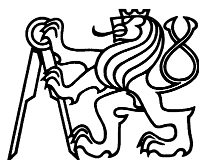
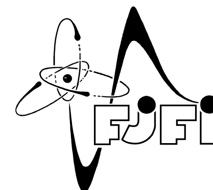


ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská



Paralelizace sítí typu Sum-Product-Transform pro architekturu GPU

Parallelization of Sum-Product-Transform type networks for the GPU architecture

Diplomová práce

Autor: **Bc. Ondřej Poláček**

Vedoucí práce: **Ing. Tomáš Oberhuber, Ph.D.**

Akademický rok: 2020/2021

Katedra: matematiky Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Ondřej Poláček
Studijní program:	Aplikace přírodních věd
Studijní obor:	Matematické inženýrství
Název práce (česky):	Paralelizace sítí typu Sum-Product-Transform pro architekturu GPU
Název práce (anglicky):	Paralelization of Sum-Product-Transform type networks for the GPU architecture

Pokyny pro vypracování:

1) Nastudujte implementaci sítě SPTN pro CPU.
2) Navrhněte a implementujte paralelní algoritmus pro běh na GPU s pomocí nástroje CUDA.
3) Proveďte výpočetní studii za účelem naměření urychlení paralelního algoritmu.
4) Aplikujte paralelní SPTN na reálná data.

Doporučená literatura:

1) T. Pevný, V. Šmídl, M. Trapp, O. Poláček, T. Oberhuber, Sum-Product-Transform Networks: Exploiting Symmetries using Invertible Transformations, accepted in Proceedings of Machine Learning Research, 2020

2) C. C. Aggarwal, Neural Networks and Deep Learning: A Textbook, Springer, 2018

3) J. Han, B. Sharma, Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++, Packt Publishing, 2019

Jméno a pracoviště vedoucího diplomové práce:

Ing. Tomáš Oberhuber, Ph.D.
Katedra matematiky FJFI, ČVUT v Praze, Trojanova 13, 120 00 Praha 2

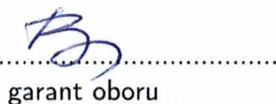
Jméno a pracoviště konzultanta:

Datum zadání diplomové práce: 31.10.2020

Datum odevzdání diplomové práce: 3.5.2021

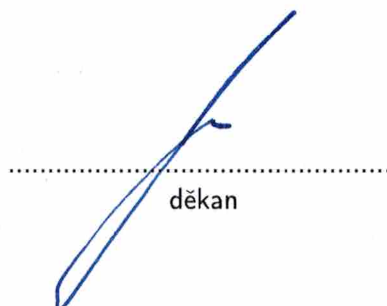
Doba platnosti zadání je dva roky od data zadání.

V Praze dne 21. října 2020



garant oboru

vedoucí katedry



děkan

Poděkování:
Chtěl bych zde poděkovat především svému školiteli Ing. Tomáši Oberhuberovi, Ph.D., za pečlivost, ochotu, vstřícnost a odborné i lidské zázemí při vedení mé diplomové práce.

Čestné prohlášení:
Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl jsem všechnu použitou literaturu.

V Praze dne 3. května 2021 Bc. Ondřej Poláček

Název práce:
Paralelizace sítí typu Sum-Product-Transform pro architekturu GPU

Autor: Bc. Ondřej Poláček

Obor: Matematické inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Ing. Tomáš Oberhuber, Ph.D.
Katedra matematiky FJFI, ČVUT v Praze,
Trojanova 13, 120 00 Praha 2

Abstrakt: Sítě typu Sum-Product-Transform jsou rozšířením sítí typu Sum-Product pomocí přidání transformačního uzlu. Toto rozšíření zvyšuje výpočetní náročnost všech operací se sítí. Tato práce popisuje implementaci sítí typu Sum-Product-Transform pro grafické procesory. Vysoce paralelní architektura GPU umožňuje dosáhnout většího výpočetního výkonu, než je výkon CPU. Je popsána implementace algoritmů v programovacím jazyce Julia pro inferenci, učení, generování náhodných vzorků a marginalizaci.

Klíčová slova: Butterfly rozklad, CUDA, GPU, hustota pravděpodobnosti, Julia, marginalizace, paralelizace, pravděpodobnostní distribuce, sampling, SPTN, strojové učení, SVD rozklad, zpětná propagace

Title:
Parallelization of Sum-Product-Transform type networks for the GPU architecture

Author: Bc. Ondřej Poláček

Abstract: Sum-Product-Transform type networks are an extension of Sum-Product type networks by an addition of a transformation node. This extension increases the computational complexity of all operations concerning the network. This work describes an implementation of Sum-Product-Transform type networks for graphics processing units. The massively parallel architecture of GPU allows achieving higher computing power than the power of CPU. Implementation of algorithms, written in the Julia programming language, for inference, learning, sampling and marginalization is described.

Key words: backpropagation, Butterfly decomposition, CUDA, GPU, inference, Julia, machine learning, marginalization, parallelization, probability density, probability distribution, sampling, SPTN, SVD decomposition

Obsah

Úvod **11**

1 Sítě typu Sum-Product-Transform **13**

1.1 Definice . . . 13
1.1.1 Listový uzel . . . 15
1.1.2 Součtový uzel . . . 15
1.1.3 Součinový uzel . . . 15
1.1.4 Transformační uzel . . . 16
1.2 Regularizace pro GPU implementaci . . . 16
1.2.1 Realizace listových uzlů . . . 16
1.2.2 Realizace transformačních uzlů . . . 17
1.2.3 Restrikce struktury sítě . . . 20
1.2.4 Zjednodušení poslední transformační vrstvy . . . 23
1.3 Učení SPTN . . . 23
1.3.1 Zpětná propagace . . . 23
1.3.2 Zpětná propagace pro SPTN . . . 25

2 Implementace **31**

2.1 GPU, CUDA a Julia . . . 31
2.2 Datová struktura SPTN . . . 33
2.3 Učení . . . 34
2.3.1 Značení v pseudo-algoritmech . . . 34
2.3.2 Inference . . . 34
2.3.3 Zpětná propagace . . . 42
2.4 Mapování stromů . . . 47
2.5 Inference vzhledem ke stromu . . . 51
2.6 Samplování . . . 52
2.7 Marginalizace . . . 53

3 Výsledky **57**

3.1 Paralelní urychlení . . . 57
3.2 Samplování . . . 59

Závěr **61**

Úvod

Modelování a manipulace se složitými pravděpodobnostními distribucemi jsou hlavní cíle strojového učení. Jejich důležitost plyne z faktu, že pravděpodobnostní modely mohou být chápány jako víceúčelové nástroje, což umožňuje řešení mnoha problémů strojového učení pomocí pravděpodobnostní inference. Pro tento účel lze použít sítě typu Sum-Product-Transform (zkráceně SPTN). SPTN jsou rozšířením sítí typu Sum-Product (SPN) pomocí nového typu uzlu, nazvaného transformační uzel. Toto rozšíření umožňuje SPTN lépe využít symetrie v distribucích, pro jejichž reprezentaci je použita. Transformační uzel může být omezen na pouze afinní transformace, a matici popisující lineární část této transformace lze uložit pomocí SVD rozkladu, což umožňuje snadnou invertovatelnost transformace, výpočet determinantu jejího jakobiánu a derivací vůči němu. SPTN navíc nejsou na rozdíl od SPN omezeny dimenzí dat [15]. Toto umožňuje konstruovat hluboké sítě pro popis složitých pravděpodobnostních distribucí. S velikostí sítě ovšem roste i výpočetní náročnost, a následně i čas potřebný pro naučení pravděpodobnostní distribuce z naměřených dat a následnou manipulaci s touto distribucí.

Tato práce se zabývá implementací SPTN pro architekturu grafických karet (graphics processing unit, GPU). Práce je rozdělena do tří kapitol. První kapitola obsahuje obecný popis SPTN, jejich regularizaci do tvaru vhodného pro implementaci pro architekturu GPU, a popis zpětně propagačního algoritmu použitého pro učení SPTN. Druhá kapitola obsahuje popis implementace SPTN pro GPU. Tato implementace zahrnuje inferenci, pravidla pro zpětné derivace pro jednotlivé funkce umožňující učení sítě, funkce umožňující generovat náhodné vzorky ze získané pravděpodobnostní distribuce a funkce umožňující vypočítat marginální rozdělení reprezentované distribuce. Třetí kapitola se skládá z měření paralelního urychlení a demonstrace generování náhodných vzorků z naučené SPTN.

Kapitola 1

Sítě typu Sum-Product-Transform

1.1 Definice

Síť typu Sum-Product-Transform je určena pro reprezentaci hustoty pravděpodobnosti. Využívá k tomu směsi, sdružení a transformace jednoduchých hustot pravděpodobností (např. Gaussovských).

Definice 1.1.1. Potenční množinou množiny M rozumíme množinu, která obsahuje všechny podmnožiny M. Značíme

$$2^M = \{N \,|\, N \subset M\}\,. \tag{1.1}$$

Definice 1.1.2. Nechť $G = (V, E)$ je orientovaný acyklický graf a $v \in V$ jeho libovolný vrchol. Pokud existuje orientovaná hrana z vrcholu v do vrcholu $w \in V$, tak w je potomek v. Množinu všech potomků v značíme

$$ch\,(v) = \{w \in V \,|\, (v, w) \in E\}\,. \tag{1.2}$$

Podobně, pokud existuje orientovaná hrana z vrcholu w do vrcholu v, tak w je rodič v. Množinu všech rodičů v značíme

$$p\,(v) = \{w \in V \,|\, (w, v) \in E\}\,. \tag{1.3}$$

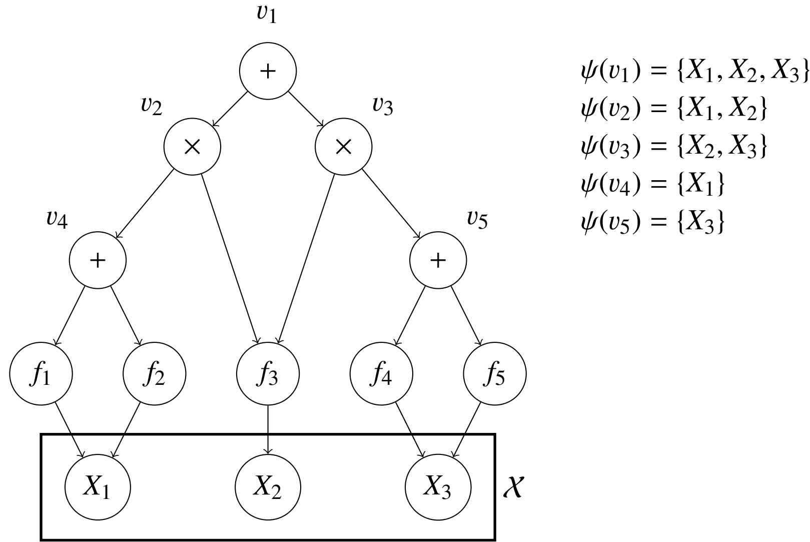
Definice 1.1.3. Nechť $G = (V, E)$ je orientovaný acyklický graf. Poté vrchol $v \in V$ je kořen, pokud není potomkem žádného vrcholu V. Množinu všech kořenů značíme

$$r(G) = \{v \in V \,|\, p\,(v) = \emptyset\}\,. \tag{1.4}$$

Podobně, vrchol v je list, pokud není rodičem žádného vrcholu V. Množinu všech listů značíme

$$l(G) = \{v \in V \,|\, ch\,(v) = \emptyset\}\,. \tag{1.5}$$

Definice 1.1.4. Nechť X je množina náhodných veličin. Poté sítí typu Sum-Product-Transform (zkráceně SPTN) nazveme uspořádanou trojici (G, ψ, θ), kde $G = (V, E)$ je orientovaný acyklický graf, $\psi : V \to 2^X$



Obrázek 1.1: Příklad SPTN.

je rozsahová funkce a θ je množina parametrů. Graf G obsahuje listové uzly, součtové uzly, součinové uzly a transformační uzly (definovány později). Rozsahová funkce ψ musí splňovat následující vlastnosti

$$(\forall v \in V \setminus l(G))\left(\psi(v) = \cup_{w \in ch(v)} \psi(w)\right), \tag{1.6}$$

$$(\forall v \in r(G))\left(\psi(v) = \mathcal{X}\right), \tag{1.7}$$

$$(\forall \text{ součinové uzly } p \in V)\left(\cap_{v \in ch(p)} \psi(v) = \emptyset\right), \tag{1.8}$$

$$(\forall \text{ součtové uzly } s \in V)\left(\forall v \in ch(s)\right)\left(\psi(s) = \psi(v)\right). \tag{1.9}$$

Poznámka 1.1.4.1.

