recall und F_1-Score einer Testmenge berechnet. Wird keine Testmenge angegeben, so werden diese Werte für die Trainingsmenge berechnet. Für die Gütemaße wurden allerdings keine parallelen Algorithmen verwendet, diese werden sequenziell von der CPU bestimmt.

5.4.5 Parameterupdate

Für das Update der Parameter wurden die beiden stochastischen Optimierungsverfahren SGD und SMD implementiert. Die entsprechenden Algorithmen sind in Alg. 23 angegeben. Sie sind eine direkte Implementierung der Updateregeln aus Abschnitt 3.8.2. Bei SGD wird das η-fache des negativen Gradienten auf den aktuellen Parametervektor addiert. Bei SMD besitzt jeder Parameter eine eigene Schrittweite. Diese wird in Zeile 3 auf Basis des Gradienten sowie der Gradienten-Historie $\mathbf{v}$ angepasst. Die Aktualisierung von $\mathbf{v}$ erfolgt unter Verwendung des Gradienten sowie dem Produkt von Hesse-Matrix und $\mathbf{v}$ (`Hv`, Zeile 5).

Die Parallelisierung ist dabei identisch zu derjenigen die zur Berechnung des Gradienten verwendet wurde. Wie in dem o.g. Abschnitt erwähnt, muss für SMD der Datentyp `FAD` verwendet werden. Da alle vorherigen Algorithmen zur Berechnung des Gradienten bereits parallelisiert wurden, wurde die Berechnung der Ableitungsinformationen zweiter Ordnung durch die Verwendung von `FAD` ebenfalls parallelisiert. Die Schritt- und Arbeitskomplexitäten von SGD und SMD sind identisch zu denjenigen der Gradientenberechnung. Man beachte, dass sich die Komplexität durch die Verwendung von `FAD` asymptotisch nicht verändert. Das Hesse-Matrix-Vektor-Produkt `Hv` (Alg. 23, `SMDParameterUpdate`, Zeile 5) wird in Linearzeit berechnet.

Die hier verwendete `FAD` Implementierung basiert auf dem original Programmcode der Implementierung die in [72] verwendet wurde. Es mussten lediglich einige Deklarationen angepasst werden, um den Code zu CUDA C kompatibel zu machen. Dieser ist ebenfalls auf der CD-ROM in Anhang D zu finden.

Sowohl CUDA als auch OpenCL und DirectCompute besitzen C-Schnittstellen. Daher sollte die `FAD` Implementierung auf alle GPGPU Frameworks übertragbar sein. Betrachtet man die Parallelisierung, so fällt auf, dass die Berechnung des Gradienten sowie das Parameterupdate zu einer Methode kombiniert werden könnten. Davon wurde allerdings abgesehen, da SGD und SMD in der jetzigen Form auch auf andere Zielfunktionen angewendet werden können. Es wurden keine speziellen Eigenschaften von CRFs oder der Likelihood in der Formulierung der Algorithmen verwendet.

Algorithmus 23 Paralleles Parameterupdate bei SGD und SMD.

```
1: SGDParameterUpdate(){
2:   forall(Block+Thread, W_B, W_T) Parameter p ∈ θ_B{
3:     Parameter[p] -= eta * Gradient[p]
4:   }
5: }
```

```
1: SMDParameterUpdate(){
2:   forall(Block+Thread, W_B, W_T) Parameter p ∈ θ_B{
3:     eta[p] *= max(0.5, 1.0 - mu*Gradient[p]*v[p]);
4:     Parameter[p] -= eta[p] * Gradient[p]
5:     v[p] = lambda*v[p] - eta[p]*(Gradient[p] + lambda*Hv[p]);
6:   }
7: }
```

5.5 Eigene Implementierung

Dieser Abschnitt soll einen Überblick über die hier entwickelte CRF Implementierung geben und einige laufzeitkritische Implementierungsdetails erläutern. Die in den obigen Abschnitten erläuterten Datenstrukturen und Algorithmen wurden in einer Programmbibliothek zusammengefasst. Diese wird hier auch als *Framework* bezeichnet. Die Implementierung sollte so strukturiert werden, dass zentrale Komponenten wie Inferenz- oder Optimierungsalgorithmen austauschbar sind. Entsprechend dem Paradigma der objektorientierten Programmierung erfolgte die Strukturierung der Programmbibliothek mit Hilfe einer C++ Klassenhierarchie [52, 66]. Diese ist in Abbildung 5.6 dargestellt. Die Hierarchie basiert auf abstrakten Klassen (hellgrau), welche keinen GPU Code enthalten. Von diesen kann kein Objekt instanziiert werden, sie geben lediglich Schnittstellen vor, die zur Wahrung der Interoperabilität der Klassen eingehalten werden müssen. Der Postfix `<T>` bedeutet dabei, dass der verwendete Gleitkommadatentyp der Klasse erst bei der Instanziierung eines entsprechenden Objekts angegeben werden muss. Dies ist zum einen für die Anwendung von SMD erforderlich, da dort der Datentyp `FAD` verwendet wird und zum anderen unterstützen nicht alle CUDA-Hardware-Versionen eine native 64 Bit Gleitkomma-Arithmetik. Daher werden sog. *Funktions-Templates* [66] verwendet, wodurch eine generische Implementierung von Algorithmen ohne Festlegung eines konkreten Datentyps ermöglicht wird. Unterstützt die verwendete CUDA-Hardware also keine 64 Bit Arithmetik, so wird der 32 Bit Datentyp `float` verwendet – andernfalls der 64 Bit Datentyp `double`. Im Folgenden werden die einzelnen Klassen erläutert, dabei werden nur die wichtigsten Funktionen erwähnt.

Optimizer ist die Oberklasse aller Optimierungsalgorithmen. Eine Klasse die die Schnittstelle `Optimizer` implementiert, muss im Kontruktor ein Objekt vom Typ `Optimizable` übergeben bekommen und mindestens die Methode `iterateSingle()` anbieten. Ein Aufruf dieser Methode soll eine Iteration des entsprechenden Optimierungsalgorithmus ausführen. In dem CRF Framework wird diese Schnittstelle von den Klassen `SGD` und `SMD` implementiert, welche die aus dem vorherigen Abschnitt bekannten Methoden für das Parameterupdate enthalten. Hierbei ist zu beachten, dass die Optimierungsalgorithmen vollkommen generisch implementiert wurden. Diese sind nicht auf CRFs oder die Likelihood spezialisiert und können auch zur Optimierung jeder anderen Zielfunktion eingesetzt werden.

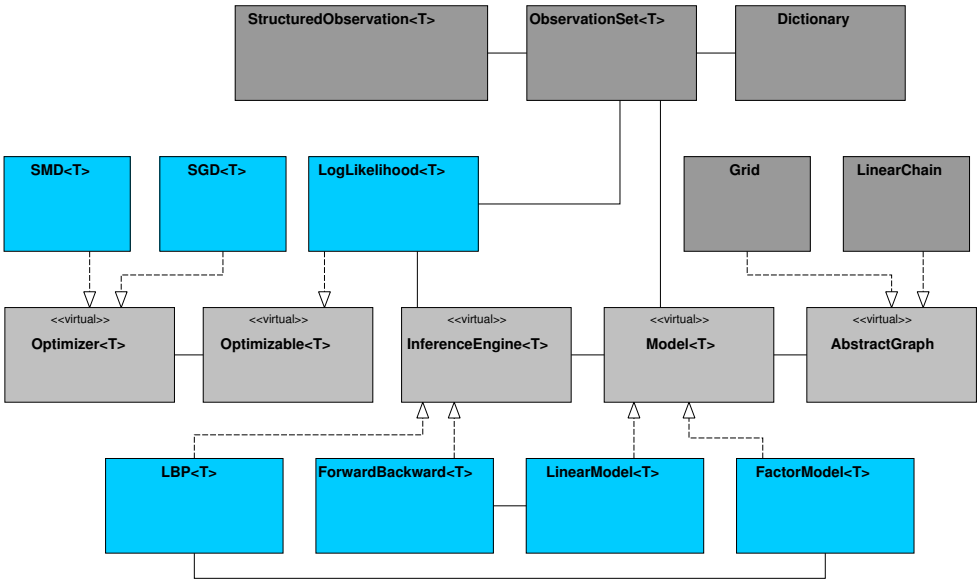


Abbildung 5.6: Klassenhierarchie der Programmbibliothek. Die gestrichelten Pfeile signalisieren eine Vererbungsrelation und die ungerichteten Kanten eine Assoziation [52]. Alle grau eingefärbten Klassen enthalten keinen GPGPU Code. Hellgraue Klassen sind abstrakt und können nicht instanziiert werden. Die blauen und dunkelgrauen Klassen enthalten die eigentlichen Algorithmen und Datenstrukturen.

Optimizable ist die von einer Zielfunktionen zu implementierende Schnittstelle. Jede von `Optimizable` erbende Klasse muss mindestens die beiden Methoden `compute()` sowie `computeGradient()` anbieten, welche den Zielfunktionswert und den Gradienten berechnen. Die einzige Klasse des Frameworks, die diese Schnittstelle implementiert, ist die Klasse `LogLikelihood`. Diese enthält die aus Abs. 5.4.4 bekannte Methode `berechneGradient()`. Zur Berechnung des Funktionswerts wird eine Summenreduktion durchgeführt.

InferenceEngine ist die Schnittstelle des Frameworks zu Implementierungen von Inferenzalgorithmen. Eine solche Klasse enthält mindestens die Methoden `computeMarginals(..)`, `computeMaxPropAssignment(..)` sowie `getExpectations()`. Die Methoden `computeMarginals(..)` und `computeMaxPropAssignment(..)` erwarten ein Objekt von Typ `ObservationSet`. Die Schnittstelle wird von den Klassen `LBP` sowie `ForwardBackward` implementiert. Dort sind die Methoden zur Berechnung der Randverteilungen und Modellerwartungen sowie zur Klassifikation untergebracht.

AbstractGraph ist die Oberklasse graphischer Strukturen. Eine solche Klasse muss sicherstellen, dass beim Konstruktoraufruf die in Abschnitt 5.3.4 erläuterten Datenstrukturen zur Repräsentation von Faktorgraphen erzeugt werden. Diese Schnittstelle wird von den Klassen `Linear-Chain` und `Grid` verwendet, welche die aus Abschnitt 3.4 bekannten graphischen Strukturen erzeugen.

Model ist die abstrakte Elternklasse jedes Modells und enthält alle Datenstrukturen zur Repräsentation der Modellparameter sowie der Trainingsdaten. Da beim Erzeugen des Modells ausgenutzt werden kann, dass die graphische Struktur bekannt ist, wurde eine separate Implementierung für den Forward-Backward Algorithmus geschrieben (`LinearModel`). Die Klasse `FactorModel` enthält die entsprechenden Daten bei der Anwendung von BP und LBP. In `Model`-Klassen findet ebenfalls das zentralisierte Kopieren der Daten in den globalen Speicher der GPU statt. Um den Parametervektor zu erzeugen, bzw. zu erweitern, sowie das Kopieren der Daten zu

veranlassen, muss die Methode `init(..)` aufgerufen werden. Diese erwartet ein Objekt vom Typ `ObservationSet`. Um im Kernelcode auf alle Datenstrukturen zugreifen zu können, wurde eine Konfigurationsstruktur verwendet, die ausschließlich aus Zeigern auf die Datenstrukturen besteht. Diese wird vor jeder Iteration aktualisiert und in den Konstanten-Speicher der GPU kopiert.

Klassen zur Datenhaltung. Die Klasse `ObservationSet` ist für das Einlesen von Trainings- bzw. Testmengen sowie deren Repräsentation im Hauptspeicher der CPU zuständig. Das Einlesen erfolgt durch die Methoden `readFromFile(..)` oder `readFromStream(..)`, wobei entweder ein Dateiname oder ein C++ Datenstrom (`IStream`) als Argument erwartet wird. Ein einzelnes Beispiel wird durch ein Objekt vom Typ `StructuredObservation` dargestellt. Die Wertebereiche $\mathcal{X}$ und $\mathcal{Y}$ werden mit Instanzen des Typ `Dictionary` repräsentiert, mit dessen Hilfe die eingelesenen Zeichenketten auf ihre eindeutigen Identifikationsnummern abgebildet werden.

Algorithmus 24 Einfaches Linear-Chain CRF mit SGD Optimierung. Als Batchgröße wurde $|\mathcal{B}| = 8$ und als Schrittweite $\eta = 10^{-2}$ gewählt. Jedes Beispiel der Trainingsdaten darf maximal 100 versteckte Knoten enthalten und insgesamt dürfen 32 verschiedene Label in den Trainingsdaten vorkommen.

```
 1: #include ‘‘crfgpu.h’’
 2:
 3: int main(int argc, char** args){
 4:   AbstractGraph*          G = new LinearChain(100);
 5:   LinearModel<float>*     M = new LinearModel<float>(G, 32);
 6:   InferenceEngine<float>* P = new ForwardBackward<float>(M);
 7:   ObservationSet<float>*  B = new ObservationSet<float>(M);
 8:   LogLikelihood<float>*   L = new LogLikelihood<float>(P, B);
 9:   Optimizer<float>*       O = new SGD(0.01, L);
10:
11:   while(!std::cin.eof()){
12:     B->readFromStream(&std::cin, 8);
13:     M->init(B);
14:     O->iterateSingle();
15:     M->clear();
16:     B->clear();
17:   }
18:
19:   M->saveToFile(‘‘model.dat’’);
20:   return 0;
21: }
```

Anwendung der Bibliothek Die Programmbibliothek kann nun verwendet werden, um die hier implementierten Algorithmen und Datenstrukturen in ein lauffähiges Programm einzubetten. Beispielsweise zeigt Algorithmus 24 den vollständigen Quellcode der erforderlich ist, um ein Linear-Chain CRF mit SGD Optimierung zu trainieren. In der ersten Zeile werden alle erforderlichen Klassen eingebunden. In den Zeilen 4-9 werden Instanzen der benötigten Klassen erzeugt. In der Schleife (Zeile 11-17) werden die Beispiele aus dem Eingabedatenstrom (`std::cin`) gelesen. Die Daten des Stroms müssen das in Abs. 5.3.1 vorgestellte Format besitzen. In Zeile 12 werden jeweils acht Beispiele aus dem Datenstrom gelesen. Der Parametervektor wird in Zeile 13 erzeugt. Zeile 14 stößt die Berechnung der Randverteilungen und Erwartungen an, berechnet anschließend

den Gradient und führt ein SGD Parameterupdate durch. Anschließend werden alle den Batch betreffenden Daten aus dem Speicher entfernt (Zeilen 15 und 16). Lediglich die Modellparameter bleiben nach dem Aufruf der `clear()` Methoden erhalten. Man beachte, dass das Training nach dem Einlesen der ersten $b = 8$ Beispiele sofort beginnt. Sobald keine Daten mehr über den Eingabestrom ankommen wird das gelernte Modell in die Datei `model.dat` geschrieben und das Programm terminiert.

Algorithmus 24 zeigt selbstverständlich ein Minimalbeispiel. Mit Hilfe des Frameworks wurde eine Programm mit dem Arbeitstitel CRF^{GPU} implementiert. Dieses ist auf der CD-ROM in Anhang D enthalten. Alle Kommandozeilenparameter des Programms sind in Anhang A aufgelistet. Obwohl das Programm viele Einstellungsmöglichkeiten besitzt, kann ein einfaches Linear-Chain CRF Training mit Forward-Backward Inferenz mit dem Befehl

```
crfgpu < train.dat
```

ausgeführt werden. Abschließend wird die Laufzeit in Sekunden sowie die Accuracy auf der Trainingsmenge ausgegeben. Soll stattdessen die Güte auf einer Testmenge berechnet werden, so muss diese mit dem Parameter `-T` übergeben werden. Sollen Precision, Recall und F_1-Score ebenfalls ausgegeben werden, so muss die Option `--prf` verwendet werden. Um anstatt der SGD Optimierung den SMD Algorithmus zu verwenden, wird der Parameter `-O3` übergeben. Unter Berücksichtigung dieser Optionen ergibt sich der Befehl:

```
crfgpu -T test.dat --prf -O3 < train.dat
```

Welche Optionen für die Experimente verwendet wurden, wird im entsprechenden Kapitel erläutert.

Das Programm besitzt allerdings auch Einschränkungen. Beispielsweise sind bisher nur zwei graphische Strukturen (Linear-Chain und Grid) implementiert. Wird eine andere Struktur benötigt, so muss diese auf Basis der Klasse `AbstractGraph` implementiert werden. Aus Abschnitt 5.4.1 ist bekannt, dass für den Forward-Backward Algorithmus maximal 64 Label verwendet werden können. Diese Schranke wurde ebenfalls für BP und LBP übernommen, da die Primzahlfunktion `nextPrime()` nicht implementiert wurde. Stattdessen wird zur Kodierung von Labelmengen die Primzahl 67 und zur Kodierung von Knotenmengen die 2063 verwendet, wodurch die Länge des längsten Beispiels ebenfalls einschränkt wurde. Werden diese oberen Schranken überschritten, so wird die Terminierung des Programms erzwungen. Eine Einschränkung, die sich aus den beiden `Model` Implementierungen ergibt, ist, dass der Viterbi Algorithmus nur in Kombination mit der Klasse `LinearModel` und der MBK Algorithmus nur in Verbindung mit dem `FactorModel` genutzt werden kann.