- Rozsahová funkce $\psi(v)$ vyjadřuje, na kterých náhodných veličinách z $\mathcal{X}$ pravděpodobnostní hustota reprezentovaná uzlem v explicitně závisí.
- Vlastnost (1.6) vyjadřuje, že pokud libovolný vrchol v explicitně závisí na náhodné veličině $X \in \mathcal{X}$, poté na ní závisí i jeho rodič.
- Vlastnost (1.7) vyjadřuje, že všechny kořeny G explicitně závisí na všech náhodných veličinách v $\mathcal{X}$.
- Vlastnost (1.8) vyjadřuje, že libovolní potomci libovolného součinového uzlu nemohou explicitně záviset na stejné náhodné veličině z $\mathcal{X}$.
- Vlastnost (1.9) vyjadřuje, že libovolný součtový uzel explicitně závisí na stejných náhodných veličinách z $\mathcal{X}$ jako všichni jeho potomci.

Příklad SPTN je zobrazen na Obrázku 1.1, kde součtové uzly jsou znázorněny pomocí +, součinové uzly pomocí × a transformační uzly pomocí f_i pro $i \in \{1, \dots, 5\}$.

Graf SPTN nemusí být nutně strom. Pokud stromem není, tak dochází ke sdílení parametrů, jelikož některé uzly mají více než jednoho rodiče. Na obrázku 1.1 je sdílen například transformační uzel reprezentující transformaci f_3, jelikož je potomkem jak uzlu v_2, tak uzlu v_3.

1.1.1 Listový uzel

Listový uzel reprezentuje hustotu pravděpodobnosti p náhodné veličiny $X_i \in \mathcal{X}$ s parametry θ, což lze zapsat jako

$$p_L(x) = p(x|\theta). \tag{1.10}$$

Logaritmus věrohodnosti lze zapsat jako

$$\log\left(p_L(x)\right) = \log\left(p(x|\theta)\right). \tag{1.11}$$

Rozsahová funkce pro listový uzel splňuje

$$\psi(u) = X_i. \tag{1.12}$$

1.1.2 Součtový uzel

Součtový uzel reprezentuje směs hustot pravděpodobnosti reprezentovaných jeho potomky. Součtový uzel se skládá z n potomků reprezentujících hustoty pravděpodobnosti $p_1, p_2, \dots, p_n$ a uspořádané n-tice reálných čísel $a = (\alpha_1, \alpha_2, \dots, \alpha_n)$, které určují váhy jednotlivých potomků. Aby součtový uzel reprezentoval hustotu pravděpodobnosti, musí být všechny váhy nezáporné, a jejich součet musí být roven jedné. Toho můžeme dosáhnou pomocí funkce

$$\operatorname{softmax}(a)_i = \frac{\exp(\alpha_i)}{\sum_{j=1}^{n} \exp\left(\alpha_j\right)}, \quad \forall i \in \{1, \dots, n\}. \tag{1.13}$$

Jednotlivé váhy definujeme jako

$$w_i = \operatorname{softmax}(a)_i. \tag{1.14}$$

Poté lze definovat hustotu pravděpodobnosti reprezentovanou součtovým uzlem jako

$$p_\Sigma(x) = \sum_{i=1}^{n} w_i p_i(x). \tag{1.15}$$

Logaritmus věrohodnosti lze zapsat jako

$$\log\left(p_\Sigma(x)\right) = \log \sum_{i=1}^{n} \exp\left(\log\left(w_i\right) + \log\left(p_i(x)\right)\right). \tag{1.16}$$

1.1.3 Součinový uzel

Součinový uzel reprezentuje sdružení marginálních hustot pravděpodobností reprezentovaných jeho potomky do jedné hustoty pravděpodobnosti. Součinový uzel se skládá z n potomků, které reprezentují hustoty pravděpodobnosti $p_1, p_2, \dots, p_n$. Hustotu pravděpodobnosti reprezentovanou součinovým uzlem definujeme jako

$$p_\Pi(x) = \prod_{i=1}^{n} p_i(x_i), \tag{1.17}$$

kde x_i jsou části vektoru x příslušející jednotlivým potomkům. Logaritmus věrohodnosti lze zapsat jako

$$\log\left(p_\Pi(x)\right) = \sum_{i=1}^{n} \log\left(p_i(x_i)\right). \tag{1.18}$$

1.1.4 Transformační uzel

Transformační uzel reprezentuje transformaci hustoty pravděpodobnosti. Transformační uzel se skládá z právě jednoho potomka, který reprezentuje hustotu pravděpodobnosti p_{ch}, a invertovatelné funkce g, která reprezentuje inverzi transformační funkce. Hustotu pravděpodobnosti reprezentovanou transformačním uzlem definujeme jako

$$p_T(x_0) = p_{ch}(g(x_0)) \left| \det \left(\frac{dg}{dx}(x_0) \right) \right|. \tag{1.19}$$

Logaritmus věrohodnosti lze zapsat jako

$$\log\left(p_T(x_0)\right) = \log\left(p_{ch}(g(x_0))\right) + \log\left(\left| \det \left(\frac{dg}{dx}(x_0) \right) \right| \right). \tag{1.20}$$

1.2 Regularizace pro GPU implementaci

SPTN popsaná v sekci 1.1 je velmi obecná, a implementace v plné obecnosti na GPU by byla velmi obtížná. V této práci bude implementována pouze malá podmnožina všech přípustných SPTN. V obecné SPTN může být listový uzel tvořen libovolnou hustotou pravděpodobnosti, implementovány budou pouze listové uzly reprezentující standardní normální rozdělení, blíže popsané v sekci 1.2.1. Transformační uzel může obsahovat libovolnou invertovatelnou funkci, implementována bude pouze afinní transformace, blíže popsána v sekci 1.2.2. Následně budou omezeny i přípustné struktury sítě, blíže popsány v sekci 1.2.3.

1.2.1 Realizace listových uzlů

Pro implementaci SPTN na GPU budeme uvažovat pouze listové uzly reprezentující standardní s-rozměrné normální rozdělení.

Definice 1.2.1 (Vícerozměrné normální rozdělení)**.** Řekneme, že s-rozměrná náhodná veličina X má nedegenerované s-rozměrné normální rozdělení s parametry $\mu \in \mathbb{R}^s$ a pozitivně definitní maticí $\Sigma \in \mathbb{R}^{s,s}$, jestliže její hustota pravděpodobnosti má tvar

$$f_X(x) = \frac{\exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)}{\sqrt{(2\pi)^s |\Sigma|}}, \quad x \in \mathbb{R}^s. \tag{1.21}$$

Značíme $X \sim \mathcal{N}_s(\mu, \Sigma)$. Má-li s-rozměrné normální rozdělení parametry $\mu = 0$ a $\Sigma = I$, kde I je matice identity, tak jej nazveme standardní.

Poznámka 1.2.1.1. Hustota pravděpodobnosti standardního s-rozměrného náhodného rozdělení je rovna

$$f_X(x) = \frac{\exp\left(-\frac{1}{2}x^T x\right)}{\sqrt{(2\pi)^s}}, \quad x \in \mathbb{R}^s. \tag{1.22}$$

Logaritmus hustoty pravděpodobnosti s-rozměrného normálního rozdělení je roven

$$\log\left(f_X(x)\right) = -\frac{1}{2}\left(x^T x + s\log(2\pi)\right). \tag{1.23}$$

Všechny listové uzly tedy budou obsahovat funkci určenou rovnicí (1.23). Tato funkce nemá žádné parametry s výjimkou dimenze s, ta je ovšem ale pevně spojená se strukturou sítě.

1.2.2 Realizace transformačních uzlů

Pro implementaci SPTN na GPU budeme uvažovat pouze transformační uzly obsahující afinní transformace.

Definice 1.2.2 (Afinní transformace)**.** Funkci $f : \mathbb{R}^s \to \mathbb{R}^s$ nazveme afinní transformací, pokud lze zapsat ve tvaru

$$f(x) = a(x) + b, \quad x \in \mathbb{R}^s, \tag{1.24}$$

kde $a \in \mathcal{L}(\mathbb{R}^s)$ je lineární bijekce a $b \in \mathbb{R}^s$.

Poznámka 1.2.2.1. Funkce $f : \mathbb{R}^s \to \mathbb{R}^s$ je afinní transformace právě tehdy, když lze zapsat pomocí regulární matice $A \in \mathbb{R}^{s,s}$ a vektoru $b \in \mathbb{R}^s$ ve tvaru

$$f(x) = Ax + b, \quad x \in \mathbb{R}^s. \tag{1.25}$$

Dosadíme-li afinní transformaci (1.25) do rovnice (1.19), obdržíme

$$p_T(x_0) = p_{ch}(Ax_0 + b) \left| \det \left(\frac{d(Ax + b)}{dx}(x_0) \right) \right|. \tag{1.26}$$

Jakobián v rovnici (1.26) lze zjednodušit

$$\left(\frac{d(Ax + b)}{dx}(x_0) \right)_{i,j} = \frac{d\left(A_{ij}x_j + b_i\right)}{dx_j}(x_0)_i = A_{ij}, \quad \forall i, j \in \{1, \ldots, n\}. \tag{1.27}$$

Dosazením (1.27) do (1.26) získáme

$$p_T(x_0) = p_{ch}(Ax_0 + b) \left|\det A\right|. \tag{1.28}$$

Výpočet determinantu matice A je velmi náročný, navíc pro učení sítě je potřeba spočíst i derivace determinantu vůči prvkům matice A. Tento problém je vyřešen použitím singulárního rozkladu matice A, který umožní efektivní výpočet determinantu matice A včetně derivací vůči parametrům.