Des Weiteren bestehen diverse Möglichkeiten, um die Anzahlen der verwendeten Blöcke und Threads zu ändern (s. Anhang A). Alle Anzahlen von Worker-Blöcken, die nicht bereits in den obigen Abschnitten erläutert wurden, werden mit 32 initialisiert. Es zeigte sich während der Implementierung, dass diese Größe sowohl für die Hardware-Version 1.3 als auch für 2.0 stets gute Ergebnisse im Bezug auf die Laufzeit liefert. Da die optimale Anzahl allerdings von der konkreten Hardware abhängig ist, können diese Werte auch geändert werden.

Im folgenden Kapitel werden nun die Laufzeiten und Klassifikationsgüten von CRF^{GPU} mit den zu Anfang des Kapitels als Referenz ausgewählten Implementierungen verglichen.

6 Evaluation

In diesem Kapitel werden Versuche mit der im vorherigen Kapitel entworfenen parallelen CRF Implementierung durchgeführt. Zur Einordnung der Ergebnisse werden die Experimente ebenfalls auf den beiden Referenzimplementierungen CRF++ sowie CRFSGD durchgeführt. Die Versuche sollen nicht zeigen, dass mit den hier verwendeten Methoden und Merkmalen die bestmöglichen F_1-Scores auf den jeweiligen Datensätzen erzielt werden können, sondern Hinweise darauf geben,

- ob die Klassifikationsgüte der hier entwickelten Implementierung mit denen der Referenzimplementierungen vergleichbar ist,
- wie sich die Laufzeiten der parallelen Algorithmen bei unterschiedlichen Batchgrößen verhalten,
- und ob das Training von CRFs durch die Verwendung von GPGPU beschleunigt werden kann.

Die Laufzeiten der hier entwickelten Implementierung werden ebenfalls für die Klassifikation gemessen. Erste Versuche zeigten bereits, das die Laufzeit der hier entwickelten LBP Implementierung nicht praxistauglich ist. Diese benötigte über 30 Stunden für Versuche, die der parallele FB Algorithmus in unter 5 Minuten absolvierte. Allerdings zeigte sich ebenfalls, dass die Güte von BP und LBP in allen Testläufen identisch waren. Aufgrund der unverhältnismäßig hohen Laufzeit sowie der Redundanz der Ergebnisse wurde hier auf die Darstellung der LBP Ergebnisse verzichtet. Auf die Resultate einzelner LBP Läufe wird in der Diskussion der Ergebnisse an entsprechender Stelle hingewisen. Alle Versuche wurden auf der Linear-Chain Struktur durchgeführt.

In den folgenden Abschnitten werden die in den Versuchen verwendete Soft- und Hardware sowie die Datensätze vorgestellt. In Abschnitt 6.4 werden die Versuche erläutert und die Ergebnisse in Form von Abbildungen und Tabellen präsentiert. Im letzten Abschnitt erfolgt eine ausführliche Diskussion der einzelnen Versuchsreihen.

6.1 Software

Hier wird erläutert welche Version des jeweiligen Programmes genutzt und welche Einstellungen verwendet wurden. Die genutzten Versionen der Programme befinden sich auf der CD-ROM im Anhang D.

Von CRF++ wurde die aktuelle[14] Version 0.54 genutzt. Diese wurde von dem in Abs. 5.1 genannten Internetauftritt heruntergeladen. Zum Training wurde das Programm mit dem folgenden Kommando gestartet:

```
crf_learn -m 50 -p 4 -a CRF VORVERARBEITUNG TRAININGSDATEN MODELL
```

Dabei legt die Option "`-m 50`" die Anzahl zu absolvierender Iterationen auf 50 fest. Durch die Option "`-p 4`" werden vier Threads verwendet und mit "`-a CRF`" wird die L-BFGS Optimierung aktiviert. Die Datei `VORVERARBEITUNG` enthält die in Abs. 5.3.1 erläuterte Anweisungen zur Generierung der jeweiligen Merkmale. Dort ist beschrieben, wie aus den entsprechenden Rohdaten die Realisationen der beobachteten Knoten zu erzeugen sind. Auf eine Beschreibung der Syntax dieser Vorverarbeitungsdatei wurde hier verzichtet, da die in den Versuchen genutzten Dateien auf

[14] Stand: März 2011.

der CD-ROM im Anhang D enthalten sind. Die `TRAININGSDATEN` haben das aus dem vorherigen Kapitel bekannte (Tab. 5.1) Format. Das fertige Modell, bestehend aus dem Parametervektor, den Mengen $\mathcal{X}$ und $\mathcal{Y}$, sowie der Vorverarbeitungsdatei, wird in die Datei `MODELL` geschrieben. Um das Modell zur Klassifikation der Testdaten einzusetzen, wird der folgende Aufruf verwendet:

```
crf_test -m MODELL TESTDATEN
```

Die `TESTDATEN` müssen dieselbe Form wie die `TRAININGSDATEN` besitzen. Die Ausgabe erfolgt als Liste von Labeln. Enthielten die `TESTDATEN` bereits Label, so werden beide nebeneinander in der Form "`REFERENZLABEL VORHERSAGE`" ausgegeben.

Von CRFSGD wurde die aktuelle Version 1.3 verwendet, welche ebenfalls von dem entsprechenden Internetauftritt heruntergeladen wurde. Das Training wurde in den Versuchen mit dem Befehl:

```
crfsgd -f 1 -r 50 -h 50 MODELL VORVERARBEITUNG TRAININGSDATEN
```

gestartet. Die Option "`-f 1`" bedeutet, dass jeder Parameter, der mindestens einmal in den Trainingsdaten vorkommt, erzeugt werden soll. Trotzdem schien CRFSGD in den Versuchen auch die Parameter der unbekannten Realisationen zu erzeugen. Dies wurde daraus abgeleitet, dass die Anzahl erzeugter Parameter den Faktor ≈ 10 größer war als bei CRF^{GPU}. Die Optionen "`-r 50 -h 50`" veranlassen die Ausführung von 50 Iterationen mit anschließender Berechnung des Trainingsfehlers. Ohne die Option "`-h 50`" wird der Trainingsfehler alle 10 Iterationen berechnet, was die Laufzeitergebnisse zu ungunsten von CRFSGD ändern würde. Das Format der Eingabedaten sowie der Vorverarbeitung ist identisch zu dem von CRF++. Obwohl CRFSGD einen stochastischen Gradientenabstieg durchführt, können weder Batchgröße noch Schrittweite eingestellt werden. Die Batchgröße ist konstant 1 und die Schrittweite wird initial mit einer Line-Search bestimmt. Soll das Programm zur Klassifikation eingesetzt werden, so ist der Aufruf ähnlich zu dem von `crf_test`, mit dem Unterschied, dass anstelle von "`-m`" die Option "`-t`" verwendet werden muss. CRFSGD besitzt dasselbe Ausgabeformat für klassifizierte Beispiele wie CRF++.

Die verwendete Version der hier entwickelten Implementierung CRF^{GPU} wurde mit dem Kommando:

```
crfgpu -T TESTDATEN -I 50 -b |B| -Y |Y| -M n_max -A num -O num --eta 0.01
       --lambda 0.75 --mu 0.01 --iob --prf -L AUSGABE < TRAININGSDATEN
```

gestartet. Die `TEST-` und `TRAINGSDATEN` müssen das in Abs. 5.3.1 erläuterte Format besitzen. Diese Dateien werden, wie dort erwähnt, mit Hilfe von Python-Skripten erzeugt. Da der Code der Skripte ausschließlich aus der Manipulation von Zeichenketten besteht, wird er hier nicht weiter erläutert. Er befindet sich ebenfalls in Anhang D. Die durchgeführten Vorverarbeitungen werden im Abschnitt über die Datensätze beschrieben. Die Option "`-I 50`" veranlasst die Ausführung von 50 Iterationen und "`-b` $|\mathcal{B}|$" die Verwendung der Batchgröße $|\mathcal{B}|$. Die Optionen "`-Y`" und "`-M`" legen obere Schranken für die Anzahl der Klassen und Knoten fest. Diese Werte sind zwar nach vollständigem Einlesen der Trainingsdaten bekannt, da das Training aber sofort beginnt, nachdem die ersten $|\mathcal{B}|$ Beispiele eingelesen wurden, werden diese oberen Schranken benötigt um genug Speicher für die Datenstrukturen zu reservieren. Diese Werte müssen nicht angegeben werden. In diesem Fall werden sie automatisch auf $|\mathcal{Y}| = 32$ und $n_{max} = 2048$ festgelegt. Eine Angabe führt allerdings zu einem geringeren Speicherverbrauch. Der zu verwendende Inferenzalgorithmus wird mit "`-A`" ausgewählt, wobei die Nummer `num` des entsprechenden Algorithmus (s. Anhang A) angegeben werden muss. In den Experimenten wurde "`-A 1`" für den Forward-Backward Algorithmus sowie "`-A 2`" für die serielle Belief-Propagation verwendet. Die Wahl des Optimierungsalgorithmus verläuft analog. Hier wurde "`-O 1`" für SGD und "`-O 3`" für SMD verwendet. Die Schrittweite "`--eta`",

die Meta-Schrittweite "`--mu`" sowie die Gewichtung der SMD Gradienten-Historie "`--lambda`" werden auf allen Datensätzen oben dargestellten Konstanten verwendet. Sie orientieren sich an den in [72] vorgeschlagenen Werten. Bei der Verwendung von SGD werden die, nur für SMD relevanten, Parameter μ und λ ignoriert. Die Optionen "`--iob`" und "`--prf`" veranlassen die Berechnung von Precision, Recall und F_1-Score auf den `TESTDATEN`. Die vorhergesagten Label werden in die Datei `AUSGABE` geschrieben. Diese Datei besitzt dasselbe Format wie bei CRF++ und CRFSGD.

6.2 Hardware

Für die Versuche stand ein *GPU-System* mit CUDA-Hardware-Version 2.0 zur Verfügung. Mit einer Ausnahme wurden alle Ergebnisse auf einem Rechner mit AMD Phenom II X6 1090T Prozessor, 8 GiB Hauptspeicher sowie einer NVidia Tesla C2050 GPU durchgeführt. Die GPU verfügt über 14 Multiprozessoren mit jeweils 32 Kernen, 48 KiB gemeinsamen Speicher pro MP und 3 GiB globalen Speicher. Da der Hauptspeicher dieses Systems für die CRF++ Versuche auf dem zweiten Datensatz (s.u.) nicht ausreichte, wurden diese stattdessen auf einem *Ersatz-System* mit AMD Opteron 2220 Prozessor sowie 30 GiB Hauptspeicher durchgeführt. In Anbetracht der geringen Größe des globalen Speichers und der Tatsache, dass CRF++ in einigen Versuchen mehr als 20 GiB Hauptspeicher benötigte, wird anschaulich klar, warum das Auffinden einer modifizierbaren Implementierung im vorherigen Kapitel scheiterte.

6.3 Datensätze

In diesem Abschnitt werden die drei zur Evaluation verwendeten Datensätze vorgestellt. Zu jedem Datensatz wird kurz die konkrete Lernaufgabe erläutert, die Häufigkeitsverteilung der jeweiligen Klassen betrachtet sowie die verwendeten Merkmale erklärt. Die Datensätze sowie die verwendeten Vorverarbeitungsskripte sind auf der CD-ROM in Anhang D enthalten.

6.3.1 CoNLL-2000

Die ersten Versuche werden auf der *gemeinsamen Aufgabestellung* (*shared task*) des *Workshop on Computational Natural Language Learning* (CoNLL) aus dem Jahr 2000 durchgeführt. Solche Aufgabenstellungen können im Rahmen eines Workshops von den Teilnehmern innerhalb eines festgelegten Zeitraums bearbeitet werden. Den Teilnehmern wird eine Trainingsmenge zur Verfügung gestellt, auf deren Basis sie ihre Lösungsansätze entwickeln können. Nach Ablauf der Bearbeitungszeit wird die Güte der einzelnen Ansätze auf einer vorher unbekannten Testmenge miteinander verglichen. Im Anschluss werden häufig die Trainings- und Testdaten öffentlich zugänglich gemacht, um die Reproduktion der Ergebnisse zu vereinfachen oder neue Verfahren mit den bekannten Ansätzen zu vergleichen. Beim damaligen Workshop[15] hat das beste Verfahren einen F_1-Score von $93,48\%$ erreicht.

Obgleich diese Daten über 10 Jahre alt sind, stellen die meisten der in Kapitel 5 vorgestellten Implementierungen auf ihrem Internetauftritt eine Anleitung zur Bearbeitung des CoNLL-2000 Datensatzes bereit. Er wurde für die Versuche in diesem Kapitel verwendet, da auf diese Weise eine Einordnung der Güte sowie der Laufzeit von CRF^{GPU} in ein breites Feld von anderen CRF Implementierungen möglich ist. Des Weiteren wurde dieser Datensatz auch in [72] für CRF Versuche mit SGD Optimierung verwendet.

[15] http://www.clips.ua.ac.be/conll2000/chunking. Stand: März 2011.

Not	all	of	NASA	's	science	work	will	be	so	auspicious
RB	DT	IN	NNP	POS	NN	NN	MD	VB	RB	JJ
B-NP	I-NP	B-PP	B-NP	B-NP	I-NP	I-NP	B-VP	I-VP	B-ADJP	I-ADJP

Tabelle 6.1: Ausschnitt aus dem CoNLL-2000 Trainingsdatensatz. Die erste Zeile enthält die strukturierte Beobachtung, die zweite POS-Tag und die dritte die Label.

Label	NP	VP	PP	ADVP	SBAR	ADJP	PRT	CONJP	INTJ	LST	UCP
Train.	51%	20%	20%	4%	2%	2%	1%	0% (56)	0% (31)	0% (10)	0% (2)
Test.	52%	20%	20%	4%	2%	1%	1%	0% (9)	0% (2)	0% (5)	0% (0)

Tabelle 6.2: Häufigkeiten der Wortgruppen im CoNLL-2000 Trainings- und Testdatensatz. Die Trainingsmenge enthält insgesamt 106978 Wortgruppen. Die Testmenge 23852 Gruppen. Alle Werte sind gerundet. Bei 0,00% ist die absolute Häufigkeit in Klammern mit angegeben.

Die Aufgabe beim CoNLL-2000 shared task bestand darin, Sätze in sich nicht überlappende, syntaktisch in Beziehung stehende Wortgruppen zu segmentieren. [68]. Die aufzuspürenden *Wortgruppen* (*Phrases*) sind *Adjective Phrase* (ADJP), *Adverb Phrase* (ADVP), *Noun Phrase* (NP), *Verb Phrase* (VP), *Interjection* (INTJ), *List marker* (LST), *Conjunction Phrase* (CONJ), *Prepositional Phrase* (PP), *Particle* (PRT) *Unlike Coordinated Phrase* (UCP), sowie *Complementizer* (SBAR). Die Bezeichnungen sind aus [68] übernommen. Dort ist eine ausführliche Einführung in diesen Datensatz zu finden. Die soeben aufgezählten Wortgruppen entsprechen den Labeln des strukturellen Klassifikationsproblems, wobei das aus Kapitel 2 bekannte IOB-Tagging verwendet wird um die Zugehörigkeit zu einer Wortgruppe zu markieren. Zusätzlich wird angenommen, dass jedes Wort bereits mit seiner *Wortart* (*Part-of-speech*, POS) versehen wurde. Eine Übersicht über die Wortgruppen und Wortarten ist in [43] zu finden.

Ein Ausschnitt aus dem CoNLL-2000 Trainingsdatensatz ist in Tabelle 6.1 dargestellt. Die Trainingsmenge besteht aus 8936 englischsprachigen Sätzen, die Testmenge aus 2012. In Tabelle 6.2 ist die Häufigkeitsverteilung der einzelnen Wortgruppen zu sehen. Obwohl diese Darstellung auch in [68] verwendet wird, verschleiert sie einen Teil der Komplexität dieser Lernaufgabe. Die Häufigkeitsverteilung der entsprechenden IOB-Tags ist in Tab. 6.3 aufgetragen. Dort wird deutlich, dass die `O` Klasse (keine Gruppenzugehörigkeit) 13,17% aller Worte ausmacht. Damit ist `O` das vierthäufigste Label, obwohl es in Tab. 6.1 nicht vorkommt. Außerdem fällt auf, dass das Label `I-LST` nur in der Testmenge vorkommt. Da diese allerdings zur Entwicklungszeit der Methoden nicht bekannt war, wird diese auch hier nicht mit in die Labelmenge $\mathcal{Y}$ aufgenommen. Darin sind

Label	B-NP	B-VP	B-PP	B-ADVP	B-SBAR	B-ADJP	B-PRT
Train.	26,01	10,13	10,05	1,99	1,04	0,97	0,26
Test.	26,21	9,83	10,15	1,82	1,12	0,92	0,22

O	I-NP	I-VP	I-PP	I-ADVP	I-SBAR	I-ADJP	I-PRT
13,17	29,90	5,66	0,13	0,20	0 (70)	0,30	0 (2)
13,04	30,34	5,58	0,10	0,18	0 (4)	0,35	0 (0)

B-INTJ	B-LST	B-UCP	B-CONJP	I-INTJ	I-LST	I-UCP	I-CONJP
0 (31)	0 (10)	0 (2)	0 (56)	0 (9)	0 (0)	0 (6)	0 (73)
0 (2)	0 (5)	0 (0)	0 (9)	0 (0)	0 (2)	0 (0)	0 (13)

Tabelle 6.3: Häufigkeiten der Label im CoNLL-2000 Trainings- und Testdatensatz. Die Trainingsmenge enthält insgesamt 211727 Worte. Die Testmenge 47377 Worte. Alle Angaben sind in Prozent. Bei 0,00% ist die absolute Häufigkeit in Klammern angegeben.

Label	DNA	RNA	cell_line	cell_type	protein
Train.	18,58	1,85	7,46	13,09	59,00
Test.	12,19	2,36	5,77	22,17	58,49

Tabelle 6.4: Häufigkeiten der molekularbiologischen Entitäten im JNLPBA-2004 Trainings- und Testdatensatz. Die Trainingsmenge enthält insgesamt 51301 Entitäten, die Testmenge 8662.

ausschließlich die $|\mathcal{Y}| = 22$ Label der Trainingsmenge enthalten.

Im Abschnitt 5.3.1 wurde angedeutet, dass zur Segmentierung von Texten häufig Kombinationen aus aufeinanderfolgenden Worten als Merkmale bzw. Realisationen der beobachteten Knoten gewählt werden. Wie oben erwähnt, werden hier ebenfalls die POS zur Erzeugung der Merkmale verwendet. Ist w_t das Wort an Satzposition t sowie pos_t der entsprechende Part-of-speech Tag, so erhält jeder Faktorknoten eines Linear-Chain CRF die folgenden beobachteten Nachbarknoten:

w_{t-2}, w_{t-1}, w_t, w_{t+1}, w_{t+1}, w_{t-1}/w_t, w_t/w_{t+1}, pos_{t-2}, pos_{t-1}, pos_t, pos_{t+1}, pos_{t+2}, pos_{t-2}/pos_{t-1}

pos_{t-1}/pos_t, pos_t/pos_{t+1}, pos_{t+1}/pos_{t+2}, $pos_{t-2}/pos_{t-1}/pos_t$, $pos_{t-1}/pos_t/pos_{t+1}$, $pos_t/pos_{t+1}/pos_{t+2}$

Diese werden hier auch *Wort-* bzw. *POS-Merkmale* genannt. Ein Schrägstrich bedeutet dabei, dass die Zeichenketten einfach hintereinander geschrieben werden, um die Realisation eines beobachteten Knotens zu erzeugen. Ist das Wort an Position t beispielsweise "`all`" gefolgt von dem Wort "`of`", so wird der beobachtete Knoten w_t/w_{t+1} die Realisation "`all/of`" haben. Die entsprechenden Realisationen werden an allen Faktorknoten f_t sowie $f_{\{t-1,t\}}$ des Linear-Chain CRFs erzeugt. Durch das Hinzufügen der Präfixe (s. Abs. 5.3.1) wird sichergestellt, dass beispielsweise die Realisationen der Knoten w_{t-1} und w_t nicht verwechselt werden können. Die Wahrscheinlichkeit, dass ein Wort zu einer bestimmten Gruppe gehört, ist nach diesem Modell von den beiden vorangegangenen und den beiden folgenden Worten sowie deren Part-of-Speech Tags abhängig. Auf diese Weise ergeben sich $|\mathcal{X}| = 338547$ verschiedene Realisationen aller beobachteten Knoten. Der aus dieser Trainingsmenge von CRF^{GPU} erzeugte Parametervektor enthält $|\boldsymbol{\theta}| = 1039407$ Gewichte. Mit diesen Merkmalen wird in [72] für SGD von einem F_1-Score in Höhe von $93,46\%$ und für SMD in Höhe von $93,6\%$ berichtet. Ein ähnlicher Merkmalssatz wurde erstmalig von Sha und Pereira [62] verwendet.

6.3.2 JNLPBA-2004

Die zweite Versuchsreihe wird auf den Daten der gemeinsamen Aufgabenstellung *des Joint workshop on natural language processing in biomedicine and its applications* (JNLPBA) aus dem Jahr 2004 durchgeführt. Dort bestand die Aufgabe darin, technische Ausdrücke der Molekularbiologie in Fachtexten zu identifizieren und in eine von fünf Klassen einzuordnen. Auch dort wurde das IOB-Tagging verwendet, um zusammenhängende Worte einer Klasse zu kennzeichnen. Die automatische Erkennung der Semantik bestimmter Wortgruppen oder Worten innerhalb von Texten wird als *Named Entity Recognition* (NER) bezeichnet. In der Aufgabenstellung des JNLPBA-2004 sollten die Entitäten *DNA*, *R*NA, *Cell Line*, *Cell Type* sowie *Protein* erkannt werden. Eine ausführlichere Beschreibung der Daten lässt sich in [61] finden. Dort wurden ebenfalls CRFs eingesetzt und ein F_1-Score von $69,8\%$ erreicht.

Die Trainingsmenge besteht aus 18546 englischsprachigen Sätzen, die Testmenge aus 3856. In Tabelle 6.4 ist die Häufigkeitsverteilung der einzelnen Entitäten zu sehen. Tabelle 6.5 zeigt die Häufigkeitsverteilung der entsprechenden IOB-Tags. Dort ist gut zu erkennen, dass die Klasse `O` mit großem Abstand die häufigste ist. In dieser Lernaufgabe kommt es also primär darauf an, die wenigen interessanten Entitäten von dem irrelevanten Rest zu unterscheiden. Alle Label kommen

Label	B-DNA	B-RNA	B-cell_line	B-cell_type	B-protein
Train.	1,93	0,19	0,77	1,36	6,14
Test.	1,04	0,11	0,49	1,90	5,01

O	I-DNA	I-RNA	I-cell_line	I-cell_type	I-protein
77,75	3,2	0,31	1,49	1,77	5,04
80,80	1,77	0,18	0,97	2,96	4,72

Tabelle 6.5: Häufigkeiten der Label im JNLPBA-2004 Trainings- und Testdatensatz. Die Trainingsmenge enthält insgesamt 492551 Worten. Die Testmenge 101039 Worten. Alle Angaben sind in Prozent.

sowohl in der Trainings- als auch in der Testmenge vor. Jeder versteckte Knoten des Linear-Chain CRF kann sich damit zu einem von $|\mathcal{Y}| = 11$ möglichen Labeln realisieren.

Auch für diesen Datensatz werden die Wort-Merkmale w_{t-2}, w_{t-1}, w_t, w_{t+1}, w_{t+2}, w_{t-1}/w_t, w_t/w_{t+1}, $w_{t-2}/w_{t-1}/w_t$, $w_{t-1}/w_t/w_{t+1}$, $w_t/w_{t+1}/w_{t+2}$ verwendet. Allerdings stehen bei dieser Lernaufgabe keine Part-of-speech Tags o.ä. zur Verfügung. Settles [61] bemerkte, dass Fachbegriffe der Molekularbiologie häufig bestimmte Buchstabenkombinationen, Ziffern, Sonderzeichen oder die Namen griechischer Buchstaben enthalten. Daher schlug er sog. *orthographische Merkmale* für diese Aufgabenstellung vor. Diese entsprechen binären Indikatorfunktionen die genau dann 1 (wahr) sind, wenn das Wort w eine bestimmte Eigenschaft hat, z.B. wenn es eine Ziffer enthält.

Da in [61] nicht angegeben wurde, wie genau diese Merkmale zu erzeugen sind, wurden hier die 18 regulären Ausdrücke (s. Anhang B) von Bundschus et al. [10] verwendet. Um CRF^{GPU} auch auf sehr großen Merkmalsmengen zu testen, wurde jedes der o.g. Merkmale für jede der Realisationen w_{t-2}, w_{t-1}, w_t, w_{t+1}, w_{t+2}, w_{t-1}/w_t, w_t/w_{t+1} an jedem Faktorknoten ausgewertet. Das heißt, für jedes orthographische Merkmal o_k erhält jeder Faktorknoten 7 beobachtete Nachbarn, nämlich $o_k(w_{t-2})$, $o_k(w_{t-1})$, $o_k(w_t)$, $o_k(w_{t+1})$, $o_k(w_{t+2})$, $o_k(w_{t-1})/o_k(w_t)$ und $o_k(w_t)/o_k(w_{t+1})$. Jeder dieser Knoten hat allerdings nur zwei mögliche Realisationen, ‘‘`wahr`’’ oder ‘‘`falsch`”. Zusammen mit den zehn o.g. Wort-Merkmalen ergeben sich insgesamt 139 beobachtete Nachbarn pro Faktorknoten sowie $|\mathcal{X}| = 1362222$ möglichen Realisationen.

Allerdings wurde oben erwähnt, dass die Klasse `O` besonders häufig in den Daten vorkommt. Da die orthographischen Merkmale aber darauf ausgelegt sind, molekularbiologische Entitäten zu erkennen, werden diese für die meisten anderen Worte die Realisation ‘‘`falsch`” haben. Da das Eingabeformat von CRF^{GPU} es erlaubt, dass jeder Faktorknoten zu einer unterschiedlichen Anzahl beobachteter Knoten adjazent sein kann, wurden bei der Vorverarbeitung nur diejenigen Realisationen erzeugt in denen mindestens ein ‘‘`wahr`”-Wert vorkommt. Bei CRF++ und CRFSGD war dies nicht möglich. Dort mussten auch sämtliche Realisationen erzeugt werden, da dort jede Zeile die gleiche Anzahl an Merkmalen enthalten muss (s. Abs. 5.3.1). Das von CRF^{GPU} erzeugte Linear-Chain CRF enthält $|\boldsymbol{\theta}| = 2973640$ Parameter.

6.3.3 PREFETCH

Der dritte Datensatz stammt aus einer von Fricke et al. [23] durchgeführten Versuchsreihe zum Server-basierten Lernen von Dateizugriffsmustern mobiler Endgeräte. Zu diesem Zweck wurden *Betriebssystemaufrufe* (engl. *System Calls*) eines laufenden Systems aufgezeichnet und die darin enthaltenen Dateizugriffe mit den Klassen `FULL`, `READ` und `ZERO` annotiert. Dateizugriffe eines Prozesses wurden zu einem Beispiel zusammengefasst. Der Datensatz wird ausführlich in [23] erläutert. Dort wurde mit CRF++ ein F_1-Score von 96.93% erreicht. Da die zuverlässige Vorhersage von Dateizugriffen ein Vorausladen von benötigten Daten erlauben würde, wird dieser Datensatz hier als PREFETCH bezeichnet. Er wird hier als Beispiel für ein online Lernszenario verwendet.

Label	FULL	READ	ZERO
Train.	22,56	37,14	40,28
Test.	22,58	38,32	39,09

Tabelle 6.6: Häufigkeiten der Label im PREFETCH Trainings- und Testdatensatz. Die Trainingsmenge enthält insgesamt 605241 dateibezogene Systemaufrufe. Die Testmenge 68646 Worte. Alle Angaben sind in Prozent.

FB	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
D1	44,38	31,92	23,85	18,63	15,3	13,36	12,26	11,47	10,6	10,1	9,93
	9,32	6,55	4,54	3,21	2,37	1,85	1,48	1,27	1,1	1,02	0,95
D2	159,7	93,50	58,67	40,16	30,64	25,74	23,02	21,99	21,09	20,54	20,21
	39,33	23,17	13,42	7,96	5,05	3,52	2,65	2,19	1,92	1,74	1,7
D3	16,23	9,66	6,51	4,85	4,01	3,59	3,39	3,31	3,12	3,04	3,03
	1,62	0,9	0,52	0,36	0,27	0,22	0,19	0,17	0,17	0,15	0,17
(L)BP											
D1	86,38	61,04	43,69	32,62	25,05	19,79	16,78	14,62	12,92	12,21	11,13
	21,87	15,32	10,36	7,12	5,02	3,5	2,59	1,94	1,53	1,27	1,07
D2	232,9	133,2	81,20	53,42	38,88	31,07	27,11	25,09	23,08	22,45	21,69
	59,27	34,37	19,84	11,52	6,97	4,56	3,29	2,59	2,08	1,9	1,69
D3	24,92	13,9	8,64	5,92	4,48	3,76	3,43	3,11	2,96	2,8	2,9
	2,49	1,33	0,75	0,46	0,32	0,24	0,19	0,19	0,18	0,16	0,17

Tabelle 6.7: Für das Einlesen der Datensätze benötigte Sekunden bei verschiedenen Batchgrößen. BP und LBP benötigen etwas mehr Zeit, da sie die graphische Struktur berücksichtigen müssen. D1: CoNLL-2000. D2: JNLPBA-2004. D3: PREFETCH.

Die Trainingsmenge enthält 72530 Beispiele und die Testmenge 8131 unklassifizierte Instanzen. In diesem Datensatz wird kein IOB-Tagging verwendet, daher sind genau die $|\mathcal{Y}| = 3$ o.g. Klassen in der Labelmenge enthalten, deren Häufigkeitsverteilung in Tabelle 6.6 dargestellt ist. Die Merkmale werden aus Eigenschaften der System Calls erzeugt. Es sei $file_t$ die Datei auf die zum Zeitpunkt t vom Programm $exec_t$ zugegriffen wird. Daraus wurden die beiden Merkmale $file_{t-2}/file_{t-1}/file_t$ sowie $exec_{t-1}/exec_t$ erzeugt. Mit diesen Merkmalen sind in der Trainingsmenge insgesamt $|\mathcal{X}| = 5006$ Realisationen der beobachteten Knoten enthalten. Das von CRF^{GPU} erzeugte Modell besitzt $|\boldsymbol{\theta}| = 10801$ Parameter.

6.4 Versuche

Dieser Abschnitt enthält die Beschreibung sowie die Ergebnisse der durchgeführten Versuche. Wie bereits zu Anfang dieses Kapitels erwähnt, werden die aus Kapitel 2 bekannten Gütemaße sowie die Laufzeit auf den oben erläuterten Datensätzen gemessen. Zur Identifikation der Ergebnisse wird das verwendete Optimierungsverfahren jeweils im Subskript des Inferenzalgorithmus angegeben, z.B. FB_{SGD}. Im Text werden dafür die Ausdrücke *Variante* und *Kombination* verwendet.

In den Tabellen 6.8, 6.9 und 6.10 sind die benötigte Trainingszeit, Accuracy, Precision (**P**), Recall (**R**) und F_1-Score (F_1) der Implementierungen CRFSGD, CRF++ und CRF^{GPU} auf den Testmengen der Datensätze dargestellt. Die Werte beziehen sich für FB_{SGD}, FB_{SMD} sowie BP_{SGD} auf eine Batchgröße von $|\mathcal{B}| = 256$ und bei der Kombination BP_{SMD} auf die Batchgröße $|\mathcal{B}| = 8$. Die Verfahren wurden auf den ersten beiden Datensätzen für jeweils 50 Iterationen ausgeführt. Der PREFETCH Datensatz wurde als Beispiel für ein online Training verwendet. Dort wurde die Klassifikationsgüte bereits nach einer Iteration gemessen. Die jeweils geringste Laufzeit bzw. die höchste Güte einer jeden Versuchsreihe wurde fettgedruckt gesetzt.

Die beiden ersten Datensätze enthalten Skripte zur Berechnung von Precision, Recall und F_1-Score des IOB-Taggings. Obwohl beide dasselbe Eingabeformat akzeptieren, liefern sie auf denselben Daten unterschiedliche F_1-Scores mit Schwankungen um $\approx 0,5\%$. Zur Berechnung der Werte in den Tabellen wurden die jeweiligen Skripte verwendet. Auf dem dritten Datensatz wurde die Güte mit CRF^{GPU} berechnet, wobei die jeweilige Gesamtgüte als Mittelwert der Einzelgütemaße bestimmt wurde. Die Zeiten wurden jeweils inklusive des Einlesens der Daten gemessen um einen realisatischen Eindruck der Laufzeit des Programms zu erhalten.

Die Abbildungen 6.1 und 6.3 zeigen die Entwicklung des F_1-Scores der einzelnen Algorithmenkombinationen (links) sowie bei verschiedenen Batchgrößen (rechts) über 50 Iterationen. In der jeweils linken Abbildung wurden dieselben Batchgrößen wie in den Tabellen verwendet (s.o.). Da LBP dieselben F_1-Scores wie BP liefert, wären die entsprechenden Kurven mit denen von BP identisch. Für die ersten beiden Datensätze ist auf der jeweils rechten Seite der Abbildung der Verlauf der F_1-Scores der Kombination FB_{SGD} für die Batchgrößen 8, 32, 128 und 512 zu sehen. Diese Plots sollen prototypisch verdeutlichen, welchen Einfluss eine Änderung der Batchgröße auf den Verlauf der Optimierung hat. Zur Generierung dieser Grafiken wurden auf allen Datensätzen die von GPU^{CRF} berechneten Gütmaße verwendet. Die Werte entsprechen denen des CoNLL-2000 Skripts zur Berechnung der Güte.

Die Abbildungen 6.2, 6.4 und 6.6 zeigen die Entwicklung der Trainingszeit einzelner Verfahren bei Variation der Batchgröße. Die jeweils linke Abbildung zeigt einen Gesamtüberblick und die auf der jeweils rechten Seite eine vergrößerte Darstellung der Laufzeiten der FB Varianten. Die Trainingszeiten beziehen sich auf eine Iteration über die gesamte Trainingsmenge inklusive des Einlesens der Daten. Die Versuche wurden für die Batchgrößen $|\mathcal{B}| \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ durchgeführt. Da es bei diesen Versuchen mit der BP_{SMD} Variante zu Systeminstabilitäten ab einer Batchgröße von 16 kam, wurde diese Kombination hier ausgelassen. Die o.g. Abbildungen sowie Abbildung 6.5 zeigen die soeben erläuterten Versuche auch für die Klassifikationszeit anstelle der Trainingszeit. Die Klassifikationszeiten sind in den Abbildungen ohne die zum Einlesen der Daten benötigte Zeit dargestellt. Diese kann Tabelle 6.7 entnommen werden. Das *Einlesen* umfasst dabei nicht nur den reinen Datentransfer, sondern ebenfalls das Erzeugen der Datenstrukturen sowie das Kopieren dieser in den globalen Speicher der GPU. Die Einlesezeiten beim FB Algorithmus sind etwas geringer als die von BP/LBP, da die graphische Struktur beim Einlesen mitberücksichtigt werden muss und diese beim FB Algorithmus fest ist. Im Folgenden werden die hier erzielten Ergebnisse diskutiert.