Definice 1.2.3 (Ortogonální matice [7])**.** Matici $Q \in \mathbb{R}^{n,n}$, kde $n \in \mathbb{N}$, nazveme ortogonální, pokud

$$Q^T Q = I. \tag{1.29}$$

Poznámka 1.2.3.1. Pro každou ortogonální matici $Q \in \mathbb{R}^{n,n}$ platí

$$\left|\det Q\right| = 1. \tag{1.30}$$

Důkaz. Z (1.29) získáváme

$$1 = \det I = \det\left(Q^T Q\right) = (\det Q)^2, \tag{1.31}$$

tedy nutně platí $\det Q = \pm 1$, a po aplikaci absolutní hodnoty $|\det Q| = 1$. □

Tvrzení 1.2.1 (Singulární rozklad [7])**.** Je-li $A \in \mathbb{R}^{n,n}$, poté existují ortogonální matice $U \in \mathbb{R}^{n,n}$ a $V \in \mathbb{R}^{n,n}$, a diagonální matice $D \in \mathbb{R}^{n,n}$ takové, že

$$A = UDV. \tag{1.32}$$

Dosazením (1.32) do (1.28) a využitím (1.30) získáme

$$\begin{aligned} p_T(x_0) &= p_{ch}(UDVx_0 + b)\,|\det(UDV)| \\ &= p_{ch}(UDVx_0 + b)\,|\det U \det D \det V| \\ &= p_{ch}(UDVx_0 + b)\,|\det D|\,. \end{aligned} \tag{1.33}$$

Logaritmus hustoty pravděpodobnosti má tedy tvar

$$\log\left(p_T(x_0)\right) = \log\left(p_{ch}(UDVx_0 + b)\right) + \sum_{i=1}^{n} \log\left(|d_{ii}|\right), \tag{1.34}$$

kde d_{ii} značí prvky matice D.

Výpočet determinantu diagonální matice a derivací vůči parametrům je snadné.

Nyní potřebujeme parametrizaci ortogonálních matic U a V, u které půjde navíc spočíst derivace vůči parametrům. Pro tento účel využijeme Givensovy rotace a Butterfly rozklad.

Definice 1.2.4 (Givensova rotace [7])**.** Nechť $i, j \in \{1, \dots, n\}$, $i < j$, kde $n \in \mathbb{N}$ a nechť $\theta \in \mathbb{R}$. Matici ve tvaru

$$G(i, j, \theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ n \end{matrix}, \qquad \begin{matrix} & & i & & j & & n \end{matrix} \tag{1.35}$$

kde $c = \cos(\theta)$ a $s = \sin(\theta)$, nazveme Givensovou rotací dimenze n.

Při násobení libovolného vektoru $x \in \mathbb{R}^n$ Givensovou rotací $G(i, j, \theta)$ jsou změněny pouze komponenty vektoru x na pozicích i a j, zbytek zůstává stejný. Tohoto faktu můžeme využít při paralelizaci.

Předpokládejme, že chceme vynásobit vektor x dvěma Givensovými rotacemi $G(i_1, j_1, \theta_1)$ a $G(i_2, j_2, \theta_2)$. Pokud $\{i_1, j_1\} \cap \{i_2, j_2\} = \emptyset$, poté obě Givensovy rotace mění různé komponenty x, a tedy jimi můžeme

násobit v libovolném pořadí, nebo i zároveň. Například, pro $n = 4$, $i_1 = 1$, $j_1 = 2$, $i_2 = 3$, $j_2 = 4$

$$G(i_1, j_1, \theta_1)\, G(i_2, j_2, \theta_2) = \begin{pmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_2 & -s_2 \\ 0 & 0 & s_2 & c_2 \end{pmatrix} =$$

$$G(i_2, j_2, \theta_2)\, G(i_1, j_1, \theta_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_2 & -s_2 \\ 0 & 0 & s_2 & c_2 \end{pmatrix} \begin{pmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$\Gamma(\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{\theta}) = \begin{pmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & c_2 & -s_2 \\ 0 & 0 & s_2 & c_2 \end{pmatrix}, \tag{1.36}$$

kde $c_i = \cos(\theta_i)$, $s_i = \sin(\theta_i)$ pro $i \in \{1, 2\}$.

Možnosti násobení více Givensovými rotacemi zároveň velmi dobře využívá Butterfly rozklad.

Tvrzení 1.2.2 (Butterfly rozklad [16])**.** Ortogonální matici $U \in \mathbb{R}^{k,k}$, kde $k \in \{2^n \mid n \in \mathbb{N}\}$, lze rozložit do tvaru

$$U = \prod_{i=1}^{k-1} \prod_{j=1}^{k/2} G\left(\alpha(i, j), \alpha(i, j) + \kappa(i), \theta^{(i,j)}\right), \tag{1.37}$$

kde

$$\kappa(i) = 2\hat{}\left(\min_{m \in \mathbb{N}} \{\operatorname{mod}(i, 2^m) \neq 0\} - 1\right), \tag{1.38}$$

a

$$\alpha(i, j) = 2\kappa(i) \left\lfloor \frac{j-1}{\kappa(i)} \right\rfloor + \operatorname{mod}(j - 1, \kappa(i)) + 1. \tag{1.39}$$

Navíc, indexy rotací ve vnitřním produktu (1.37) mají prázdný průnik, tudíž rozklad lze zapsat ve tvaru

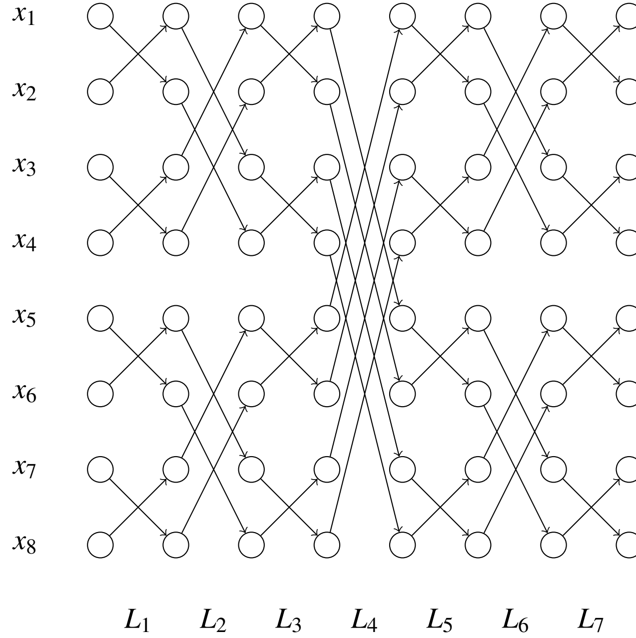
$$U = \prod_{i=1}^{k-1} \Gamma\left(\boldsymbol{\alpha}^{(i)}, \boldsymbol{\alpha}^{(i)} + \kappa(i), \boldsymbol{\theta}^{(i)}\right). \tag{1.40}$$

Tuto dekompozici lze znázornit pomocí flow grafu (např. Obrázek 1.2), na kterém je každá rotace reprezentována jedním úhlopříčným křížkem, jehož konce symbolizují indexy rotace. Každý sloupec rotací ve flow grafu koresponduje s jednou Γ z (1.40). Tyto sloupce, označené L_1 až L_7, nazveme *Butterfly vrstvy*.

Butterfly dekompozice lze použít i v případě, kdy velikost k ortogonální matice $U \in \mathbb{R}^{k,k}$ není mocnina dvou. Řešením je doplnit rozkládanou matici řádky a sloupci identity na nejbližší vyšší mocninu dvou, a poté provést Butterfly rozklad. Tento postup koresponduje s rozložením rozšířené matice, a následným odstraněním všech Givensových rotací v (1.37), u kterých by alespoň jeden index byl mimo rozsah pro původní velikost matice.

Nevýhodou tohoto postupu je, že využívá více parametrů, než je nezbytně nutné.

Tvrzení 1.2.3 (Stupně volnosti orthogonální matice)**.** Ortogonální matice $Q \in \mathbb{R}^{n,n}$, kde $n \in \mathbb{N}$, má $\frac{n(n-1)}{2}$ stupňů volnosti.



Obrázek 1.2: Flow graf pro Butterfly dekompozici ortogonální matice $U \in \mathbb{R}^{8,8}$.

Důkaz. Nechť $v_1, v_2, \ldots, v_n \in \mathbb{R}^n$ značí sloupce Q. Podmínka (1.29) lze ekvivalentně přepsat na

$$\left\langle v_i, v_j \right\rangle = 0, \quad \forall i, j \in \{1, 2, \ldots, n\}, \quad i < j, \tag{1.41}$$

$$\langle v_i, v_i \rangle = 1, \quad \forall i \in \{1, 2, \ldots, n\}. \tag{1.42}$$

Podmínka (1.41) vytváří

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \tag{1.43}$$

vazeb, a podmínka (1.42) vytváří dalších n vazeb. Tedy, ortogonální matice Q má

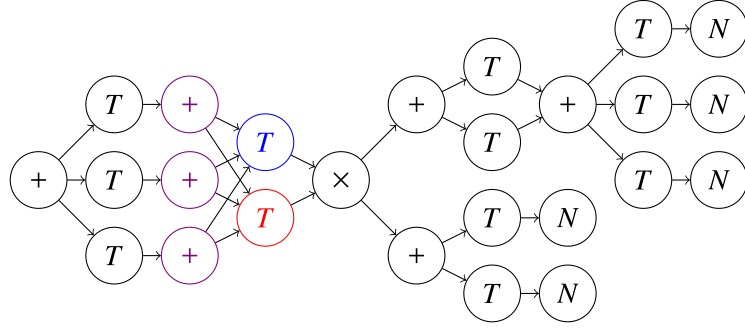
$$n^2 - \frac{n(n-1)}{2} - n = \frac{n(n-1)}{2} \tag{1.44}$$

stupňů volnosti. □

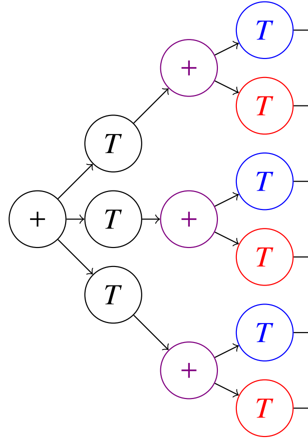
Například pro ortogonální matici $U \in \mathbb{R}^{5,5}$ získáme z Butterfly rozkladu 13 parametrů, což lze zjistit z obrázku 1.2 vynecháním složek vektoru x_6, x_7, x_8 a rotací modifikujících alespoň jeden z těchto prvků, a spočtením zbylých rotací. Tato matice má ovšem pouze 10 stupňů volnosti. Druhou nevýhodou je nerovnoměrný počet Givensových rotací v jednotlivých vrstvách (někdy i pouze jedna), což vede na obtíže s využitím veškerého dostupného výpočetního výkonu při paralelní implementaci.