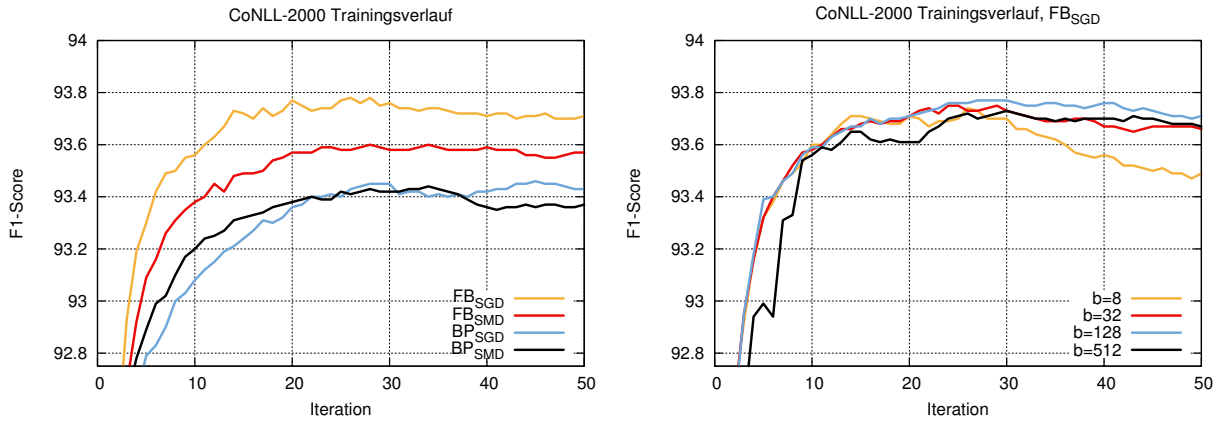


Abbildung 6.1: Entwicklung des F_1-Scores auf dem CoNLL-2000 Testdatensatz. Links: Verschiedene Inferenz- und Optimierungsverfahren. Rechts: Trainingsverlauf von FB_{SGD} bei verschiedenen Batchgrößen.

	CRFSGD			CRF++			CRF^{GPU}, FB_{SGD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	**93,91**	93,62	**93,77**	93,73	**93,72**	93,72	93,78	93,64	93,71
ADJP	79,70	**73,52**	**76,48**	77,56	72,60	75,00	**80,00**	72,15	75,87
ADVP	82,33	80,72	81,52	**82,86**	**80,95**	**81,89**	81,00	80,72	80,86
CONJP	62,50	**55,56**	58,82	55,56	**55,56**	55,56	57,14	44,44	50,00
INTJ	**100,0**	**50,00**	**66,67**	**100,0**	**50,00**	**66,67**	**100,0**	**50,00**	**66,67**
NP	**94,42**	**94,06**	**94,24**	94,03	94,12	94,08	94,24	94,03	94,13
PP	96,83	97,73	97,28	**96,86**	**98,07**	**97,46**	96,82	97,96	97,39
PRT	80,21	72,64	76,24	**82,11**	**73,58**	**77,61**	77,78	72,64	75,12
SBAR	88,61	84,30	86,40	**89,76**	85,23	**87,44**	88,78	**85,79**	87,26
VP	93,78	**94,25**	**94,01**	93,78	94,12	93,95	**93,86**	94,14	94,00
Accuracy		**96,08**			96,03			96,03	
Laufzeit		1034,83			6097,85			**222,1**	

	CRF^{GPU}, BP_{SGD}			CRF^{GPU}, FB_{SMD}			CRF^{GPU}, BP_{SMD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	93,54	93,31	93,43	93,62	93,51	93,57	93.5	93.25	93.37
ADJP	77,97	71,92	74,82	79,75	71,92	75,63	78.73	71,00	74.67
ADVP	80,42	79,21	79,81	81,59	80,83	81,21	81.88	79.33	80.59
CONJP	**71,43**	**55,56**	**62,50**	66,67	44,44	53,33	62,50	**55,56**	58,82
INTJ	**100,0**	**50,00**	**66,67**	**100,0**	**50,00**	**66,67**	**100,0**	**50,00**	**66,67**
NP	94,03	93,72	93,88	94,03	93,91	93,97	93.86	93.71	93.79
PP	96,69	97,88	97,28	96,67	97,90	97,28	96.71	97.88	97.29
PRT	**82,11**	**73,58**	**77,61**	79,17	71,70	75,25	81.25	73.58	77.23
SBAR	88,47	83,18	85,74	88,91	85,42	87,13	88.82	83.18	85.91
VP	93,52	93,95	93,73	93,56	93,95	93,75	93.31	93.71	93.51
Accuracy		95,91			95,94			95,85	
Laufzeit		2578			240,3			31860	

Tabelle 6.8: Klassifikationsgüte und Trainingszeit auf dem CoNLL2000 Datensatz nach jeweils 50 Iterationen. Die Laufzeiten sind in Sekunden angegeben, die übrigen Werte in Prozent.

6.5 Diskussion

In diesem Abschnitt werden die erzielten Ergebnisse diskutiert. Dazu werden die einzelnen Versuche der Reihe nach abgehandelt.

CoNLL-2000. Auf dem CoNLL-2000 Datensatz (Tab. 6.8) sind die Klassifikationsgüten aller Verfahren sehr ähnlich. Die Standardabweichung der Gesamt-F_1-Scores beträgt $0,14\%$. Die Güte der Vorhersagequalität von CRF^{GPU} kann auf diesem Datensatz als gleichwertig zu denen der Referenzimplementierungen eingestuft werden. Bei Betrachtung von Abb. 6.1 fällt auf, dass FB_{SGD} zwischen der 20. und 30. Iteration in der Nähe einer besseren Lösung war. Die von CRF^{GPU} erzeugten Logdateien (s. Anhang D) zeigen, dass der F_1-Score in der 28. Iteration sein Maximum von $73,8\%$ Prozent erreichte und dies bereits nach $128,1$ Sekunden. Dass dieser danach wieder abnimmt, kann als eine Überanpassung an die Trainingsmenge verstanden werden.

Die BP Varianten sind den FB Varianten auf diesem Datensatz unterlegen, was sowohl die Gesamtgüte als auch die Laufzeit betrifft. Der Unterschied in der Laufzeit kann mit der höheren asymptotischen Laufzeitkomplexität von BP erklärt werden. Die Unterschiede in der Güte ergeben sich aus der Verwendung eines anderen Klassifikationsalgorithmus. Auf dem Datensatz scheinen Sequenz-

informationen von besonderer Bedeutung zu sein, da die Verfahren, die den Viterbi Algorithmus zur Klassifikation einsetzen, die besten Ergebnisse erzielen. Lediglich auf den Klassen `CONJP` und `PRT` wurden mit der Kombination BP_{SGD} die besten Ergebnisse erzielt. Überraschend ist hier das "schlechte" Abschneiden der SMD Optimierung, deren Güte bei beiden Inferenzalgorithmen nicht an diejenige der SGD Optimierung herankommt. Vishwanathan et al. berichten in [72] von einem von SMD erreichten F_1-Score in Höhe von $93,6\%$ sowie $93,4\%$ bei Verwendung von SGD, allerdings mit der Konfiguration $\mu = \eta = 10^{-1}$ sowie $\lambda = 1$. Mit diesen Einstellungen wurde von CRF^{GPU} mit FB_{SGD} ein F_1-Score in Höhe von $93,27\%$ und von FB_{SMD} $92,2\%$ erreicht. In beiden Fällen kann das Ergebnis aus [72] also nicht bestätigt werden.

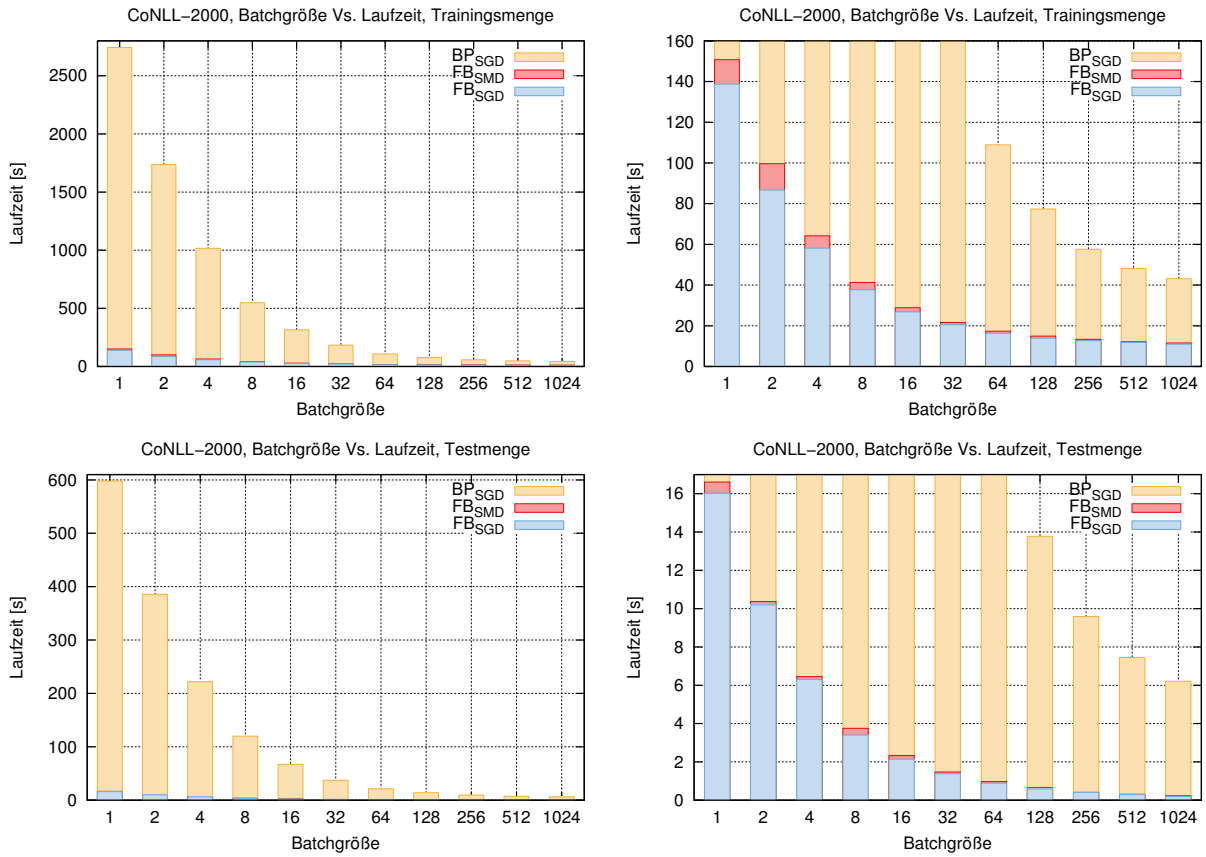


Abbildung 6.2: Entwicklung der Laufzeit einer Trainingsiteration (oben) sowie der Klassifikation (unten) bei zunehmender Batchgröße auf dem CoNLL-2000 Datensatz. Links: Gesamtübersicht. Rechts: Laufzeiten der FB Varianten vergrößert.

Die Laufzeiten von CRF^{GPU} sind, zumindest für die Batchgröße 256, geringer als diejenigen der Referenzimplementierungen. Selbst die BP Variante mit SGD Optimierung braucht fast eine Stunde weniger als CRF++. Die sehr Hohe Laufzeit der BP_{SMD} Variante ist auf die geringe Batchgröße von 8 zurückzuführen. Die Auswirkungen einer Veränderung der Batchgröße auf die Trainingszeit sind in Abbildung 6.2 dargestellt. Im Falle kleiner Batchgrößen fällt die benötigte Laufzeit um ca. 33% durch eine Verdopplung der Batchgröße, und damit einer Verdopplung der zu nutzenden Thread-Blöcke. Dasselbe gilt für die zur Klassifikation benötigte Laufzeit. Dies ist ein Indiz dafür, dass die hier entwickelte Parallelisierung insofern funktioniert, als dass die Hinzunahme zusätzlicher Recheneinheiten die Laufzeit verkürzt. Da die Anzahl tatsächlich verfügbarer Kerne konstant ist (in diesem Fall 484), wird die durch die Parallelisierung erreichte Verbesserung stetig geringer. Ab einer Batchgröße von 256 verändern sich die Laufzeiten nur noch marginal. Die von CRFSGD zur Klassifikation benötigte Zeit betrug 2,28 und die von CRF++ 2,31 Sekunden. Aus Abb. 6.2 ist zu erkennen, dass die FB Varianten die Klassifikationszeiten der

Referenzimplementierungen ab einer Batchgröße von 32 unterbieten. Die Klassifikationszeiten von BP – und damit die von LBP ebenfalls – sind als nicht praxistauglich zu bezeichnen. Die hier nicht dargestellte Trainingszeit von LBP ist ≈ 10 mal höher als die von BP, wobei die resultierenden F_1-Scores identisch sind. Ein Versuch mit Factorie zeigte, dass das dort verwendete Sampling für 10 Iterationen über 400 Sekunden benötigt und einen F_1-Score von unter 90% erzielt.

	CRFSGD			CRF++			CRF^{GPU}, FB_{SGD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	**68,83**	49,11	57,32	38,46	20,09	26,39	64,41	65,33	64,87
`DNA`	**79,41**	43,47	56,18	3,12	31,13	5,68	64,19	62,97	**63,58**
`RNA`	**75,00**	50,85	60,61	0,00	0,00	0,00	60,81	38,14	46,88
`cell_line`	**57,77**	39,40	46,85	7,20	10,17	8,43	50,00	39,00	43,97
`cell_type`	**79,48**	38,31	51,70	6,51	44,96	11,37	69,72	**56,69**	62,53
`protein`	**65,85**	55,30	60,12	40,83	30,51	34,93	64,00	72,33	67,91
Accuracy	89,29			82,59			91,17		
Laufzeit	2918,55			7909,58			**364,9**		

	CRF^{GPU}, BP_{SGD}			CRF^{GPU}, FB_{SMD}			CRF^{GPU}, BP_{SMD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	62,07	**66,57**	64,24	64,96	64,92	64,94	64,50	67,42	**65,93**
`DNA`	63,81	61,27	62,51	61,19	**63,16**	62,16	66,67	60,04	63,18
`RNA`	53,85	59,32	56,45	62,03	41,53	49,75	62,61	**61,02**	**61,80**
`cell_line`	42,53	**55,20**	48,04	49,87	38,00	43,13	42,59	**55,20**	**48,08**
`cell_type`	68,71	56,48	62,00	70,44	56,43	62,66	74,01	56,48	**64,07**
`protein`	62,33	72,78	67,15	65,26	71,70	68,33	64,23	**74,46**	**68,97**
Accuracy	**91,72**			90,97			91,07		
Laufzeit	4063			388,5			73130		

Tabelle 6.9: Klassifikationsgüte und Trainingszeit auf dem JNLPBA-2004 Datensatz nach jeweils 50 Iterationen. Die Laufzeiten sind in Sekunden angegeben, die übrigen Werte in Prozent.