1.2.3 Restrikce struktury sítě

Pro implementaci na GPU nebudeme uvažovat síť zcela obecné struktury. Budeme se zabývat implementací sítí skládající se z vrstev složených ze součtových uzlů následovaných transformačními uzly. Navíc všechny transformační uzly v jedné vrstvě mohou mít jako potomky součinové uzly. Příklad takovéto sítě lze vidět na obrázku 1.3.



Obrázek 1.3: Příklad SPTN splňující restrikce. Parametry červeně a modře zvýrazněných transformačních uzlů jsou sdíleny fialově zvýrazněnými součtovými uzly.



Obrázek 1.4: Příklad části SPTN z obrázku 1.3 bez sdílení parametrů.

Na obrázku 1.3 nejsou žádné vrstvy, které by nesdílely parametry. Příklad části sítě z obrázku 1.3 před součinovým uzlem bez sdílených parametrů je znázorněn na obrázku 1.4. Důvodem je, že vrstvy bez sdílených parametrů lze zjednodušit, aniž by se snížila schopnost sítě reprezentovat hustotu pravděpodobnosti, bez zvýšení výpočetní náročnosti. Toto zjednodušení je založeno na skutečnosti, že složení dvou afinních transformací je opět afinní transformace.

Tvrzení 1.2.4 (Složení afinních transformací)**.** Nechť funkce $f, g : \mathbb{R}^s \to \mathbb{R}^s$, kde $s \in \mathbb{N}$, jsou afinní transformace. Poté existuje afinní transformace $h : \mathbb{R}^s \to \mathbb{R}^s$ tak, že platí

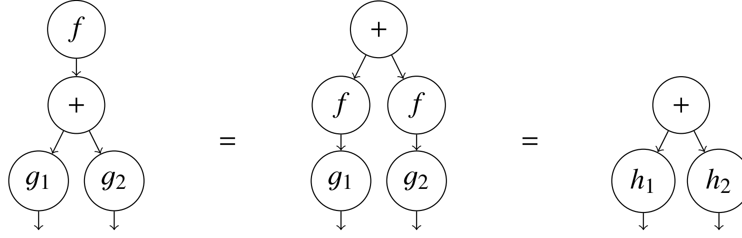
$$h = f \circ g. \tag{1.45}$$

Důkaz. Podle poznámky 1.2.2.1 můžeme reprezentovat funkce f, g jako

$$f(x) = A_f x + b_f, \tag{1.46}$$
$$g(x) = A_g x + b_g. \tag{1.47}$$

Složením (1.46) a (1.47) získáme

$$(f \circ g)(x) = A_f \left(A_g x + b_g\right) + b_f = A_f A_g x + A_f b_g + b_f = A_h x + b_h = h(x). \tag{1.48}$$



Obrázek 1.5: Příklad zjednodušení sítě sloučením transformačních uzlů pro $n = 2$.

Kde jsme označili $A_h = A_f A_g$, což je jistě regulární matice, a $b_h = A_f b_g + b_f$. Opět dle poznámky 1.2.2.1 získáváme, že h je afinní transformace. □

Dále budeme potřebovat fakt, že součin absolutních hodnot determinantů jakobiánu dvou afinních transformací je absolutního hodnota determinantu jakobiánu jejich složení.

Tvrzení 1.2.5. Nechť funkce $f, g : \mathbb{R}^s \to \mathbb{R}^s$, kde $s \in \mathbb{N}$, jsou afinní transformace. Poté platí

$$\left|\det\left(\frac{df}{dx}\right)\right|\left|\det\left(\frac{dg}{dx}\right)\right| = \left|\det\left(\frac{d(f \circ g)}{dx}\right)\right|. \tag{1.49}$$

Důkaz. Využitím poznámky 1.2.2.1 a úpravy (1.27) získáme

$$\left|\det\left(\frac{df}{dx}\right)\right|\left|\det\left(\frac{dg}{dx}\right)\right| = \left|\det A_f\right|\left|\det A_g\right| = \left|\det A_f A_g\right| = \left|\det\left(\frac{d(f \circ g)}{dx}\right)\right|. \tag{1.50}$$

□

Uvažujme síť skládající se z transformačního uzlu s afinní transformací f, jehož potomkem je součtový uzel, který má n transformačních uzlů jako potomky, každý reprezentující afinní transformaci g_i pro $i \in \{1, \ldots, n\}$. Tato síť je pro $n = 2$ znázorněna na obrázku 1.5. Hustota pravděpodobnosti reprezentovaná první sítí na obrázku 1.5 lze vyjádřit a následně pomocí (1.50) upravit jako

$$\begin{aligned} p(x_0) &= \left(\sum_{i=1}^{n} w_i p_i(g_i(f(x_0))) \left|\det\left(\frac{dg_i}{dx} x_0\right)\right|\right)\left|\det\left(\frac{df}{dx} x_0\right)\right| \\ &= \sum_{i=1}^{n} w_i p_i(g_i(f(x_0))) \left|\det\left(\frac{dg_i}{dx} x_0\right)\right|\left|\det\left(\frac{df}{dx} x_0\right)\right| \\ &= \sum_{i=1}^{n} w_i p_i(g_i(f(x_0))) \left|\det\left(\frac{dh_i}{dx} x_0\right)\right|, \end{aligned} \tag{1.51}$$

kde jsme označili $h_i = g_i \circ f$ a symbolem p_i hustotu pravděpodobnostmi reprezentovanou transformačními uzly ve spodní vrstvě.

Pokud by tedy v síti existovaly dvě vrstvy součtových a transformačních uzlů po sobě bez sdílených parametrů, je možné je sjednotit do jedné beze ztráty schopnosti reprezentace hustoty pravděpodobnosti a zmenšit tak výpočetní náročnost. Takovéto sítě tedy uvažovat nebudeme.

1.2.4 Zjednodušení poslední transformační vrstvy

Jelikož používáme pouze listové uzly s normálním rozdělením, a transformační uzly s afinní transformací rozloženou pomocí singulárního rozkladu, máme možnost zjednodušit poslední transformační vrstvu. Dosadíme-li do výrazu pro věrohodnost listového uzlu (1.23) výraz $x = UDVy + b$, získáme

$$\log\left(f_X\left(UDVy + b\right)\right) = -\frac{1}{2}\left(\left(UDVy + b\right)^T\left(UDVy + b\right) + s\log\left(2\pi\right)\right). \tag{1.52}$$

Naproti tomu vynecháním ortogonální matice V a dosazením $x = DVy + \tilde{b}$, získáme

$$\log\left(f_X\left(DVy + \tilde{b}\right)\right) = -\frac{1}{2}\left(\left(DVy + \tilde{b}\right)^T\left(DVy + \tilde{b}\right) + s\log\left(2\pi\right)\right). \tag{1.53}$$

Porovnáním pravých stran (1.52) a (1.53) a využitím (1.29) získáme

$$\begin{aligned}
-\frac{1}{2}\left(\left(UDVy + b\right)^T\left(UDVy + b\right) + s\log\left(2\pi\right)\right) &= -\frac{1}{2}\left(\left(DVy + \tilde{b}\right)^T\left(DVy + \tilde{b}\right) + s\log\left(2\pi\right)\right)\\
\left(UDVy + b\right)^T\left(UDVy + b\right) &= \left(DVy + \tilde{b}\right)^T\left(DVy + \tilde{b}\right)\\
\left(UDVy\right)^T UDVy + \left(UDVy\right)^T b + b^T UDVy + b^T b &= \left(DVy\right)^T DVy + \left(DVy\right)^T \tilde{b} + \tilde{b}^T DVy + \tilde{b}^T\tilde{b}\\
\left(DVy\right)^T U^T UDVy + \left(UDVy\right)^T b + b^T UDVy + b^T b &= \left(DVy\right)^T DVy + \left(DVy\right)^T \tilde{b} + \tilde{b}^T DVy + \tilde{b}^T\tilde{b}\\
\left(DVy\right)^T U^T b + \left(U^T b\right)^T DVy + \left(U^T b\right)^T U^T b &= \left(DVy\right)^T \tilde{b} + \tilde{b}^T DVy + \tilde{b}^T\tilde{b}.
\end{aligned}$$

Vidíme tedy, že pokud platí $U^T b = \tilde{b}$, tak si jsou obě strany rovny, a tudíž se rovnají i věrohodnosti. Jelikož vektor b je parametr, a může nabývat libovolných hodnot, tak vynecháním matice U v singulárním rozkladu afinní transformace poslední vrstvy transformačních uzlů neztratí síť žádnou schopnost reprezentace hustoty pravděpodobnosti. Naopak toto vede na úspory ve výpočetní náročnosti, jelikož není třeba násobit a následně počítat derivace vůči prvkům matice U.

1.3 Učení SPTN

Cílem sítě je reprezentovat hustotu pravděpodobnosti. Pokud máme množinu pozorování a chceme, aby SPTN reprezentovala hustotu pravděpodobnosti, ze které tato pozorování pochází, musíme být schopni určit parametry uzlů tak, aby byla maximalizována věrohodnostní funkce. Pro minimalizování ztrátové funkce, kterou bude v našem případě záporně vzatá hodnota logaritmu věrohodnosti, lze použít např. stochastický sestup po gradientu, nebo ADAM [12]. Obě tyto metody vyžadují znalost derivací ztrátové funkce vzhledem k parametrům SPTN. SPTN mohou mít mnoho uzlů, tudíž ztrátová funkce může být velmi složitá funkce s velkým množstvím parametrů. Pro spočtení derivací využijeme zpětně propagační algoritmus [1].

1.3.1 Zpětná propagace

Pokud bychom počítali derivace bez zpětně propagačního algoritmu pouze s využitím řetězového pravidla, rostla by výpočetní náročnost exponenciálně s počtem vnořených funkcí s parametry. Zpětně propagační algoritmus umožňuje tuto sumu efektivně spočíst s pomocí dynamického programování. Algoritmus se skládá ze dvou částí, nazvaných dopředná a zpětná fáze.

V dopředné fázi spočteme ztrátovou funkci se stávajícími parametry SPTN. Poté v zpětné fázi spočteme derivace ztrátové funkce vzhledem k parametrům SPTN. Tyto derivace poté využijeme pro optimalizaci parametrů sítě. Ilustrujme na jednoduchém příkladu jak funguje zpětná fáze.

Uvažujme složenou funkci $f(a, g(b, h(c, x)))$, kde $f, g, h : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, $x \in \mathbb{R}^n$ je vstupní vektor a $a, b, c \in \mathbb{R}^n$ jsou vektory parametrů. Naším cílem je spočíst gradient ztrátové funkce $L(f(a, g(b, h(c, x))))$ vzhledem k parametrům a, b a c.