JNLPBA-2004. Auf dem JNLPBA-2004 Datensatz (Tab. 6.9) übertrifft CRF^{GPU} beide Referenzimplementierungen sowohl hinsichtlich der Klassifikationsgüte als auch der benötigten Laufzeit. Nach Ansicht des Autors ist dies damit zu erklären, dass CRF++ und CRFSGD die Parameter unbekannter Realisationen erzeugen. In Kombination mit der hohen Anzahl hier verwendeter Merkmale resultiert dies in einem hochgradig überangepassten Modell. Dies lässt sich auch an den hohen Genauigkeiten der CRFSGD Vorhersagen erkennen. Falls CRFSGD eine von `O` verschiedene Klasse vorhersagt, ist diese mit hoher Wahrscheinlichkeit richtig, allerdings wird meist die Klasse `O` vorhergesagt, was dazu führt, dass der Recall und auch die F_1-Scores aller Klassen entsprechend niedrig sind. Eine Auswertung des CRFSGD Modells nach 40 Iterationen zeigte, dass die Optimierung offenbar um die Optimalstelle oszilliert. Dabei konnten verschiedene F_1-Scores beobachtet werden, wobei der beste Parametervektor eine Precision von 62,50%, einen Recall von 64,79% und einen F_1-Score von 63,62% besaß. Dieses Verhalten war bei einigen Läufen von CRF^{GPU} ebenfalls zu erkennen. In Abbildung 6.3 weisen sowohl die großen Zacken in den Trainingsverläufen beider SGD Varianten (linker Plot), als auch bei der Batchgröße 512 (rechter Plot) auf eine starke Osszilation hin. Dieses Verhalten ist in der Regel auf eine zu groß gewählte Schrittweite zurückzuführen (s. Abs. 3.8.1). Bei CRFSGD kann dies nicht vermieden werden, da dort die Schrittweite automatisch bestimmt wird und nicht manuell einstellbar ist. Bei den SMD Varianten existiert dieses Problem nicht (Abb. 6.3, links). Dort sorgt die automatische Schrittweitenanpassung für eine "glatte" Konvergenz zum erwarteten Optimum. Dies ist auch der Grund dafür, warum diese Versuchsreihe von den beiden SMD Varianten dominiert wird. Diese erreichen

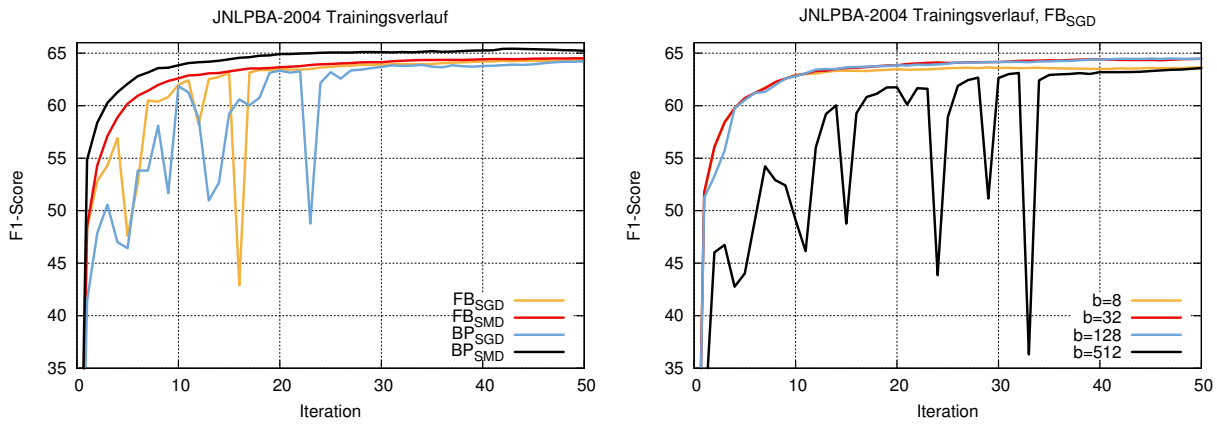


Abbildung 6.3: Entwicklung des F_1-Scores auf dem JNLPBA-2004 Testdatensatz. Links: Verschiedene Inferenz- und Optimierungsverfahren. Rechts: Trainingsverlauf von FB_{SGD} bei verschiedenen Batchgrößen.

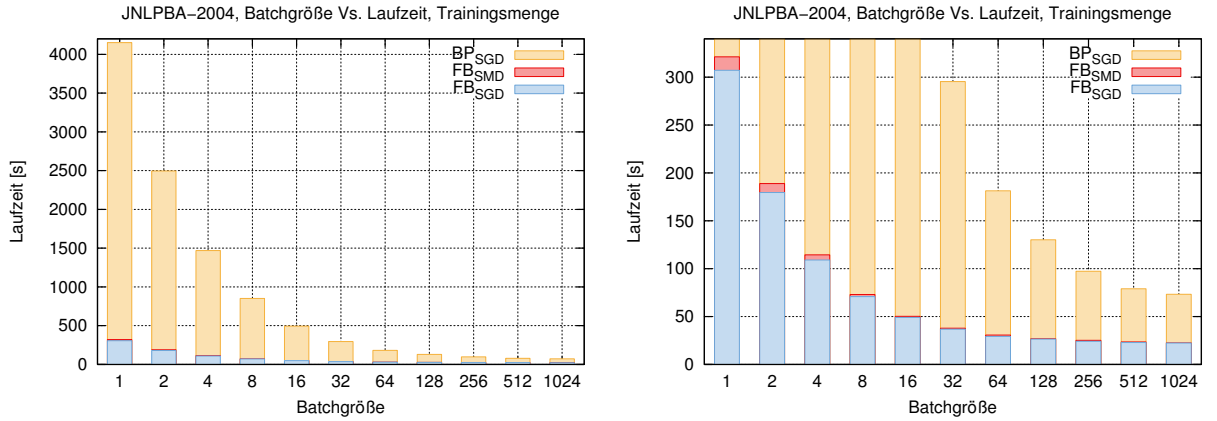


Abbildung 6.4: Entwicklung der Laufzeit einer Trainingsiteration bei zunehmender Batchgröße auf dem JNLPBA-2004 Trainingsdatensatz. Links: Gesamtübersicht. Rechts: Laufzeiten der FB Varianten vergrößert.

den höchsten sowie den zweithöchsten Gesamt-F_1-Score. Die Kombination BP_{SMD} erzielte, mit Ausnahme der Klasse `DNA`, die höchsten F_1-Scores in allen Klassen. Der Recall ist bei den Klassen `DNA`, `RNA` und `cell_line` allerdings um mehr als 10% schlechter als derjenige von CRFSGD. Die SGD Läufe von CRF^{GPU} ordnen sich zwischen CRFSGD und den SMD Läufen ein. Unter Berücksichtigung der (o.g.) Oszillation sind die Lösungen von CRFSGD und CRF^{GPU} insgesamt als gleichwertig anzusehen, wobei die SMD Schrittweitensteuerung klar überlegen ist.

Die sehr geringen F_1-Scores von CFR++ sind in diesem Ausmaß nur auf die Komplexität des Optimierungsproblems zurückzuführen. Während die beiden anderen Implementierungen eine stochastische Optimierung durchführen, wird bei CRF++ L-BFGS eingesetzt. In jeder Iteration muss also jedes Gewicht des Parametervektors auf Basis der kompletten Trainingsmenge aktualisiert werden. Laut der Bildschirmausgabe von CRF++ wurden insgesamt 175688865 Parameter erzeugt. Mit jedem zusätzlichen Parameter wird das Auffinden einer "guten" Suchrichtung schwerer. In Kombination mit dem oben erwähnten Problem der Überanpassung bei Berücksichtigung unbekannter Realisationen scheint CRF++ nicht in der Lage zu sein in die Nähe des Optimums zu gelangen. Auch in Versuchen, bei denen nur Merkmale verwendet wurden, die mindestens drei Mal in den Trainingsdaten vorkommen, konnten keine wesentliche Verbesserung erzielt werden.

Die beste Gesamtlaufzeit wurde erneut von der Kombination FB_{SGD} erzielt. Die Kombination BP_{SGD} ist zumindest schneller als CRF++, wobei hier berücksichtigt werden muss, dass die

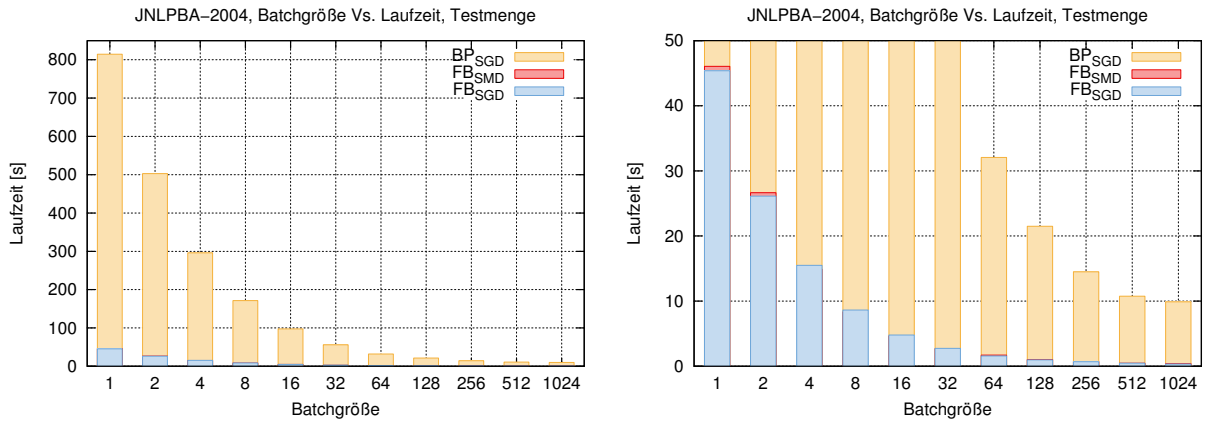


Abbildung 6.5: Entwicklung der zur Klassifikation benötigten Laufzeit bei zunehmender Batchgröße auf dem JNLPBA-2004 Testdatensatz. Links: Gesamtübersicht. Rechts: Laufzeiten der FB Varianten vergrößert.

CRF++ Versuche für diesen Datensatz auf einer anderen Hardware durchgeführt wurden, da der Hauptspeicher des GPU-Systems nicht ausreichte. Zur Einordnung der Laufzeit wurde der JNLPBA-2004 Versuch ebenfalls mit CRFSGD auf dem Ersatz-System durchgeführt. Dort wurden $5087,52$ Sekunden benötigt, also ≈ 2000 Sekunden mehr als auf dem GPU-System. Aufgrund des effizienteren Designs von CRFSGD im Vergleich mit CRF++ wird nicht vermutet, dass CRF++ mit der CPU des GPU-Systems ebenfalls ≈ 2000 Sekunden schneller gewesen wäre. Allerdings hätte in diesem Fall auch ein Vorsprung von 2000 Sekunden für CRF++ nicht gereicht, um BP_{SGD} zu überholen.

Auch bei den JNLPBA-2004 Versuchen hängt sowohl die Laufzeit des Trainings (Abb. 6.4) als auch die der Klassifikation (Abb. 6.5) stark von der gewählten Batchgröße ab. Wie bereits bei den CoNLL-2000 Versuchen nimmt der Geschwindigkeitszuwachs stetig ab. Die Referenzimplementierungen benötigten 6,19 (CRFSGD) sowie 8,57 Sekunden (CRF++) zur Klassifikation der Testmenge. Hier waren die FB Varianten ab einer Batchgröße von 16 schneller. Die Klassifikationszeiten der BP und LBP Varianten sind auch hier wesentlich höher als die der reinen Linear-Chain Implementierungen.

Abschließend sei hier betont, dass der verwendete Merkmalsatz gewählt wurde, um zu zeigen, dass die hier entwickelte Implementierung dazu in der Lage ist, auch bei vielen Merkmalen praxistaugliche Laufzeiten und Klassifikationsgüten zu erzielen. Motiviert wurde dies durch eine Aussage von Sutton [67], nach dem die Stärke von CRFs gerade in der Verwendung zahlreicher, untereinander abhängiger Merkmale liegt.

PREFETCH. Die letzte Versuchsreihe zeigt, dass die GPGPU-Implementierung in Bezug auf die Laufzeit den Referenzimplementierungen nicht uneingeschränkt überlegen ist. Da dieser Datensatz aus einem Strom von Systemaufrufen erzeugt wurde, wurde er hier als online Lernaufgabe betrachtet. Aufgrund der geringen Anzahl an Klassen (3) und beobachteten Merkmalsausprägungen (5006), besitzt das resultierende Modell im Gegensatz zu den beiden anderen Datensätzen nur wenige Parameter. In Tabelle 6.10 ist zu sehen, dass beide Referenzimplementierungen den Lauf über die Trainingsdaten eine bis zwei Sekunden schneller absolvieren als die FB Varianten von CRF^{GPU}. Hierbei ist allerdings zu beachten, dass CRFSGD und CRF++ keinen Datenstrom verarbeiten. Sie lesen die Trainingsdatei einmal vollständig ein und führen anschließend die Iteration durch. Da CRF^{GPU} bereits mit dem Training beginnt, sobald die ersten $|\mathcal{B}|$ Beispiele eingetroffen sind, entsteht ein Mehraufwand durch das wiederholte Erzeugen der Datenstrukturen.

Die Laufzeiten der BP Varianten sind wesentlich höher. Diese scheinen stark von der Anzahl zu

	CRFSGD			CRF++			CRF^{GPU}, FB_{SGD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	**96,24**	**95,64**	**95,93**	83,23	76,36	79,64	83,96	80,51	82,19
FULL	98,79	94,85	**96,87**	98,42	66,55	79,40	95,70	95,07	95,38
READ	**95,61**	**94,42**	**95,01**	67,79	88,96	76,94	65,28	92,83	76,66
ZERO	**94,34**	**97,67**	**95,97**	83,50	73,57	78,22	90,92	53,65	67,48
Accuracy	**95,79**			77,88			78,02		
Laufzeit	**5,90**			5,10			6,89		

	CRF^{GPU}, BP_{SGD}			CRF^{GPU}, FB_{SMD}			CRF^{GPU}, BP_{SMD}		
	P	**R**	F_1	**P**	**R**	F_1	**P**	**R**	F_1
Gesamt	94,36	92,55	93,44	84,07	80,69	82,34	94,86	93,99	94,42
FULL	**99,14**	87,42	92,91	95,84	**95,50**	95,67	98,03	92,29	95,07
READ	92,28	93,20	92,74	65,42	92,91	76,78	93,81	93,41	93,61
ZERO	91,67	97,04	94,28	90,96	53,67	67,51	92,76	96,27	94,48
Accuracy	93,4			78,15			94,28		
Laufzeit	2347			7,23			4416		

Tabelle 6.10: Klassifikationsgüte und Trainingszeit auf dem PREFTECH Datensatz nach einer Iteration. Die Laufzeiten sind in Sekunden angegeben, die übrigen Werte in Prozent.

verarbeitender Beispiele abhängig zu sein, da die anderen o.g. Größen allesamt geringer als bei den beiden vorherigen Datensätzen sind. Eine genauere Betrachtung des PREFETCH Datensatzes zeigt allerdings, dass hier noch ein zweiter, bisher unbeachteter Wert heraussticht. Während die maximale Anzahl versteckter Knoten beim CoNLL-2000 Datensatz 78 und bei den JNLPBA-2004 Daten 204 beträgt, besteht hier das längste Beispiel aus 1420 versteckten Knoten. Beim FB Algorithmus geschieht die Iteration über diese Knoten innerhalb des Kernels. Bei den Varianten der Belief-Propagation berechnet ein Kernel lediglich die ausgehenden Nachrichten eines Knotens, wodurch wesentlich mehr parallele Kernelaufrufe stattfinden. Hier zeigt sich, dass diese Designentscheidung zwar zu einer sehr hohen Parallelität führt, diese aber die Fähigkeiten der hier verwendeten Hardware anscheinend übersteigt. Wie in den Abbildungen 6.4 und 6.5 zu erkennen ist, fällt die Laufzeit von BP zwar Anfangs mit steigender Batchgröße, allerdings beginnt sie ab einer Batchgröße von ≈ 128 wieder zu steigen. Von diesem Punkt an, ist die GPU überlastet. Dass die Laufzeit der seriellen Belief-Propagation mit Batchgröße 256 leicht unter der der Batchgrößen 128 und 512 liegt, ist hier kein Versehen. Dieses Verhalten von BP konnte in allen Versuchen auf dem PREFETCH Datensatz beobachtet werden. Bei LBP ist der Grad an Parallelität um den Faktor $|F|$ größer als bei BP, da dort die Nachrichten an allen Faktorknoten parallel berechnet werden. LBP Experimente deren Ergebnisse hier nicht grafisch aufbereitet wurden, zeigten, dass die Laufzeit von LBP bereits ab einer Batchgröße von 2 ansteigt. Die GPU ist dort also bereits mit den Berechnungen eines Beispiels vollständig ausgelastet.

Auf diesem Datensatz benötigte CRFSGD lediglich 0,2 Sekunden zur Klassifikation und CRF++ nur 0,14. Daher sind hier die FB Varianten erst ab einer Batchgröße von 512 schneller. Die BP/LBP Varianten sind, wie auch auf den anderen Datensätzen, mehr als eine Größenordnung langsamer als die Linear-Chain Implementierungen. Die Güten der BP Varianten sind mit denen von CRFSGD vergleichbar, wobei letztere ein insgesamt besseres Resultat liefert (s. Tab. 6.10). Die FB Varianten sowie CRF++ erreichen diese Güte nicht, sind aber untereinander ähnlich. Bei Versuchen mit 10 und 20 Iterationen fiel allerdings auf, dass sich die Güten aller Verfahren kaum verbesserten. Einzig CRF++ erreichte eine ähnlich optimale Lösung wie CRFSGD und die BP Varianten.

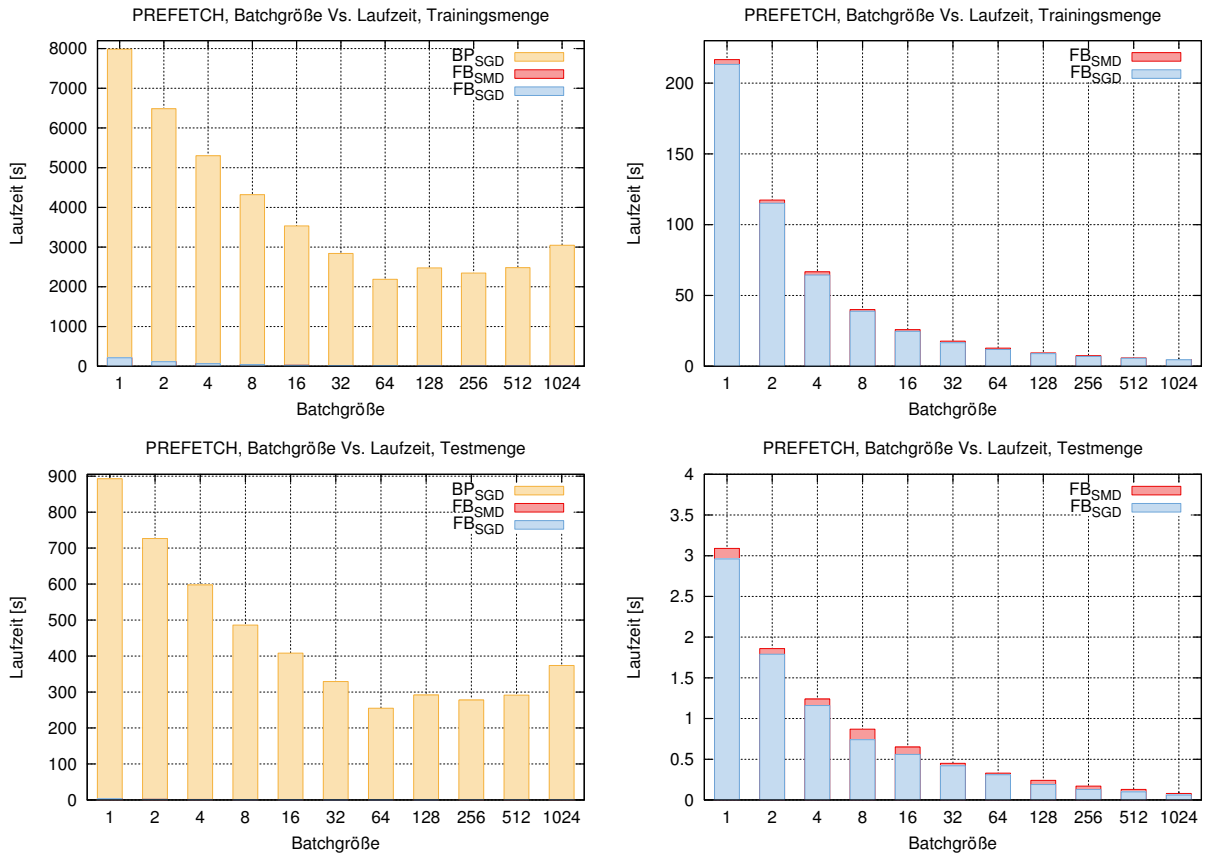


Abbildung 6.6: Entwicklung der Laufzeit einer Trainingsiteration (oben) sowie der Klassifikation (unten) bei zunehmender Batchgröße auf dem PREFETCH Datensatz. Links: Gesamtübersicht. Rechts: Laufzeiten der FB Varianten.

Gesamtbetrachtung. Die zu Anfang dieses Kapitels gestellte Frage, ob die von CRF^{GPU} erzielte Güte mit der der Referenzimplementierungen vergleichbar ist, kann im Rückblick auf die Ergebnisse mit "ja" beantwortet werden. Auch von den Resultaten, die von Vishwanathan et al. [72], Settles [61] und Fricke et al. [23] auf den jeweiligen Datensätzen erzielt wurden, weicht die hier erreichte Klassifikationsgüte nur unwesentlich ab.

Die Batchgrößen haben offensichtlich einen starken Einfluss auf die Laufzeit der Implementierung. Durch eine Erhöhung der Batchgröße konnte auf allen Datensätzen eine Beschleunigung erreicht werden. Ausgenommen davon sind die LBP Varianten, deren hohe Parallelität bereits bei einer Batchgröße von 1 die GPU voll auslasten. Ebenso konnte auch bei BP beobachtet werden, dass die Laufzeit steigt, sobald die Batchgröße zu hoch gewählt wurde.

Zumindest für Linear-Chain CRFs konnte gezeigt werden, dass das Training von CRFs durch die Verwendung von GPGPU beschleunigt werden kann. Ist die Anzahl der verwendeten Merkmale gering, so scheint die Verwendung paralleler Algorithmen nicht sinnvoll bzw. notwendig zu sein, da hier die CPU Implementierungen schneller waren. Steigt die Anzahl der Merkmale, so ändert sich dies. Auf dem CoNLL-2000 Datensatz konnte das Training im Vergleich mit CRFSGD um den Faktor 4,65 und mit CRF++ um den Faktor 27,45 beschleunigt werden. In den JNLPBA-2004 Versuchen war CRF^{GPU} 7,99 Mal schneller als CRFSGD sowie 21,6 Mal schneller als CRF++.

Darüberhinaus zeigte sich, dass die Optimierung mit SMD nur unwesentlich langsamer als die mit SGD ist. Allerdings kommt es anscheinend auf den Datensatz an, ob die mit SMD erzielte Klassifikationsgüte diejenige von SGD übertrifft oder nicht.

7 Zusammenfassung und Ausblick

Es folgt ein Rückblick auf diese Arbeit sowie ein anschließender Ausblick auf zukünftige Möglichkeiten bezüglich der Parallelisierung von CRFs sowie der Anwendung der hier entwickelten GPGPU-Programmbibliothek.

In dieser Arbeit wurde aufbauend auf einer Einführung in die Grundbegriffe des Maschinellen Lernens (Kap. 2) die Theorie der Conditional Random Fields erläutert (Kap. 3). Dabei wurde der Unterschied zwischen generativen und diskriminativen Modellen betrachtet und so die Verwendung von CRFs zur Lösung des Klassifikationsproblems mit strukturiertem Ausgaberaum motiviert. Es wurden verschiedene Inferenz-, Klassifikations- und Optimierungsalgorithmen vorgestellt, wobei stets versucht wurde, auf parallelisierungsrelevante Eigenschaften zu achten.

Im 4. Kapitel wurde die grundsätzliche Idee paralleler Berechnungsmodelle vermittelt, und die Work-Time Komplexität als Maß für die Komplexität paralleler Algorithmen eingeführt. Im Anschluss wurde eine Einführung in die hier verwendete GPGPU Plattform CUDA gegeben. Dabei wurde das Konzept paralleler Methodenaufrufe anhand von Beispielen erläutert und die Speicherhierachie der verwendeten GPUs erklärt. Diese wurde abschließend mit alternativen Architekturen verglichen und festgestellt, dass grundsätzliche Konzepte wie Blöcke, Threads oder gemeinsamer Speicher in allen GPGPU Architekturen verfügbar sind.

Im Anschluss wurden in Kapitel 5 verschiedene CRF Implementierungen vorgestellt und auf ihre Eignung zur Integration paralleler GPU Algorithmen überprüft, von denen zwei als Referenzimplementierungen im 6. Kapitel genutzt wurden. Allerdings konnte keine, für eine GPGPU Portierung geeignete, Implementierung gefunden werden, da die erzeugten Modelle entweder zu groß für den globalen Speicher der GPU waren oder C++ Strukturen verwendet wurden, die im GPU Code nicht verfügbar sind. Daher wurde eine eigene Implementierung entwickelt deren paralleler Kontrollfluss auf Basis zweier teilparallelisierter CPU Implementierungen entworfen wurde. Es wurden Datenstrukturen sowie parallele Algorithmen für drei Inferenz-, zwei Klassifikations und zwei Optimierungsalgorithmen vorgestellt. Diese wurden in einem auf GPGPU angepasstem Pseudocode präsentiert um eine möglichst einfache Portierung auf andere Architekturen zu erlauben. Dort wurde ebenfalls versucht dem Vorschlag aus [5, 40] nachzugehen und die parallele Struktur der Loopy-Belief-Propagation in einen parallelen Algorithmus umzusetzen. Die Datenstrukturen und Algorithmen wurden zu einer Programmbibliothek und schließlich zu einer lauffähigen CRF Implementierung zusammengefügt, deren Verwendung im letzten Abschnitt kurz erläutert wurde.

Im 6. Kapitel wurde die hier entwickelte parallele GPGPU Implementierung von Conditional Random Fields letztendlich auf drei verschiedenen Datensätzen getestet und mit zwei Referenzimplementierungen verglichen. Die Resultate waren für zwei der drei Inferenzalgorithmen eher ernüchternd. Obgleich LBP eine inhärent parallele Struktur aufweist, war es nicht möglich praxistaugliche Laufzeiten zu erzielen. Die LBP war zu langsam um sie angemessen evaluieren zu können. Die serielle Belief-Propagation war zwar schneller als die Referenzimplementierung, die L-BFGS zur Optimierung verwendet, jedoch langsamer als die SGD Referenz. Ein anderes Bild zeigte sich bei der Evaluation der parallelen Implementierung des Forward-Backward Algorithmus. Dieser war auf zwei von drei Datensätzen zwischen 4 und 27 mal so schnell wie die Referenzimplementierungen, wobei der Vorsprung auf die SGD Referenz mit wachsender Anzahl an Merkmalen zunahm.

Insgesamt wurden die im ersten Kapitel formulierten Ziele dieser Diplomarbeit erreicht. Ein hochparalleles CRF-Design wurde unter Berücksichtigung bereits bekannter Parallelisierungen entworfen, implementiert und evaluiert. Zumindest für Linear-Chain CRFs konnte das Training sowie die Klassifikation beschleunigt werden. CRF^{GPU} ist die erste CRF Implementierung unter Verwendung von GPGPU.

7.1 Ausblick

Auf dem Weg von der Theorie graphischer Modelle bis hin zu einer parallelen Implementierung für GPUs wurden einige Methoden und Konzepte nur am Rande erwähnt. Neben Inferenzverfahren wie Junction-Trees und dem Variablen-Eliminationsalgorithmus scheint insbesondere die in Factorie verwendete Sampling-Methode eine effiziente Alternative zur Loopy-Belief-Propagation zu sein. In [75] wird erwähnt, dass mit Hilfe mehrerer paralleler Samplings der Approximationsfehler stark reduziert werden kann.

Abgesehen von der Implementierung neuer Verfahren könnte zumindest für kleine Graphen versucht werden, die Nachrichtenberechnung bei BP und LBP im gemeinsamen Speicher zu berechnen. Zur Durchführung von Versuchen auf anderen graphischen Strukturen wäre es sinnvoll das Einlesen von GraphML Dateien zu ermöglichen. Diese erlauben eine komfortable Spezifikation von Graphen mit Hilfe einer XML-Syntax. Es ist ebenfalls angedacht CRF^{GPU} auf OpenCL zu portieren, und die Performanz der Implementierung auch auf anderen GPGPU Architekturen testen zu können. Abgesehen von einer Änderungen der Soft- oder Hardware, wären auch Resultate auf anderen Datensätzen interessant.

Literatur

[1] Srinivas M. Aji and Robert J. McEliece, *The Generalized Distributive Law*, IEEE Transactions on Information Theory **46** (2000), no. 2, 325–342.

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski, *Complexity of finding embeddings in a k-tree*, SIAM J. Algebraic Discrete Methods **8** (1987), 277–284.

[3] Ole E. Barndorff-Nielsen, *Information and exponential families*, John Wiley and Sons Ltd, Chichester, West Sussex, UK, April 1978.

[4] Christopher M. Bishop, *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford, November 1995.

[5] Christopher M. Bishop, *Pattern recognition and machine learning (information science and statistics)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Antoine Bordes, Léon Bottou, and Patrick Gallinari, *SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent*, J. Mach. Learn. Res. **10** (2009), 1737–1754.

[7] Léon Bottou, *Large-scale machine learning with stochastic gradient descent*, Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010) (Paris, France) (Yves Lechevallier and Gilbert Saporta, eds.), Springer, August 2010, pp. 177–187.

[8] Léon Bottou and Yann LeCun, *On-line learning for very large datasets*, Applied Stochastic Models in Business and Industry **21** (2005), no. 2, 137–151.

[9] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, *Brook for gpus: stream computing on graphics hardware*, ACM SIGGRAPH 2004 Papers (New York, NY, USA), SIGGRAPH '04, ACM, 2004, pp. 777–786.

[10] Markus Bundschus, Mathaeus Dejori, Martin Stetter, Volker Tresp, and Hans P. Kriegel, *Extraction of semantic biomedical relations from text using conditional random fields*, BMC Bioinformatics **9** (2008), no. 1.

[11] David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley Professional, May 1997.

[12] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, Massachusetts, October 2000.

[13] E. Chow and D. Hysom, *Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters*, Tech. Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001.

[14] Peter Clifford, *Markov random fields in statistics*, Disorder in physical systems (1990), 19–32.

[15] Corinna Cortes and Vladimir Vapnik, *Support-Vector Networks*, Mach. Learn. **20** (1995), 273–297.

[16] Koby Crammer and Yoram Singer, *Ultraconservative online algorithms for multiclass problems*, J. Mach. Learn. Res. **3** (2003), 951–991.

[17] Stephen Della Pietra, Vincent Della Pietra, and John Lafferty, *Inducing features of random fields*, IEEE Trans. Pattern Anal. Mach. Intell. **19** (1997), 380–393.

[18] Martin Dietzfelbinger, *From Randomized Algorithms to PRIMES is in P*, Springer, Berlin, Heidelberg, March 2004.

[19] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo, *Frequent itemset mining on graphics processors*, DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware (New York, NY, USA), ACM, 2009, pp. 34–42.

[20] M. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput. **C-21** (1972), 948–960.

[21] William T. Freeman, Egon C. Pasztor, and Owen T. Carmichael, *Learning low-level vision*, Int. J. Comput. Vision **40** (2000), 25–47.

[22] Brendan J. Frey and David J. C. MacKay, *A revolution: belief propagation in graphs with cycles*, Proceedings of the 1997 conference on Advances in neural information processing systems 10 (Cambridge, MA, USA), NIPS '97, MIT Press, 1998, pp. 479–485.

[23] Peter Fricke, Felix Jungermann, Katharina Morik, Nico Piatkowski, Olaf Spinczyk, and Marco Stolpe, *Towards adjusting mobile devices to user's behaviour*, Proceedings of the International Workshop on Mining Ubiquitous and Social Environments (MUSE 2010), 2010.

[24] S. Geman and D. Geman, *Stochastic relaxation, gibbs distributions, and the bayesian restoration of images*, pp. 452–472, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[25] Geoffrey Grimmett, *Probability on graphs - random processes on graphs and lattices*, 1nd ed., Cambridge University Press, June 2010.

[26] Khronos OpenCL Working Group, *The OpenCL Specification 1.1*, Khronos Group, September 2010.

[27] John Michael Hammersley and Peter Clifford, *Markov fields on finite graphs and lattices*, Unpublished manuscript (1971).

[28] Mark Harris, *Optimizing Parallel Reduction in CUDA*, NVIDIA Corporation, 2008.

[29] M J Harvey and G De Fabritiis, *Swan: A tool for porting CUDA programs to OpenCL*, Computer Physics Communications **182** (2011), no. 4, 1093–1099.

[30] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning*, corrected ed., Springer, July 2003.

[31] Bai Hong-tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He, *K-means on commodity gpus with cuda*, Computer Science and Information Engineering, World Congress on **3** (2009), 651–655.

[32] E. Ising, *Beitrag zur Theorie des Ferromagnetismus*, Zeitschrift für Physik **31** (1925), 253–258.

[33] Robert A. Jacobs, *Increased rates of convergence through learning rate adaptation*, Neural Networks **1** (1988), no. 4, 295–307.

[34] Joseph JáJá, *An introduction to parallel algorithms*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[35] Joerg Keller, Christoph W. Keßler, and Jesper L. Traeff, *Practical pram programming*, Wiley Series on Parallel and Distributed Computing, Wiley and Sons, Redwood City, CA, USA, 2000.

[36] Ross Kindermann and J. Laurie Snell, *Markov Random Fields and Their Applications*, Contemporary mathmatics **1** (1950).

[37] Jyrki Kivinen and Manfred K. Warmuth, *Exponentiated gradient versus gradient descent for linear predictors*, Inf. Comput. **132** (1997), 1–63.

[38] Andrei Nikolajewitsch Kolmogorow, *Grundbegriffe der Wahrscheinlichkeitstheorie*, Springer, Berlin, Germany, 1933.

[39] Frank R. Kschischang and Frey, *Iterative decoding of compound codes by probability propagation in graphical models*, IEEE Journal on Selected Areas in Communications **16** (1998), 219–230.

[40] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger, *Factor graphs and the sum-product algorithm*, IEEE Transactions on Information Theory **47** (2001), no. 2, 498–519.

[41] John Lafferty, Andrew McCallum, and Fernando Pereira, *Conditional random fields: Probabilistic models for segmenting and labeling sequence data*, Proc. 18th International Conf. on Machine Learning (2001), 282–289.

[42] Steffen L. Lauritzen and David J. Spiegelhalter, *Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems*, Journal of the Royal Statistical Society, Series B **50** (1988), no. 2, 157–224.

[43] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini, *Building a large annotated corpus of english: the penn treebank*, Comput. Linguist. **19** (1993), 313–330.

[44] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler, *Yale: Rapid prototyping for complex data mining tasks*, KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (New York, NY, USA) (Lyle Ungar, Mark Craven, Dimitrios Gunopulos, and Tina Eliassi-Rad, eds.), ACM, August 2006, pp. 935–940.

[45] Tom Minka, *Discriminative models, not discriminative training*, Tech. Report MSR-TR-2005-144, Microsoft Research, Cambridge, October 2005.

[46] Noboru Murata, *A statistical study of on-line learning*, pp. 63–92, Cambridge University Press, New York, NY, USA, 1998.

[47] K. Murphy, Y. Weiss, and M. Jordan, *Loopy belief propagation for approximate inference: An empirical study*, Uncertainty in Artificial Intelligence **15** (1999), 467–475.

[48] A. Y. Ng and M. I. Jordan, *On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes*, Advances in Neural Information Processing Systems **14** (2002), 841–848.

[49] Jorge Nocedal, *Updating quasi-newton matrices with limited storage*, Mathematics of Computation **35** (1980), no. 151, 773–782.