Pro zpřehlednění zápisu budeme v následujících rovnicích používat značení

$$\begin{aligned} f &\equiv f(a, g(b, h(c, x))) \in \mathbb{R}^n, \\ g &\equiv g(b, h(c, x)) \in \mathbb{R}^n, \\ h &\equiv h(c, x) \in \mathbb{R}^n. \end{aligned}$$

Derivace $L(f)$ vzhledem k a lze spočíst pomocí řetězového pravidla

$$\frac{\partial L(f)}{\partial a} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \frac{\partial f_i}{\partial a}. \tag{1.54}$$

Pokračujme výpočtem derivací $L(f)$ vzhledem k b. Opětovným použitím řetězového pravidla získáme

$$\frac{\partial L(f)}{\partial b} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \frac{\partial f_i}{\partial b} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \sum_{j=1}^{n} \frac{\partial f_i}{\partial g_j} \frac{\partial g_j}{\partial b}. \tag{1.55}$$

Podobně lze spočíst derivace $L(f)$ vzhledem k c

$$\frac{\partial L(f)}{\partial c} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \frac{\partial f_i}{\partial c} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \sum_{j=1}^{n} \frac{\partial f_i}{\partial g_j} \sum_{k=1}^{n} \frac{\partial g_j}{\partial h_k} \frac{\partial h_k}{\partial c}. \tag{1.56}$$

Problém s tímto přístupem je, že počet členů v sumách (1.56) roste exponenciálně s počtem vnořených funkcí. Zpětná propagace tento problém řeší následujícím způsobem.

Označme

$$\Delta^{(f)} = \frac{\partial L(f)}{\partial f} \in \mathbb{R}^n \tag{1.57}$$

gradient ztrátové funkce L vzhledem k f. Nyní můžeme zapsat (1.54) jako

$$\frac{\partial L(f)}{\partial a} = \sum_{i=1}^{n} \Delta_i^{(f)} \frac{\partial f_i}{\partial a}. \tag{1.58}$$

Před výpočtem (1.55) nejprve spočteme

$$\Delta^{(g)} = \frac{\partial L(f)}{\partial g} = \sum_{i=1}^{n} \Delta_i^{(f)} \frac{\partial f_i}{\partial g} \in \mathbb{R}^n. \tag{1.59}$$

Gradient jsme zpětně propagovali za funkci f, a $\Delta^{(g)}$ nyní značí gradient ztrátové funkce L vzhledem k g. Tento gradient můžeme využít pro zjednodušení výpočtu (1.55)

$$\frac{\partial L(f)}{\partial b} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \sum_{j=1}^{n} \frac{\partial f_i}{\partial g_j} \frac{\partial g_j}{\partial b} = \sum_{j=1}^{n} \Delta_j^{(g)} \frac{\partial g_j}{\partial b}. \tag{1.60}$$

Nyní jsme použili pouze jednu sumu při výpočtu derivací ztrátové funkce L vzhledem k b, ale museli jsme spočíst další sumu za každý prvek $\Delta^{(g)}$. Pokud bychom derivace počítali podle (1.55), také bychom museli spočíst dvě vnořené sumy. Postup je zatím stejně výpočetně náročný. Zpětná propagace bude efektivnější až při výpočtu derivací ztrátové funkce vzhledem k c.

Nejprve zpětně propagujme gradient $\Delta^{(g)}$ za funkci g směrem k h

$$\Delta^{(h)} = \frac{\partial L(f)}{\partial h} = \sum_{i=1}^{n} \Delta_i^{(g)} \frac{\partial g_i}{\partial h} \in \mathbb{R}^n. \tag{1.61}$$

Nyní můžeme spočíst derivace

$$\frac{\partial L(f)}{\partial c} = \sum_{i=1}^{n} \frac{\partial L(f)}{\partial f_i} \sum_{j=1}^{n} \frac{\partial f_i}{\partial g_j} \sum_{k=1}^{n} \frac{\partial g_j}{\partial h_k} \frac{\partial h_k}{\partial c} = \sum_{j=1}^{n} \Delta_j^{(g)} \sum_{k=1}^{n} \frac{\partial g_j}{\partial h_k} \frac{\partial h_k}{\partial c} = \sum_{k=1}^{n} \Delta_k^{(h)} \frac{\partial h_k}{\partial c}. \tag{1.62}$$

Podobně jako v (1.60) jsme museli spočíst dvě vnořené sumy (navíc ku předešlým výpočtům), abychom získali derivace ztrátové funkce vzhledem k c. Pokud bychom derivace počítali podle (1.55), tak bychom potřebovali spočíst tři vnořené sumy. Pokud by funkce f byla složitější s více vnořenými funkcemi, tak by naivní postup bez zpětné propagace byl ještě výpočetně náročnější, rostoucí exponenciálně s počtem vnořených funkcí. Díky zpětně propagačnímu algoritmu každá další vnořená funkce potřebuje pouze zpětnou propagaci gradientu Δ a výpočet jediné sumy.

1.3.2 Zpětná propagace pro SPTN

Zpětně propagační algoritmus je rozdělen na dvě části. V dopředné fázi spočteme logaritmus věrohodnosti pozorování, a následně v zpětné fázi spočteme derivace ztrátové funkce vzhledem ke všem parametrům sítě.

1.3.2.1 Listový uzel

Listový uzel nemá žádné parametry, není tedy vůči čemu počítat derivace. Jediné co potřebujeme vědět, je jak zpětně propagovat gradient.

Listový uzel implementuje funkci

$$l(x) = -\frac{1}{2}\left(x^T x + s \log(2\pi)\right), \tag{1.63}$$

kde $x \in \mathbb{R}^n$ značí vektor pozorování vstupující do listového uzlu. Gradient příchozí během zpětné propagace má tvar

$$\Delta^{(l(x))} = \frac{\partial L}{\partial l(x)} \in \mathbb{R}, \tag{1.64}$$

kde L značí ztrátovou funkci. Gradient vzhledem k x nalezneme snadno pomocí řetězového pravidla

$$\Delta^{(x)} = \frac{\partial L}{\partial x} = \frac{\partial L}{\partial l(x)} \frac{\partial l(x)}{\partial x} = \Delta^{(l(x))} \frac{\partial l(x)}{\partial x} \in \mathbb{R}^n. \tag{1.65}$$

Druhý člen (1.65) lze zjednodušit díky znalosti tvaru funkce l (1.63)

$$\frac{\partial l(x)}{\partial x_i} = \frac{\partial}{\partial x_i} - \frac{1}{2}\left(\sum_{j=1}^{n} x_j^2 + s \log(2\pi)\right) = -\sum_{j=1}^{n} \delta_{ij} x_i = -x_i, \quad \forall i \in \{1, \dots, n\}. \tag{1.66}$$

Dosazením (1.66) do (1.65) získáme

$$\Delta^{(x)} = -\Delta^{(l(x))} x \in \mathbb{R}^n. \tag{1.67}$$

1.3.2.2 Součtový uzel

Parametry součtového uzlu jsou váhy jednotlivých komponent. Potřebujeme tedy umět spočíst derivace ztrátové funkce vůči vahám komponent, a vědět jak zpětně propagovat gradient.

Součtový uzel implementuje funkci

$$s(x) = \log \sum_{i=1}^{m} \exp\left(\log\left(\mathrm{softmax}(\alpha_i)\right) + \log\left(p_i\left(x\right)\right)\right), \tag{1.68}$$

kde $m \in \mathbb{N}$ je počet komponent směsi reprezentované součtovým uzlem, p_i jsou jednotlivé komponenty, a $\alpha_i \in \mathbb{R}$ jsou parametry vah jednotlivých komponent. Tento zápis můžeme zjednodušit zavedením funkce log-sum-exp, značeno

$$\operatorname*{lse}_{i=1}^{n}(x_i) = \log \sum_{i=1}^{n} \exp(x_i), \tag{1.69}$$

kde $x \in \mathbb{R}^n$. Odvoď me derivaci funkce *lse*

$$\frac{\partial}{\partial x_j} \operatorname*{lse}_{i=1}^{n}(x_i) = \frac{\partial}{\partial x_j} \log \sum_{i=1}^{n} \exp(x_i) = \left(\sum_{i=1}^{n} \exp\left(x_i\right)\right)^{-1} \exp\left(x_j\right), \quad \forall j \in \{1, \ldots, n\}. \tag{1.70}$$

Rovnici (1.68) lze s použitím (1.69) zapsat jako

$$s(x) = \operatorname*{lse}_{i=1}^{m}\left(\alpha_i - \operatorname*{lse}_{j=1}^{m}\left(\alpha_j\right) + \log\left(p_i\left(x\right)\right)\right). \tag{1.71}$$

Příchozí gradient má tvar

$$\Delta^{(s(x))} = \frac{\partial L}{\partial s(x)} \in \mathbb{R}, \tag{1.72}$$

kde L značí ztrátovou funkci. Nejdříve spočteme derivace ztrátové funkce vzhledem k parametrům součtového uzlu

$$\frac{\partial L}{\partial \alpha_k} = \frac{\partial L}{\partial s(x)} \frac{\partial s(x)}{\partial \alpha_k} = \Delta^{(s(x))} \frac{\partial s(x)}{\partial \alpha_k}. \tag{1.73}$$

Pro lepší přehlednost označíme $\beta_i = \alpha_i - \mathrm{lse}_{j=1}^{m}\left(\alpha_j\right) + \log\left(p_i\left(x\right)\right)$, a následně upravíme výraz

$$\begin{aligned}
\frac{\partial s(x)}{\partial \alpha_k} &= \left(\sum_{i=1}^{m} \exp\left(\beta_i\right)\right)^{-1} \sum_{i=1}^{m} \left(\exp\left(\beta_i\right) \frac{\partial \beta_i}{\partial \alpha_k}\right) \\
&= \left(\sum_{i=1}^{m} \exp\left(\beta_i\right)\right)^{-1} \sum_{i=1}^{m} \left(\exp\left(\beta_i\right) \left(\delta_{ik} - \frac{\partial}{\partial \alpha_k} \operatorname*{lse}_{j=1}^{m}\left(\alpha_j\right)\right)\right) \\
&= \left(\sum_{i=1}^{m} \exp\left(\beta_i\right)\right)^{-1} \sum_{i=1}^{m} \left(\exp\left(\beta_i\right) \left(\delta_{ik} - \left(\sum_{j=1}^{m} \exp\left(\alpha_j\right)\right)^{-1} \exp\left(\alpha_k\right)\right)\right).
\end{aligned} \tag{1.74}$$

Dále potřebujeme umět zpětně propagovat gradient

$$
\begin{aligned}
\Delta^{(\log(p_k(x)))} &= \frac{\partial L}{\partial \log(p_k(x))} \\
&= \frac{\partial L}{\partial s(x)} \frac{\partial s(x)}{\partial \log(p_k(x))} \\
&= \Delta^{(s(x))} \left(\sum_{i=1}^{m} \exp(\beta_i) \right)^{-1} \sum_{i=1}^{m} \left(\exp(\beta_i) \frac{\partial \beta_i}{\partial \log(p_k(x))} \right) \\
&= \Delta^{(s(x))} \left(\sum_{i=1}^{m} \exp(\beta_i) \right)^{-1} \sum_{i=1}^{m} \exp(\beta_i)\, \delta_{ik} \\
&= \Delta^{(s(x))} \left(\sum_{i=1}^{m} \exp(\beta_i) \right)^{-1} \beta_k, \quad \forall k \in \{1, \dots, m\}.
\end{aligned} \tag{1.75}
$$

1.3.2.3 Součinový uzel

Součinový uzel nemá žádné parametry, není tedy vůči čemu počítat derivace. Potřebujeme pouze umět zpětně propagovat gradient.