[50] *nVidia Corporation*, *CUDA Programming Guide 3.2*, November 2010.

[51] *nVidia Corporation*, *CUDA Reference Manual 3.2*, November 2010.

[52] Bernd Oestereich, *Die UML 2.0 Kurzreferenz für die Praxis*, 4 ed., Oldenbourg Wissenschaftsverlag GmbH, München, Deutschland, 2005.

[53] Judea Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[54] Barak A. Pearlmutter, *Fast exact multiplication by the hessian*, Neural Computation **6** (1994), no. 1, 147–160.

[55] Xuan-Hieu Phan, Le-Minh Nguyen, Yasushi Inoguchi, and Susumu Horiguchi, *High-performance training of conditional random fields for large-scale applications of labeling sequence data*, IEICE - Trans. Inf. Syst. **E90-D** (2007), 13–21.

[56] J. R. Quinlan, *Induction of decision trees*, Mach. Learn. **1** (1986), 81–106.

[57] L. R. Rabiner, *A tutorial on hidden markov models and selected applications in speech recognition*, Proceedings of the IEEE **77** (1989), no. 2, 257–286.

[58] Lance Ramshaw and Mitch Marcus, *Text Chunking Using Transformation-Based Learning*, Proceedings of the Third Workshop on Very Large Corpora (Somerset, NJ, USA) (David Yarovsky and Kenneth Church, eds.), Association for Computational Linguistics, 1995, pp. 82–94.

[59] Mark Schmidt, Alexandru Niculescu-Mizil, and Kevin Murphy, *Learning graphical model structure using l1-regularization paths*, Proceedings of the 22nd national conference on Artificial intelligence - Volume 2, AAAI Press, 2007, pp. 1278–1283.

[60] N. Schraudolph and T. Graepel, *Combining Conjugate Direction Methods with Stochastic Approximation of Gradients*, Ninth International Workshop on Artificial Intelligence and Statistics, 2002.

[61] Burr Settles, *Biomedical named entity recognition using conditional random fields and rich feature sets*, Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications (Stroudsburg, PA, USA), JNLPBA '04, Association for Computational Linguistics, 2004, pp. 104–107.

[62] Fei Sha and Fernando Pereira, *Shallow parsing with conditional random fields*, NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (Morristown, NJ, USA), Association for Computational Linguistics, 2003, pp. 134–141.

[63] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro, *Pegasos: Primal estimated sub-gradient solver for svm*, Proceedings of the 24th international conference on Machine learning (New York, NY, USA), ICML '07, ACM, 2007, pp. 807–814.

[64] Claude Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal **27** (1948), 379–423, 623–656.

[65] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, USA, 1995.

[66] Bjarne Stroustrup, *Die C++ Programmiersprache*, 4 ed., Addison Wesley, München, Germany, 2000.

[67] Charles Sutton and Andrew McCallum, *An Introduction to Conditional Random Fields for Relational Learning*, Introduction to Statistical Relational Learning (Lise Getoor and Ben Taskar, eds.), MIT Press, 2007.

[68] Erik F. Tjong Kim Sang and Sabine Buchholz, *Introduction to the conll-2000 shared task: chunking*, Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning - Volume 7 (Morristown, NJ, USA), ConLL '00, Association for Computational Linguistics, 2000, pp. 127–132.

[69] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun, *Large margin methods for structured and interdependent output variables*, J. Mach. Learn. Res. **6** (2005), 1453–1484.

[70] C. J. van Rijsbergen, *Information Retrieval*, 2 ed., Butterworths, London, 1979.

[71] Vladimir N. Vapnik, *The nature of statistical learning theory*, Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[72] S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin P. Murphy, *Accelerated training of conditional random fields with stochastic gradient methods*, ICML '06: Proceedings of the 23rd international conference on Machine learning (New York, NY, USA), ACM, 2006, pp. 969–976.

[73] S. V.N. Vishwanathan, Nicol N. Schraudolph, and Alex J. Smola, *Step Size Adaptation in Reproducing Kernel Hilbert Space*, J. Mach. Learn. Res. **7** (2006), 1107–1133.

[74] Taro Watanabe, Jun Suzuki, Hajime Tsukada, and Hideki Isozaki, *Online large-margin training for statistical machine translation*, EMNLP-CoNLL, 2007, pp. 764–773.

[75] Michael Wick, Andrew McCallum, and Gerome Miklau, *Scalable probabilistic databases with factor graphs and MCMC*, Proc. VLDB Endow. **3** (2010), 794–804.

[76] Philip Wolfe, *Convergence conditions for ascent methods*, SIAM Review **11** (1969), no. 2, 226–235.

[77] Han Xiao, *Towards Parallel and Distributed Computing in Large-Scale Data Mining: A Survey*, Tech. report, Technical University of Munich, Garching near Munich, Germany, April 2010.

[78] Le-Minh Nguyen Xuan-Hieu Phan and Cam-Tu Nguyen, *Flexcrfs: Flexible conditional random field toolkit*, Unpublished (2005), http://flexcrfs.sourceforge.net.

[79] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss, *Understanding belief propagation and its generalizations*, Exploring artificial intelligence in the new millennium (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 2003, pp. 239–269.

[80] Nevin Zhang and David Poole, *A simple approach to Bayesian network computations*, Proceedings of the Tenth Canadian Conference on Artificial Intelligence, 1994, pp. 171–178.

A Kommandozeilenparameter von CRF^{GPU}

```
Usage: ./crfgpu [Options] < TRAININGDATA
GENERAL:
  --predict                           Don't learn. Just predict the labels of
                                       the inputdata and write them to stdout.
                                       Needs model and testfile.
  -Y [ --labelcount ] arg (=32)       Max. number of labels, <65
  -b [ --batchsize ] arg (=96)        Batchsize
  -I [ --iterations ] arg (=1)        Number of iterations
  -M [ --maxnodes ] arg (=100)        Max. number of nodes, <2049
  -w [ --weight ] arg (=0.0500000007) Initial weight
  -A [ --inference ] arg (=1)         Inference Algorithm:
                                       1: Forward-Backward, implies linear-chain graph
                                       2: Belief-Propagation, serial-schedule
                                       3: Belief-Propagation, flooding-schedule
  -G [ --graph ] arg (=1)             Graphical Structure:
                                       1: Linear-Chain
                                       2: Grid
  -O [ --optimizer ] arg (=1)         Optimizer:
                                       1: Stochastic Gradient (SGD)
                                       3: Annealed Stochastic Gradient (ASGD)
                                       3: Stochastic Meta-Descent (SMD)
  -h [ --help ]                       this help

FILE HANDLING:
  -F [ --modelfile ] arg Model file
  -L [ --labelfile ] arg Label output file
  -T [ --testfile ] arg  Testset

OPTIMIZATION:
  -e [ --eta ] arg (=0.100000001)   Stepsize
  -l [ --lambda ] arg (=1)          SMD long-term dependence [0;1]
  -m [ --mu ] arg (=0.100000001)    SMD meta stepsize
  -d [ --disable-parameter-tying ]  Disable parameter tying (BP only)

MISC:
  --iob                 Use IOB-Tagging
  --clear               Clear screen after every iteration
  --prf                 Show precision, recall and F1-score after every iteration
  --likelihood          Compute likelihood value

CUDA:
  --workerblocksP arg (=64)   Num worker blocks for parameter computations
  --workerthreadsP arg (=128) Num worker threads for parameter computations
  --workerblocksF arg (=32)   Num worker blocks for factornode computations
  --workerthreadsY arg (=16)  Num worker threads for label computations
  --workerthreadsV arg (=2)   Num worker threads for neighbour computations
  --device arg (=0)           Choose CUDA device
```

B Reguläre Ausdrücke der orthographischen Merkmale

Init Caps	[A-Z].*
Init Caps Alpha	[A-Z][a-z]*
All Caps	[A-Z]+
Caps Mix	[A-Za-z]+
Has Digit	.*[0-9].*
Single Digit	[0-9]
Double Digit	[0-9][0-9]
Natural Number	[0-9]+
Real Number	[-\+][[0-9]+[\.,]+[0-9].,]+
Alpha-Numeric	[A-Za-z0-9]+
Roman	[ivxdlcm]+|[IVXDLCM]+
Has Dash	.*-.*
Init Dash	-.*
End Dash	.*-
Punctuation	[,\.;:\?!-\+"]
Greek	(alpha|beta|...|omega)
Has Greek	.*\b(alpha|beta|...|omega)\b.*
Mutation Pattern	\w*\d+-*\D+

Tabelle B.1: Reguläre Ausdrücke der orthographischen Merkmale. Diese wurden für die Vorverarbeitung der JNLPBA-2004 Datensätze verwendet.

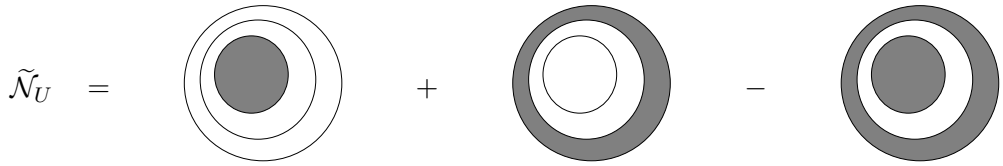


Abbildung C.1: Visualisierung des $\widetilde{\mathcal{N}_U}$-Operators (Gleichung (C.3)). Die Flächen stellen (von Innen nach Außen) die Mengen U, $\Delta(U)$ sowie $\overline{U + \Delta(U)}$ dar. Graue Flächen haben das Label **null**, weiße Flächen haben ein beliebiges Label aus $\mathcal{Y}$.

C Beweis des Hammersley-Clifford Theorems

Die Beweisführung verläuft analog zu [14, 27]. Zuerst wird ein besonderer Operator, der sog. *Blackening-Operator*, eingeführt. Anschließend wird die Menge von Funktionen charakterisiert, deren Elemente invariant gegenüber der Anwendung des Operators sind. Mit Hilfe der Markov-Eigenschaft wird gezeigt, dass die logarithmierte Dichtefunktion Graphischer Modelle in dieser Menge enthalten sein muss, woraus letztendlich (2.23) folgt.

Sei $M = (G, p)$ ein PGM und $\mathcal{Y}$ die Menge der Label. O.B.d.A. sei das Label **null** in $\mathcal{Y}$ enthalten. Die Notation $\mathbf{y}_U[W]$ bezeichne eine Realisation der Knoten in $U \subseteq V$, in der die Knoten in $W \subseteq V$ das Label **null** tragen. Eine Realisation $\mathbf{y} \in \mathcal{Y}^n$ heißt *unvollständig*, falls sie mindestens einem Knoten das Label **null** zuweist, ansonsten *vollständig*. Falls also $U \cap W = \emptyset$ wird keinem Knoten das **null**-Label zugewiesen und die Realisation $\mathbf{y}_U[W]$ ist vollständig, sofern $\mathbf{y}_U$ vollständig ist. Dabei ist zu beachten, dass das Label **null** keine besonderen Eigenschaften besitzt und bis auf seine Bezeichnung nicht von den anderen Labeln aus $\mathcal{Y}$ unterscheidbar ist. Die Annahme der Existenz des **null**-Labels dient ausschließlich der Beweisführung und wird zur Anwendung des Theorems nicht benötigt.

Der Blackening-Operator. Sei $\mathcal{R}$ die Menge aller reellwertigen Funktionen über $\mathcal{Y}^n \times \mathcal{P}(V)$. Funktionen aus $\mathcal{R}$ bilden die Label $\mathbf{y}_U$ einer Knotenmenge U auf eine reelle Zahl aus $\mathbb{R}$ ab. Ferner sei $\mathcal{N}_U$ ein einstelliger kommutativer Operator für den (C.1) und (C.2) gelten. Der Operator $\mathcal{N}_U$ verändert das Argument $\mathbf{y}$ einer Funktion aus $\mathcal{R}$ so, dass die Variablen in U die Klasse **null** zugewiesen wird. Mit diesem "Trick" kann dass Verhalten einer Funktion aus $\mathcal{R}$ beim "ausblenden" der Realisation der Knoten aus U analysiert werden.

$$\mathcal{N}_U f(\mathbf{y}) = f(\mathbf{y}[U]), \forall f \in \mathcal{R} \tag{C.1}$$

$$\mathcal{N}_U \mathcal{N}_W = \mathcal{N}_W \mathcal{N}_U = \mathcal{N}_{U+W} \tag{C.2}$$

Aus den Gleichungen (C.1, C.2) sowie $\mathcal{N}_\emptyset = 1$ lässt sich folgern, dass $\mathcal{N}_U$ eine kommutative Algebra auf $\mathcal{R}$ induziert. Eine zentrale Eigenschaft (Lemma C.1) des Operators $\mathcal{N}_U$ folgt direkt aus seiner Definition.

Lemma C.1. *Ist W eine Teilmenge von U, so ist $\mathcal{N}_U(1 - \mathcal{N}_W) = 0$.*

Dieses Lemma wird sich im weiteren Verlauf als sehr nützlich erweisen. Für den Beweis von Theorem 2.3 muss gezeigt werden, dass die Dichte p über die Realisationen der Cliquen des Graphen zerfällt. Unter Berücksichtigung der Definition einer Clique (Gleichung (2.21)) wird der Operator $\widetilde{\mathcal{N}}$ gebildet (C.3), welcher das Argument einer Funktion aus $\mathcal{R}$ entlang der Nachbarschaft eines jeden Knotens in V aufspaltet, indem das Label der übrigen Knoten auf **null** gesetzt wird

(C.4). Abbildung C.1 soll diesen Zusammenhang veranschaulichen.

$$\widetilde{\mathcal{N}} = \prod_{v \in V} \widetilde{\mathcal{N}}_v \tag{C.3}$$

$$\widetilde{\mathcal{N}}_U = \mathcal{N}_U + \mathcal{N}_{\overline{U+\Delta(U)}} - \mathcal{N}_{\overline{\Delta(U)}} = \mathcal{N}_U + \underbrace{\mathcal{N}_{\overline{U+\Delta(U)}} \left(1 - \mathcal{N}_U\right)}_{\mathcal{N}_U^*} \tag{C.4}$$

Durch das Ausmultiplizieren der Faktoren in (C.3) sieht man leicht, dass sich der Operator $\widetilde{\mathcal{N}}$ anstatt einem Produkt über alle Knoten äquivalent als Summe über alle Teilmengen der Knotenmenge V formulieren lässt (C.5). $\mathcal{P}(V)$ bezeichnet die Potenzmenge einer Menge V.

$$\widetilde{\mathcal{N}} = \prod_{v \in V} \widetilde{\mathcal{N}}_v = \sum_{U \in \mathcal{P}(V)} \mathcal{N}_{\overline{U}} \underbrace{\prod_{u \in U} \mathcal{N}_u^*}_{\widetilde{\mathcal{N}}_U^*} = \sum_{U \in \mathcal{P}(V)} \mathcal{N}_{\overline{U}} \widetilde{\mathcal{N}}_U^* \tag{C.5}$$

Falls eine dieser Teilmengen $U \in \mathcal{P}(V)$, $U \neq \emptyset$ keine Clique ist, so gibt es zwei Knoten in U die nicht benachbart sind. In diesem Fall lassen sich die Faktoren in $\widetilde{\mathcal{N}}_U^*$ so umstellen, dass Lemma C.1 angewendet werden kann. Damit ist der ganze Summand für U gleich 0 und kann somit in der Summation ausgelassen werden. Also reicht es aus, die Summe über den Cliquen von G sowie $U = \emptyset$ zu bilden.

$$\widetilde{\mathcal{N}} = \mathcal{N}_V + \sum_{C \in \mathcal{C}(G)} \mathcal{N}_{\overline{C}} \widetilde{\mathcal{N}}_C^* \tag{C.6}$$

Der Ausdruck lässt sich weiter vereinfachen. Wendet man den $\widetilde{\mathcal{N}}$-Operator auf eine Funktion $f \in \mathcal{R}$ mit Argument $\mathbf{y} \in \mathcal{Y}^n$ an, erhält man (C.7).

$$\widetilde{\mathcal{N}} f(\mathbf{y}) = f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}(G)} \mathcal{N}_{\overline{C}} \widetilde{\mathcal{N}}_C^* f(\mathbf{y}) \tag{C.7}$$

Wie oben erwähnt, ist das Label **null** in $\mathcal{Y}$ enthalten. Daher kann ein Knoten v in der Realisierung $\mathbf{y}$ bereits das **null**-Label haben, das heißt es gilt $\mathbf{y}_v = \textbf{null}$ sowie $\mathcal{N}_v f(\mathbf{y}) = f(\mathbf{y})$. Substituiert man nun $\widetilde{\mathcal{N}}_C^*$ und daraufhin $\mathcal{N}_u^*$ mit ihren jeweiligen Definitionen folgt Gleichung (C.8).

$$\begin{aligned} \widetilde{\mathcal{N}} f(\mathbf{y}) &= f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}(G)} \mathcal{N}_{\overline{C}} \prod_{u \in C} \mathcal{N}_{\overline{u+\Delta(u)}} \left(1 - \mathcal{N}_u\right) f(\mathbf{y}) \\ &= f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}(G)} \mathcal{N}_{\overline{C}} \prod_{u \in C} \mathcal{N}_{\overline{u+\Delta(u)}} \underbrace{\left(1 - \mathcal{N}_u\right) \mathcal{N}_v}_{=0, \text{falls } u=v} f(\mathbf{y}) \end{aligned} \tag{C.8}$$

Folgerichtig kann erneut Lemma C.1 angewendet werden, wodurch die Summanden aller unter $\mathbf{y}$ unvollständigen Cliquen entfallen.

Sei nun $\mathcal{C}_{\mathbf{y}}(G)$ die Menge aller unter $\mathbf{y}$ vollständigen Cliquen auf G. Gleichung (C.9) folgt dann aus der Definition von Cliquen (2.21).

$$\begin{aligned}\widetilde{\mathcal{N}} f(\mathbf{y}) &= f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \mathcal{N}_{\overline{C}} \prod_{u \in C} \mathcal{N}_{\overline{u+\Delta(u)}} (1 - \mathcal{N}_u) f(\mathbf{y}) \\ &= f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \prod_{u \in C} \mathcal{N}_{\overline{u+\Delta(u)}} (1 - \mathcal{N}_u) f\left(\mathbf{y}\left[\overline{C}\right]\right) \\ &= f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \prod_{u \in C} (1 - \mathcal{N}_u) f\left(\mathbf{y}\left[\overline{C}\right]\right) \end{aligned} \tag{C.9}$$

Gleichung (C.10) repräsentiert die endgültige Form des Blackening-Operators $\widetilde{\mathcal{N}}$, wobei die Funktion f eine beliebige Funktion aus $\mathcal{R}$ ist. Da in der Realisation $\mathbf{y}[V]$ alle Knoten auf das Label **null** gesetzt werden, ist der Wert von $f(\mathbf{y}[V])$ unabhängig von $\mathbf{y}$ und für ein festes f konstant.

$$\widetilde{\mathcal{N}} f(\mathbf{y}) = f(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \prod_{u \in C} (1 - \mathcal{N}_u) f\left(\mathbf{y}\left[\overline{C}\right]\right) \tag{C.10}$$

Invarianz. Der Blackening-Operator zerlegt eine Funktion aus $\mathcal{R}$ in eine Summe über die Cliquen mit vollständiger Realisation sowie eine Konstante. Eine Funktion $f \in \mathcal{R}$, die invariant gegenüber $\widetilde{\mathcal{N}}$ ist, muss diese Eigenschaft ebenfalls besitzen. Sei also $\mathcal{I}\left(\widetilde{\mathcal{N}}\right)$ die Menge all dieser f mit $\widetilde{\mathcal{N}} f(\mathbf{y}) = f(\mathbf{y})$. Da die Funktion $f \in \mathcal{R}$ bis jetzt beliebig gewählt war, gelten die obigen Aussagen auch für die Funktion $\Phi \in \mathcal{R}$ in Gleichung (C.11), wobei die Funktion $\phi \in \mathcal{R}$ erneut beliebig gewählt werden kann.

$$\Phi(\mathbf{y}) = \phi(\mathbf{y}[V]) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \phi\left(\mathbf{y}\left[\overline{C}\right]\right) \tag{C.11}$$

Da in den betrachteten Realisationen $\mathbf{y}\left[\overline{C}\right]$ alle Knoten, die nicht zur Clique C gehören, das Label **null** haben, ist der Funktionswert von ϕ nur von den Labeln der Knoten in C abhängig. Daher kann ϕ durch eine Menge von Funktionen $\{\phi_\emptyset\} \cup \{\phi_C\}_{C \in \mathcal{C}_{\mathbf{y}}(G)}$ ersetzt werden, deren Funktionswerte an allen relevanten Stellen mit ϕ übereinstimmen. Für diese Funktionen gilt $\phi_U(\mathbf{y}) := \phi\left(\mathbf{y}\left[\overline{U}\right]\right)$ für alle Teilmengen $U \subseteq V$. Mit dieser Feststellung lässt sich (C.11) auch als (C.12) schreiben.

$$\Phi(\mathbf{y}) = \phi_\emptyset(\mathbf{y}) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \phi_C(\mathbf{y}) \tag{C.12}$$

Sei $\mathcal{J}$ die Menge aller Φ-Funktionen, wobei sich die Funktionen in $\mathcal{J}$ allein durch die Wahl von ϕ unterscheiden. Da $\prod_{u \in C} (1 - \mathcal{N}_u) f\left(\mathbf{y}\left[\overline{C}\right]\right)$ eine Funktion von C ist, sei $\phi_C(\mathbf{y}) := \prod_{u \in C} (1 - \mathcal{N}_u) f\left(\mathbf{y}\left[\overline{C}\right]\right)$ sowie $\phi_\emptyset(\mathbf{y}) := \phi(\mathbf{y}[V])$. Setzt man dies in Gleichung (C.10) ein, wird klar, dass jede beliebige $\widetilde{\mathcal{N}}$-invariante Funktion in der Form von Gleichung (C.12) geschrieben werden kann und es gilt die Teilmengenbeziehung (C.13). Wenn eine Funktion also invariant gegenüber $\widetilde{\mathcal{N}}$ ist, dann hat sie die Form von Φ.

$$\mathcal{I}\left(\widetilde{\mathcal{N}}\right) \subseteq \mathcal{J} \tag{C.13}$$

Nun wird gezeigt das die Umkehrung ebenfalls gilt.

Die Gleichungen (C.14) bis (C.16) folgen allesamt aus der Definition von Φ, wobei c in (C.16) ein

beliebiger Knoten der Clique C ist.

$$\Phi\left(\mathbf{y}\left[\,V\,\right]\right) = \phi_{\emptyset}\left(\mathbf{y}\right) \tag{C.14}$$

$$\Phi\left(\mathbf{y}\left[\,\overline{C}\,\right]\right) = \phi_{\emptyset}\left(\mathbf{y}\right) + \sum_{X \subseteq C} \phi_X\left(\mathbf{y}\right) \tag{C.15}$$

$$\Phi\left(\mathbf{y}\left[\,\overline{C} + c\,\right]\right) = \phi_{\emptyset}\left(\mathbf{y}\right) + \sum_{X \subseteq C - c} \phi_X\left(\mathbf{y}\right) \tag{C.16}$$

Durch Subtraktion von (C.16) und (C.15) folgt (C.17).

$$\left(1 - \mathcal{N}_c\right) \Phi\left(\mathbf{y}\left[\,\overline{C}\,\right]\right) = \sum_{\{c\} \subseteq X \subseteq C} \phi_X\left(\mathbf{y}\right) \tag{C.17}$$

Dieser Schritt kann für jeden Knoten in C wiederholt werden.

$$\begin{aligned}\left(1 - \mathcal{N}_{c_{|C|}}\right) \ldots \left(1 - \mathcal{N}_{c_1}\right)\left(1 - \mathcal{N}_{c_0}\right) \Phi\left(\mathbf{y}\left[\,\overline{C}\,\right]\right) &= \sum_{C \subseteq X \subseteq C} \phi_X\left(\mathbf{y}\right) \\ \Leftrightarrow \prod_{c \in C} \left(1 - \mathcal{N}_c\right) \Phi\left(\mathbf{y}\left[\,\overline{C}\,\right]\right) &= \phi_X\left(\mathbf{y}\right)\end{aligned} \tag{C.18}$$

Wendet man nun den Blackening-Operator (C.10) auf Φ an und setzt (C.18) und (C.14) ein, so folgt die Invarianz von Φ gegenüber $\widetilde{\mathcal{N}}$.

$$\widetilde{\mathcal{N}}\Phi\left(\mathbf{y}\right) = \phi_{\emptyset}\left(\mathbf{y}\right) + \sum_{C \in \mathcal{C}_{\mathbf{y}}(G)} \phi_C\left(\mathbf{y}\right) = \Phi\left(\mathbf{y}\right) \tag{C.19}$$

Diese Aussage ist unabhängig von der Wahl von ϕ und es gilt $\mathcal{J} \subseteq \mathcal{I}\left(\widetilde{\mathcal{N}}\right)$. Demnach ist jede Funktion mit der Form von Φ invariant gegenüber $\widetilde{\mathcal{N}}$. Zusammen mit (C.13) folgt (C.20).

$$\mathcal{I}\left(\widetilde{\mathcal{N}}\right) = \mathcal{J} \tag{C.20}$$

Folglich ist die Menge aller $\widetilde{\mathcal{N}}$-invarianten Funktionen mit der Menge der Φ-Funktionen identisch.

Finale. Nun sind alle Teile vorhanden um die eigentliche Aussage des Theorems zusammenzusetzen. Tatsächlich wird hier nur die Richtung

$$\text{Modell } M = (G, p) \text{ besitzt Markov-Eigenschaft} \Rightarrow \begin{array}{c} p \text{ zerfällt in Funktionen der} \\ \text{Realisationen der Cliquen von } G \end{array}$$

gezeigt. Auf den Beweis der Gegenrichtung wird an dieser Stelle verzichtet, da diese Aussage für die in dieser Diplomarbeit vorgestellten Methoden nicht relevant ist. Der Beweis findet sich zusammen mit dem hier gezeigten Teil des Beweises in [27].

Da das betrachtete Modell die Markov-Eigenschaft besitzt, ist die Realisation einer Menge von Knoten U nur von den Labeln der benachbarten Knoten $\Delta\left(U\right)$ abhängig. Daher gilt (C.21) für alle Realisationen $\mathbf{y}_U \in \mathcal{Y}^{|U|}$,

$$p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\overline{U}}\right) = p\left(\mathbf{y}_U \,|\, \mathbf{y}_{\Delta(U)}\right) \tag{C.21}$$

also auch für diejenigen, in denen alle Knoten aus U das Label **null** haben (C.22).

$$p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\left[U\right]\right)=p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\Delta(U)}\left[U\right]\right) \tag{C.22}$$

Da die Knoten in U definitionsgemäß weder im Komplement $\overline{U}$ noch in der Nachbarschaft $\Delta\left(U\right)$ vorkommen, ändern sich die Realisationen $\mathbf{y}_{\overline{U}}$ und $\mathbf{y}_{\Delta(U)}$ nicht, wenn den Knoten in U das Label **null** zugewiesen wird. Infolgedessen gilt $\mathbf{y}_{\overline{U}}\left[U\right]=\mathbf{y}_{\overline{U}}$ sowie $\mathbf{y}_{\Delta(U)}\left[U\right]=\mathbf{y}_{\Delta(U)}$, wodurch (C.22) äquivalent zu (C.23) sein muss.

$$p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\right)=p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\Delta(U)}\right) \tag{C.23}$$

Durch Division[16] von (C.21) und (C.23) erhält man (C.24).

$$\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\right)}=\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\Delta(U)}\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\Delta(U)}\right)} \tag{C.24}$$

Da das Modell die Markov-Eigenschaft besitzt, ist die Realisation der Knoten, die außerhalb der Nachbarschaft von U liegen, für die Bestimmung der Wahrscheinlichkeit einer Realisation der Knoten innerhalb von U irrelevant. Also spielt es weder eine Rolle, ob diese äußeren Knoten überhaupt betrachtet werden, noch ob ihnen das Label **null** zugewiesen wird (C.25).

$$p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\right)=p\left(\mathbf{y}_U\mid\mathbf{y}_{\Delta(U)}\right)=p\left(\mathbf{y}_U\mid\mathbf{y}_{\Delta(U)}\left[\overline{U+\Delta\left(U\right)}\right]\right)=p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right) \tag{C.25}$$

Setzt man dies in der rechten Seite von Gleichung (C.24) ein, ergibt sich (C.26).

$$\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\right)}=\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)} \tag{C.26}$$

Diese Gleichheit bleibt auch erhalten, wenn beide Seiten mit 1 multipliziert werden (C.27).

$$\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\right)}\cdot\frac{p\left(\mathbf{y}_{\overline{U}}\right)}{p\left(\mathbf{y}_{\overline{U}}\right)}=\frac{p\left(\mathbf{y}_U\mid\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)}{p\left(\mathbf{y}_U\left[U\right]\mid\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)}\cdot\frac{p\left(\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)}{p\left(\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)} \tag{C.27}$$

Nach der Definition der bedingten Wahrscheinlichkeit, gilt $p\left(A\mid B\right)\cdot p\left(B\right)=p\left(A,B\right)$. Folgerichtig muss das Verhältnis der Wahrscheinlichkeiten in (C.27) äquivalent zu dem in (C.28) sein.

$$\frac{p\left(\mathbf{y}_U,\,\mathbf{y}_{\overline{U}}\right)}{p\left(\mathbf{y}_U\left[U\right],\,\mathbf{y}_{\overline{U}}\right)}=\frac{p\left(\mathbf{y}_U,\,\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)}{p\left(\mathbf{y}_U\left[U\right],\,\mathbf{y}_{\overline{U}}\left[\overline{U+\Delta\left(U\right)}\right]\right)} \tag{C.28}$$

Da die Vereinigung von U und seinem Komplement $\overline{U}$ die komplette Knotenmenge V ergibt, ist die Wahrscheinlichkeit der gemeinsamen Realisation von $\mathbf{y}_U$ und $\mathbf{y}_{\overline{U}}$ identisch mit der von $\mathbf{y}$, d.h.

[16] An dieser Stelle wird implizit angenommen, dass die Wahrscheinlichkeit jeder Realisation größer als 0 ist. In [27, S.15-16] wird diskutiert, warum dies keine Einschränkung darstellt.

$p\left(\mathbf{y}_U, \mathbf{y}_{\overline{U}}\right) = p\left(\mathbf{y}\right)$. Auf diese Weise lassen sich offensichtlich alle gemeinsamen Realisationen in (C.28) zusammenfassen und es folgt (C.29)

$$\frac{p\left(\mathbf{y}\right)}{p\left(\mathbf{y}\left[U\right]\right)} = \frac{p\left(\mathbf{y}\left[\overline{U + \Delta\left(U\right)}\right]\right)}{p\left(\mathbf{y}\left[\overline{\Delta\left(U\right)}\right]\right)} \tag{C.29}$$

sowie durch Multiplikation mit $p\left(\mathbf{y}\left[U\right]\right)$ schließlich (C.30).

$$p\left(\mathbf{y}\right) = \frac{p\left(\mathbf{y}\left[\overline{U + \Delta\left(U\right)}\right]\right) \cdot p\left(\mathbf{y}\left[U\right]\right)}{p\left(\mathbf{y}\left[\overline{\Delta\left(U\right)}\right]\right)} \tag{C.30}$$

Demnach kann die Wahrscheinlichkeit einer beliebigen Realisation aller Knoten des Graphen als Verhältnis von Wahrscheinlichkeiten unvollständiger Realisationen ausgedrückt werden.

Durch logarithmieren[17] von Gleichung (C.30) erhält man Gleichung (C.31) sowie durch Anwendung des $\mathcal{N}_U$-Operators Gleichung (C.32).

$$\ln p\left(\mathbf{y}\right) = \ln p\left(\mathbf{y}\left[U\right]\right) + \ln p\left(\mathbf{y}\left[\overline{U + \Delta\left(U\right)}\right]\right) - \ln p\left(\mathbf{y}\left[\overline{\Delta\left(U\right)}\right]\right) \tag{C.31}$$

$$= \mathcal{N}_U \ln p\left(\mathbf{y}\right) + \mathcal{N}_{\overline{U+\Delta(U)}} \ln p\left(\mathbf{y}\right) - \mathcal{N}_{\overline{\Delta(U)}} \ln p\left(\mathbf{y}\right) \tag{C.32}$$

Dies entspricht offensichtlich der Definition des Blackening-Operators (C.3) und es folgt Gleichung (C.33).

$$\ln p\left(\mathbf{y}\right) = \widetilde{\mathcal{N}}_U \ln p\left(\mathbf{y}\right) \tag{C.33}$$

Folglich ist $\ln p$ invariant gegenüber $\widetilde{\mathcal{N}}_U$ für alle Teilmengen $U \subseteq V$ also auch für alle $v \in V$. Da der Blackening-Operator definiert ist als $\widetilde{\mathcal{N}} = \prod_{v \in V} \widetilde{\mathcal{N}}_v$, ist $\ln p$ in $\mathcal{I}\left(\widetilde{\mathcal{N}}\right)$ enthalten und hat somit die Form einer Φ-Funktion. Durch Invertierung des Logarithmus folgt direkt Gleichung (C.34).

$$\begin{aligned}
\ln p\left(\mathbf{y}\right) &= \phi_\emptyset\left(\mathbf{y}\right) + \sum_{C \in \mathcal{C}(G)} \phi_C\left(\mathbf{y}\right) \\
\Leftrightarrow p\left(\mathbf{y}\right) &= \exp\left(\phi_\emptyset\left(\mathbf{y}\right) + \sum_{C \in \mathcal{C}(G)} \phi_C\left(\mathbf{y}\right)\right) \\
\Leftrightarrow p\left(\mathbf{y}\right) &= p\left(\mathbf{y}\left[V\right]\right) \cdot \prod_{C \in \mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}\right)\right)
\end{aligned} \tag{C.34}$$

Da in $\mathbf{y}\left[V\right]$ alle Knoten das Label **null** haben, ist $p\left(\mathbf{y}\left[V\right]\right)$ eine Konstante und es folgt letztendlich (C.35) und damit die Behauptung[18].

[17] An dieser Stelle wird ebenfalls angenommen, dass kein Label eine Wahrscheinlichkeit von 0 besitzt.

[18] Hierbei ist zu beachten, dass ϕ_C eine beliebige Funktion ist, deren Funktionswert nur von der Realisation der Knoten der Clique C abhängt. Diese Formulierung macht keinen Gebrauch von der Annahme der Existenz des **null**-Labels.

$$p(\mathbf{y}) \propto \prod_{C \in \mathcal{C}(G)} \exp\left(\phi_C\left(\mathbf{y}_C\right)\right) \tag{C.35}$$

■

D CD-ROM