Součinový uzel implementuje funkci

$$
q(x) = \sum_{i=1}^{n} \log(p_i(x_i)), \tag{1.76}
$$

kde $x \in \mathbb{R}^n$ značí vektor pozorování vstupující do součinového uzlu, $m \in \mathbb{N}$ je počet potomků součinového uzlu, x_i jsou části vektoru x příslušející jednotlivým potomkům p_i, kde $i \in \{1, \dots, m\}$. Příchozí gradient během zpětné propagace má tvar

$$
\Delta^{(q(x))} = \frac{\partial L}{\partial q(x)} \in \mathbb{R}, \tag{1.77}
$$

kde L značí ztrátovou funkci. Gradient vzhledem k $\log(p_i(x_i))$ nalezneme snadno pomocí řetězového pravidla

$$
\begin{aligned}
\Delta^{(\log(p_j(x_j)))} &= \frac{\partial L}{\partial \log\left(p_j(x_j)\right)} \\
&= \frac{\partial L}{\partial q(x)} \frac{\partial q(x)}{\partial \log\left(p_j(x_j)\right)} \\
&= \Delta^{(q(x))} \frac{\partial}{\partial \log\left(p_j(x_j)\right)} \sum_{i=1}^{n} \log(p_i(x_i)) \\
&= \Delta^{(q(x))} \sum_{i=1}^{n} \delta_{ij} \\
&= \Delta^{(q(x))}, \quad \forall j \in \{1, \dots, m\}.
\end{aligned} \tag{1.78}
$$

1.3.2.4 Transformační uzel

Většina parametrů sítě je uložena v transformačních uzlech. Potřebujeme tedy umět spočíst derivace vůči všem parametrům, a umět zpětně propagovat gradient.

Transformační uzel implementuje funkci

$$t(x) = \log\left(p_{ch}(UDVx + b)\right) + \sum_{i=1}^{n} \log\left(|d_{ii}|\right). \tag{1.79}$$

Pro transformační uzel bude výpočet probíhat ve dvou částech. Nejprve spočteme derivace vzhledem k prvkům matice D pocházejících z determinantu jakobiánu a zpětně propagujme gradient přes sumu k výrazu $\log\left(p_{ch}(UDVx + b)\right)$. Ve druhé části, poté co zpětnou propagací přes ostatní typy uzlů získáme gradient vzhledem k $UDVx+b$, spočteme derivace ke všem parametrům transformace, a následně zpětně propagujme gradient k x.

Příchozí gradient během první části zpětné propagace má tvar

$$\Delta^{(t(x))} = \frac{\partial L}{\partial t(x)} \in \mathbb{R}, \tag{1.80}$$

kde L značí ztrátovou funkci. Spočtěme derivace vzhledem ke složkám matice D

$$\frac{\partial L}{\partial d_{ii}} = \frac{\partial L}{\partial t(x)} \frac{\partial t(x)}{\partial d_{ii}} = \Delta^{(t(x))} \frac{\operatorname{sgn}(d_{ii})}{|d_{ii}|} = \frac{\Delta^{(t(x))}}{d_{ii}}, \quad \forall i \in \{1, \dots, n\}. \tag{1.81}$$

Nyní zpětně propagujme gradient, kde pro přehlednost označíme $y = \log\left(p_{ch}(UDVx + b)\right)$,

$$\Delta^{(y)} = \frac{\partial L}{\partial y} = \frac{\partial L}{\partial t(x)} \frac{\partial t(x)}{\partial y} = \Delta^{(t(x))} \cdot 1 = \Delta^{(t(x))}. \tag{1.82}$$

Příchozí gradient během druhé části zpětné propagace má tvar

$$\Delta^{(UDVx+b)} = \frac{\partial L}{\partial\left(UDVx + b\right)} \in \mathbb{R}^n. \tag{1.83}$$

Nejprve spočteme derivace vzhledem ke složkám vektoru b

$$\frac{\partial L}{\partial b_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial\left(UDVx + b\right)_j} \frac{\partial\left(UDVx + b\right)_j}{\partial b_i} = \sum_{j=1}^{n} \Delta_j^{(UDVx+b)} \delta_{ij} = \Delta_i^{(UDVx+b)}. \tag{1.84}$$

Zpětně propagujme gradient

$$\Delta^{(UDVx)} = \frac{\partial L}{\partial\left(UDVx\right)} = \sum_{j=1}^{n} \frac{\partial L}{\partial\left(UDVx + b\right)_j} \frac{\partial\left(UDVx + b\right)_j}{\partial\left(UDVx\right)} = \Delta^{(UDVx+b)} \cdot 1 = \Delta^{(UDVx+b)}. \tag{1.85}$$

Nyní potřebujeme vědět, jak spočíst derivace vzhledem k matici v Butterfly rozkladu a jak skrz ni zpětně propagovat gradient. Na matici v Butterfly rozkladu se dá nahlížet jako na postupné násobení množstvím Givensových rotací. Odvodíme tedy postup pro jedinou rotaci násobenou libovolným vektorem, a tento postup následně budeme opakovat pro všechny Givensovy rotace v Butterfly rozkladu. Příchozí gradient má tvar

$$\Delta^{(Gz)} = \frac{\partial L}{\partial\left(Gz\right)} \in \mathbb{R}^n, \tag{1.86}$$

kde L je ztrátová funkce, G je Givensova rotace (1.35) a $z \in \mathbb{R}^n$. Použitím řetězového pravidla získáme

$$\frac{\partial L}{\partial \theta} = \sum_{k=1}^{n} \frac{\partial L}{\partial (Gz)_k} \frac{\partial (Gz)_k}{\partial \theta} = \sum_{k=1}^{n} \Delta_k^{(Gz)} \frac{\partial (Gz)_k}{\partial \theta}. \tag{1.87}$$

Zjednodušíme druhý člen sumy

$$\frac{\partial (Gz)_k}{\partial \theta} = \frac{\partial \sum_{l=1}^{n} G_{kl} z_l}{\partial \theta} = \sum_{l=1}^{n} \frac{\partial G_{kl}}{\partial \theta} z_l. \tag{1.88}$$

Nyní spočteme derivaci Givensovy rotace vzhledem k úhlu θ

$$\frac{\partial G(i,j,\theta)}{\partial \theta} = \begin{pmatrix} 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & -c & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ \\ \end{matrix} \tag{1.89}$$
$$\qquad\qquad\qquad\quad i \qquad\quad j$$

a dosadíme do (1.88)

$$\sum_{l=1}^{n} \frac{\partial G_{kl}}{\partial \theta} z_l = \delta_{ki} \left(-z_i \sin(\theta) - z_j \cos(\theta) \right) + \delta_{kj} \left(z_i \cos(\theta) - z_j \sin(\theta) \right). \tag{1.90}$$

Dosazením (1.90) do (1.87) získáme

$$\frac{\partial L}{\partial \theta} = \sum_{k=1}^{n} \frac{\partial L}{\partial (Gz)_k} \frac{\partial (Gz)_k}{\partial \theta} = \Delta_i^{(Gz)} \left(-z_i \sin(\theta) - z_j \cos(\theta) \right) + \Delta_j^{(Gz)} \left(z_i \cos(\theta) - z_j \sin(\theta) \right). \tag{1.91}$$

Pokračujme zpětnou propagací gradientu za G. S pomocí řetězového pravidla získáme

$$\begin{aligned} \Delta_i^{(z)} &= \frac{\partial L}{\partial z_i} \\ &= \sum_{k=1}^{n} \frac{\partial L}{\partial (Gz)_k} \frac{\partial (Gz)_k}{\partial z_i} \\ &= \sum_{k=1}^{n} \Delta_k^{(Gz)} \frac{\partial \sum_{l=1}^{n} G_{kl} z_l}{\partial z_i} \\ &= \sum_{k=1}^{n} \Delta_k^{(Gz)} G_{ki} \\ &= \left(G^T \Delta^{(Gz)} \right)_i, \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{1.92}$$

Opakováním (1.91) a (1.92) pro každou rotaci v Butterfly rozkladu matice U jsme schopni spočíst derivace vzhledem ke všem jejím parametrům a zpětně propagovat gradient $\Delta^{(UDVx)}$ na $\Delta^{(DVx)}$.

Nyní potřebujeme spočíst derivace vůči matici D a zpětně propagovat za ni gradient. Příchozí gradient má tvar

$$\Delta^{(DVx)} = \frac{\partial L}{\partial (DVx)} \in \mathbb{R}^n. \tag{1.93}$$

Nejprve spočteme derivace vzhledem k prvkům matice D

$$\frac{\partial L}{\partial D_{ii}} = \sum_{j=1}^{n} \frac{\partial L}{\partial (DVx)_j} \frac{\partial (DVx)_j}{\partial D_{ii}} = \sum_{j=1}^{n} \Delta_j^{(DVx)} \frac{\partial D_{jj}}{\partial D_{ii}} (Vx)_j = \sum_{j=1}^{n} \Delta_j^{(DVx)} \delta_{ij} (Vx)_j = \Delta_i^{(DVx)} (Vx)_i. \tag{1.94}$$

Zpětně propagujme gradient

$$\Delta^{(Vx)} = \frac{\partial L}{\partial (Vx)} = \sum_{j=1}^{n} \frac{\partial L}{\partial (DVx)_j} \frac{\partial (DVx)_j}{\partial (Vx)} = \sum_{j=1}^{n} \Delta_j^{(Vx)} D_{jj} = D\Delta^{(Vx)}. \tag{1.95}$$

Jelikož V je také matice v Butterfly rozkladu, můžeme pro ni použít stejný postup jako pro matici U. Víme tedy, jak spočíst derivace a zpětně propagovat gradient pro transformaci $UDVx + b$.

Kapitola 2

Implementace

Tato kapitola obsahuje popis implementace algoritmů z kapitoly 1. Z důvodu velké výpočetní náročnosti byly algoritmy implementovány na architekturu GPU, která poskytuje vyšší výpočetní výkon za cenu obtížnější implementace. Pro implementaci byl zvolen jazyk Julia.

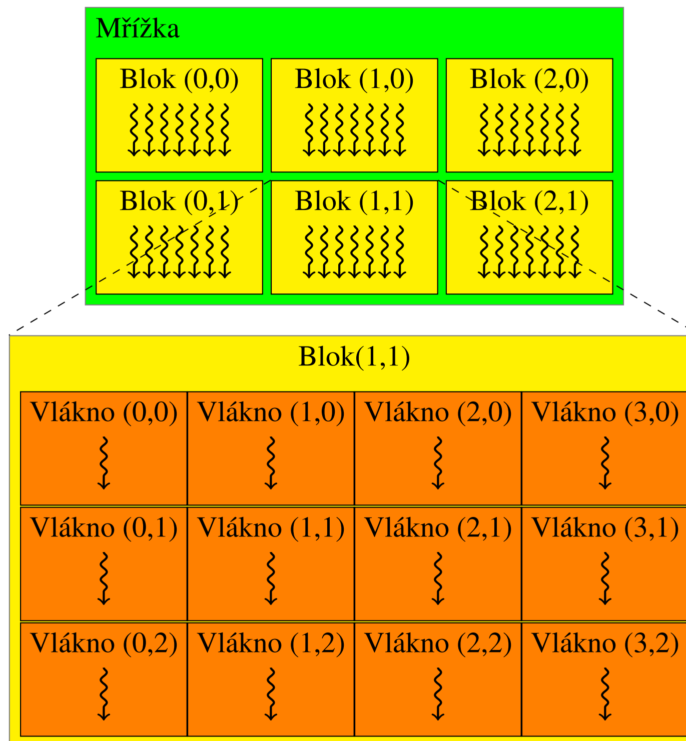
2.1 GPU, CUDA a Julia

Grafické výpočetní jednotky, zkráceně GPU z anglického *graphics processing unit*, začaly jako specializované grafické procesory, které byly schopny velmi rychle generovat snímky pro zobrazení na displayi. GPU se dále vyvinuly v procesory pro obecné výpočty vhodné pro vysoce paralelní výpočetní úkoly. Za tímto účelem společnost NVIDIA v roce 2006 představila *Compute Unified Device Architecture*, zkráceně CUDA. CUDA je programovací rozhraní umožňující kódu napsanému například v C nebo C++ efektivně fungovat na GPU.

Pro využití všeho dostupného výpočetního výkonu GPU musí být člověk obeznámen s hardwarovou architekturou GPU, jelikož na tom závisí, které části programu spolu mohou snadno komunikovat, a které části paměti jsou rychle přístupné. GPU obsahuje globální paměť, která na dnešních GPU může dosahovat velikosti až 80 GiB. Tato paměť je přístupná všem výpočetním jednotkám na GPU, ale má vysokou latenci. Výpočetní jednotky na GPU jsou seskupeny do *streaming multiprocesorů*, zkráceně SM. Jedna GPU může mít několik desítek SM. V dnešních GPU se jeden SM skládá z 64 výpočetních jednotek. Každý SM obsahuje sdílenou paměť, pomocí které mezi sebou mohou komunikovat jednotlivé výpočetní jednotky v daném SM. Tato paměť má velmi malou latenci. Výpočetní jednotky mají přístup do sdílené paměti pouze svého SM. Velikost této paměti na dnešních GPU dosahuje až 164 KiB.

CUDA umožňuje definovat speciální funkce, zvané *kernely*. Tyto kernely mohou být za běhu programu zpracovávány mnoha paralelními vlákny na GPU. Každé vlákno (anglicky *thread*) je zpracováváno na jedné výpočetní jednotce na GPU. Vlákna jsou při běhu seskupeny do bloků (*thread blocks*), což jsou trojrozměrná pole jednotlivých vláken. Každý blok vláken je zpracováván pouze jedním SM. Každé vlákno v bloku má své jedinečné ID, což je trojrozměrný vektor. Pokud je vyžadováno pouze jedno nebo dvojrozměrné indexování, je možné nadbytečné dimenze vynechat. Maximální počet vláken v bloku je na dnešních GPU 1024. Pokud je potřeba více než 1024 vláken, je možné spustit více bloků najednou. Bloky jsou poté organizovány do trojrozměrné mřížky (*thread grid*), kde každý blok má své jedinečné ID, což je opět trojrozměrný vektor. Mřížka je zpracovávána celou GPU.

V jednom bloku jsou vlákna seskupeny do *warpů*. Warp je skupina 32 vláken, která vykonává vždy pouze jeden typ instrukce zároveň. Plného výkonu je dosaženo pouze pokud se všechny vlákna ve warpu snaží vykonávat tu samou instrukci. Pokud se cesty vláken ve warpu rozejdou z důvodu rozdílných vyhodnocení podmínky, pak práci vykonávají pouze vlákna ve stejné větvi, zbylá jsou deaktivovaná.



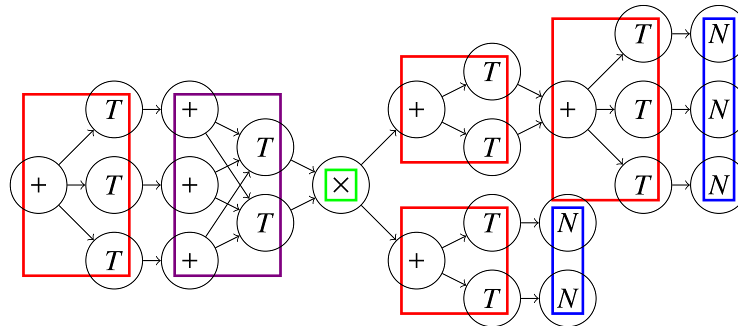
Obrázek 2.1: Mřížka bloků vláken [6].

Při přístupu do globální paměti mohou vlákna slučovat své přístupy, pokud jsou splněny specifické podmínky. Jednou z těchto podmínek je, že žádaná data jsou v globální paměti uložena sekvenčně.

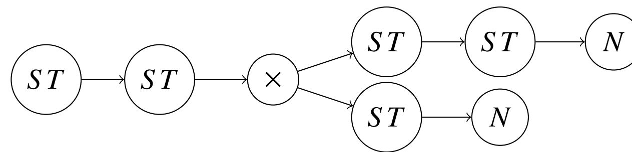
Při spuštění kernelu je velikost mřížky fixní, a všechny bloky v ním musí mít stejnou velikost. Tyto hodnoty jsou jednotlivým vláknům k dispozici jako trojrozměrné vektory dimenze mřížky a dimenze bloku [6, 10].

Julia je vysokoúrovňový dynamický programovací jazyk, který je dobře využitelný pro numerické výpočty. Důležitými vlastnostmi Julie jsou volitelné typování a multiple dispatch. Díky just-in-time kompilace je možné generovat funkce specializované pro různé typy argumentů. Díky tomuto dosahuje Julia podobného výkonu jako staticky kompilované jazyky, jako například C. Julia je open-source, distribuovaná pod MIT licencí [11].

Pro implementaci v Julii využijeme několik knihoven. První knihovnou je `CUDA.jl` [3, 4], která umožňuje nativní zpracování kódu v jazyce Julia na GPU. Dále využijeme `Flux.jl` [8, 9], což je knihovna zprostředkovávající automatické derivování. Tato knihovna umožňuje definovat parametry velmi složitých modelů, a tyto parametry následně optimalizovat. Také zvládá derivovat jednoduché funkce, a tudíž není třeba definovat každé triviální pravidlo pro derivaci a zpětnou propagaci ručně. S `Flux.jl` souvisí knihovna `ChainRules.jl` [5]. Tato knihovna umožňuje definovat pravidla pro výpočty derivací a zpětnou propagaci u námi definovaných funkcí, které jsou příliš složité pro automatickou derivaci. Poslední použitou knihovnou je `Distributions.jl` [2, 14], což je knihovna poskytující implementace pravděpodobnostních distribucí.



Obrázek 2.2: SPTN splňující restrikce s vyznačenými vrstvami.



Obrázek 2.3: Struktura SPTN z obrázku 2.2 po sloučení uzlů do SPTN vrstev.

2.2 Datová struktura SPTN

V této sekci bude popsán způsob, jakým je implementována datová struktura SPTN. Jak bylo popsáno v subsekci 1.2.3, budeme se zabývat pouze sítěmi, které se skládají z vrstev součtových uzlů následovaných transformačními uzly. Tyto vrstvy mohou mít jako potomka buď další součtově-transformační vrstvu, součinový uzel, nebo vrstvu listů. Příklad se znázorněnými SPTN vrstvami je na obrázku 2.2, kde červenou jsou vyznačeny součtově-transformační vrstvy sdílející parametry součtových uzlů, fialovou jsou vyznačeny součtově-transformační vrstvy nesdílející parametry součtových uzlů, modrou vrstvy listů a zelenou součinové vrstvy. Struktura sítě z obrázku 2.2 po sloučení uzlů do SPTN vrstev je znázorněna na obrázku 2.3. Důvodem slučování uzlů do SPTN vrstev je potřeba seskupování spouštění kernelů na GPU. Pokud bychom kernely spouštěli pro jednotlivé uzly, nedostatek paralelismu by vedl na nízké využití výpočetního výkonu GPU.

Každá SPTN vrstva obsahuje parametry všech uzlů, které jsou v ní obsaženy, a informaci, kterou SPTN vrstvou, případně kterými SPTN vrstvami, je následována. Každá SPTN vrstva dále obsahuje informaci o své dimenzi, a jestli jsou její parametry uloženy v jednoduché nebo dvojité přesnosti.

Součtově-transformační vrstvy obsahují parametry pro všechny součtové uzly uložené ve dvourozměrném poli. První index vyjadřuje, kterému transformačnímu uzlu váha přísluší, druhý index vyjadřuje, kterému součtovému uzlu váha přísluší.

Dále je třeba uložit parametry pro transformační uzly. Každý transformační uzel se skládá ze dvou matic v Butterfly rozkladu, jedné diagonální matice, a vektoru biasu. Výjimkou jsou transformační uzly přímo následované listovými uzly, které z důvodů popsaných v sekci 1.2.4 neobsahují matici U.

Jelikož každý transformační uzel ve stejné součtově-transformační vrstvě implementuje transformaci působící na vektory stejné dimenze, mají všechny matice stejnou velikost, a stejně tak vektory biasu. Toto umožňuje bias vektory všech transformací uložit do dvojrozměrného pole. Podobně můžeme do dvojrozměrného pole uložit i diagonální matice, jelikož stačí ukládat pouze prvky na diagonále. První index vyjadřuje, ke které dimenzi parametr přísluší, druhý index vyjadřuje, ke které transformaci parametr přísluší.

Nyní potřebujeme uložit parametry pro matice v Butterfly rozkladu. Jelikož každý transformační uzel v součtově-transformační vrstvě reprezentuje stejný typ transformace, pouze s jinými parametry, můžeme parametry opět uložit do vícerozměrných polí, kde poslední rozměr bude určovat, ke které transformaci parametry náleží. Z implementačních důvodů blíže popsaných v sekci 2.3.2 budou parametry pro matici v Butterfly rozkladu ukládány střídavě do trojrozměrných a dvojrozměrných polí. Trojrozměrné pole ukládá parametry pro několik po sobě jdoucích Butterfly vrstev. První index vyjadřuje, ke které dimenzi parametr přísluší, druhý index vyjadřuje, ke které Butterfly vrstvě parametr přísluší, a třetí index vyjadřuje, ke které transformaci parametr přísluší. Dvojrozměrné pole ukládá parametry pouze pro jednu Butterfly vrstvu. První index vyjadřuje, ke které dimenzi parametr přísluší, a druhý index vyjadřuje, ke které transformaci parametr přísluší.

2.3 Učení

Tato sekce popisuje průběh výpočtu logaritmu věrohodnosti pro nějaká pozorování, a následnou optimalizaci ztrátové funkce. Za ztrátovou funkci byla volena záporně vzatá hodnota logaritmu věrohodnosti pozorování, popřípadě při větším počtu pozorování v minibatchi záporně vzatý průměr logaritmů pozorování. Díky použití knihovny `Flux.jl` pro učení je však velmi snadné ztrátovou funkci změnit.

2.3.1 Značení v pseudo-algoritmech

V pseudo-algoritmech této kapitoly budeme používat jednotné značení pro různé typy proměnných. Toto značení je popsáno v tabulce 2.1.

2.3.2 Inference

Výpočet probíhá rekurzivně. Rekurze začíná voláním funkce `logpdf`, jejíž argumenty jsou kořen sítě s uzly sloučenými do SPTN vrstev, a vektory pozorování uložené ve dvourozměrném poli. Tato funkce slouží pro připravení dat pro rekurzi. Jedním z kroků je překopírování pozorování z RAM do globální paměti GPU. Julia ukládá data v dvourozměrných polích po sloupcích. Pro využití slučování přístupů do paměti na GPU je potřeba, aby sekvenčně byly uloženy nikoliv prvky jednoho vektoru, ale i-té prvky všech vektorů pozorování. Matici vektorů pozorování tedy potřebujeme mít uloženou tak, že první index určuje vektor pozorování a druhý index určuje, o kterou složku vektoru se jedná. Toto lze matematicky interpretovat jako transpozici celého výpočtu. Druhým z kroků je tedy transpozice matice pozorování. Jelikož má výpočet vysokou paměťovou náročnost, mohlo by při snaze spočíst logaritmus věrohodnosti většího množství pozorování najednou dojít k vyčerpání paměti GPU. Proto poslední přípravný krok, který funkce `logpdf` dělá, je rozdělení matice pozorování na menší části, a následné sloučení výsledků. Nyní rozeberme funkci `_logpdf`, která implementuje samotný výpočet logaritmu věrohodnosti.

Funkce `_logpdf` má za argumenty SPTN vrstvu, a matici obsahující vektory pozorování. Tato funkce má tři metody pro různé typy SPTN vrstev, součtově-transformační, součinovou a listovou. Nejprve popišme metodu pro součtově-transformační vrstvu.

2.3.2.1 `_logpdf` pro součtově-transformační vrstvu

Tato metoda implementuje složení (1.68) a (1.79)

$$st(x) = \operatorname*{lse}_{i=1}^{m} \left(\log\left(\operatorname{softmax}(\alpha_i)\right) + \log\left(p_i(U_i D_i V_i x + b_i)\right) + \operatorname{lad}\left(D_i\right)\right), \tag{2.1}$$

označení	popis
L	SPTN vrstva
ST	součtově-transformační vrstva
PL	součinová vrstva
LL	listová vrstva
B	Butterfly blok
bf	matice v Butterfly rozkladu
bfl	Butterfly vrstva
$bfml$	Butterfly multivrstva
g	matice/tenzor obsahující Givensovy rotace
θ	matice/tenzor obsahující úhly Givensových rotací
D	matice obsahující parametry diagonálních matic
b	matice obsahující parametry vektorů biasů
X	matice/tenzor obsahující vektory pozorování
l	vektor/matice obsahující logaritmy věrohodností
Δ	matice/tenzor obsahující gradient při zpětné propagaci
∇b	matice derivací ztrátové funkce vzhledem k prvkům b
$\nabla \theta$	matice/tenzor derivací ztrátové funkce vzhledem k prvkům θ
∇D	matice derivací ztrátové funkce vzhledem k prvkům D
w	struktura popisující pravděpodobnosti jednotlivých komponent SPTN
t	struktura popisující strom reprezentující komponentu SPTN
sid	index součtového uzlu v ST
tid	index transformačního uzlu v ST
p	pravděpodobnost komponenty SPTN

Tabulka 2.1: Stručný popis značení použitého ve zbytku kapitoly.

kde jsme označili *lad* jako logaritmus absolutní hodnoty determinantu

$$\mathrm{lad}\,(D) = \log|\det(D)| = \sum_{i=1}^{n} \log(|d_{ii}|)\,. \tag{2.2}$$

Zápis (2.1) lze dále zjednodušit rozepsáním logaritmu funkce softmax

$$\log(\mathrm{softmax}(\alpha_i)) = \log \frac{\exp(\alpha_i)}{\sum_{j=1}^{n} \exp\left(\alpha_j\right)} = \alpha_i - \operatorname*{lse}_{j=1}^{n}\left(\alpha_j\right). \tag{2.3}$$

Dosazením (2.3) do (2.1) získáme

$$st(x) = \operatorname*{lse}_{i=1}^{m}\left(\alpha_i - \operatorname*{lse}_{j=1}^{n}\left(\alpha_j\right) + \log(p_i(U_i D_i V_i x + b_i)) + \mathrm{lad}\,(D_i)\right). \tag{2.4}$$

Metoda nejdříve připraví vektory pozorování pro transformaci, následně transformuje, převede zpět do matice aby mohly sloužit jako argument další vnořené funkci `_logpdf`, rekurzivně spočte jejich logaritmy věrohodnosti, a následně provede vážený součet součtovými uzly. Pseudo-algoritmus je popsán algoritmem 1. Příklad přechodů mezi vektory, maticemi a tenzory v algoritmu 1 je znázorněn na ob-

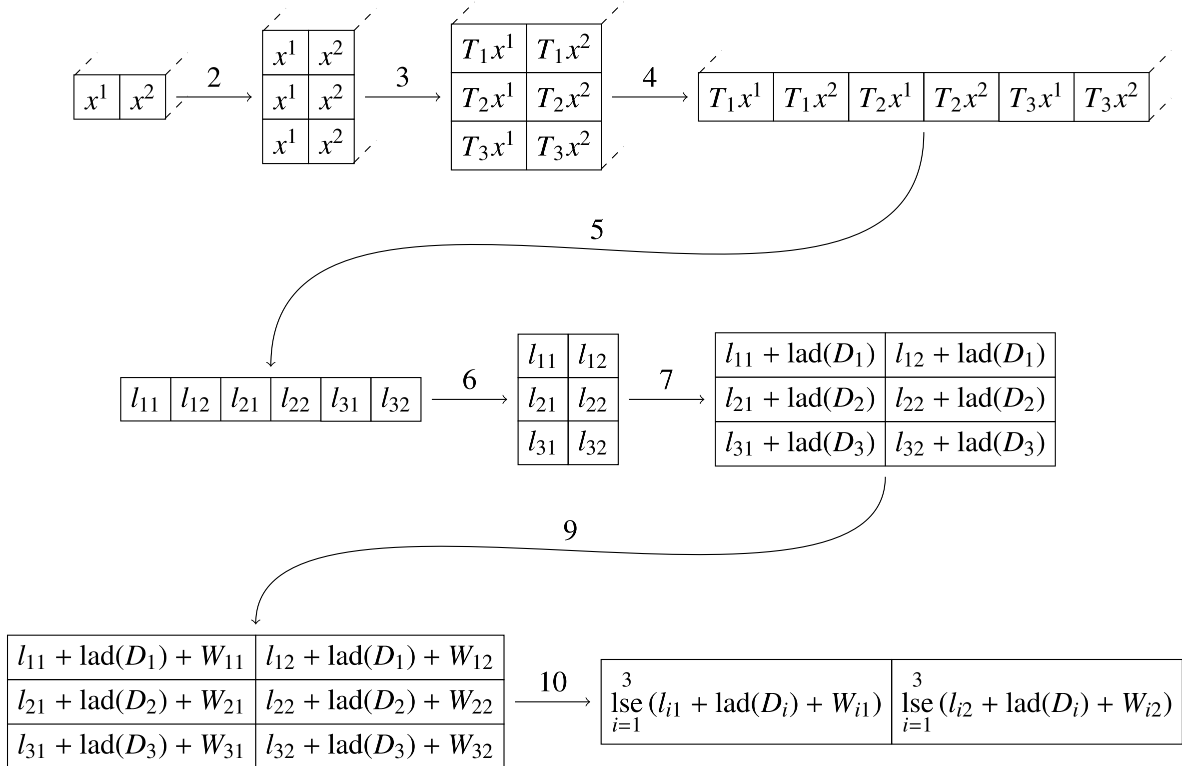
Algoritmus 1: Metoda `_logpdf` pro součtově-transformační vrstvu.

1 `_logpdf`(*ST, X*)
2 zduplikuj matici X do tenzoru, který obsahuje jednu matici X pro každý transformační uzel
3 transformuj tenzor X transformačními uzly
4 přerovnej tenzor X do matice
5 $l \leftarrow$ `_logpdf`(*potomek ST, X*)
6 přerovnej vektor l do matice
7 přičti logaritmy absolutních hodnot determinantů k l
8 $W \leftarrow \alpha - \mathrm{lse}\,(\alpha)$
9 $l \leftarrow l + W$
10 $l \leftarrow \mathrm{lse}\,(l)$
11 vrať l

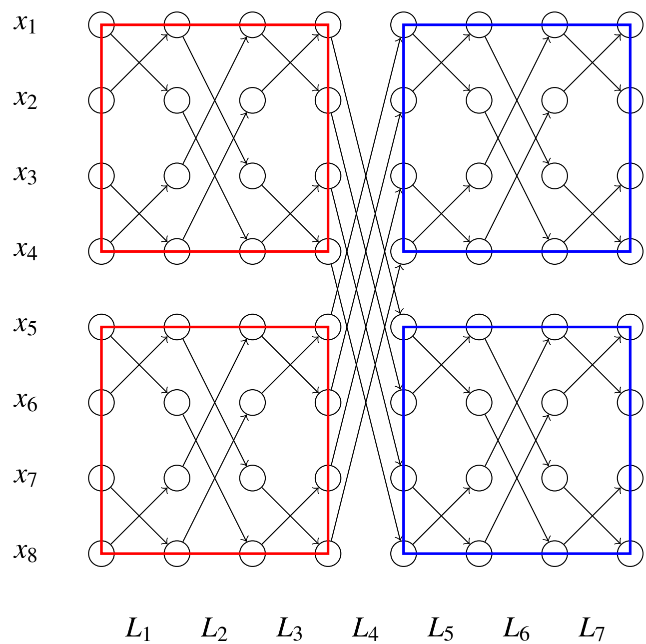
rázku 2.4 pro dva vektory x^1 a x^2.

Popišme nyní detailně jak funguje transformace vektorů na řádku 4 algoritmu 1. Na vstupu máme trojrozměrný tenzor, kde první dimenze odpovídá vektoru pozorování, druhá dimenze indexu složky vektoru, a přes třetí dimenzi jsou si prvky rovny, jelikož každá matice bude teprve transformována svým příslušným transformačním uzlem.

V první části transformace jsou vektory násobeny ortogonálními maticemi V_i uloženými v Butterfly rozkladu. V sekci 2.2 byl popsán způsob uložení těchto parametrů střídavě do trojrozměrných polí pro několik po sobě jdoucích Butterfly vrstev a dvojrozměrných polí pro pouze jednu Butterfly vrstvu. Slučování Butterfly vrstev do jedné Butterfly multivrstvy umožní nejdříve načíst data z globální paměti s vysokou latencí do sdílené paměti jednotlivých streaming multiprocesorů, následně provést výpočet pro několik Butterfly vrstev (nejlepší výsledky byly dosaženy sloučením 31 Butterfly vrstev), a poté výsledky zapsat zpět do globální paměti. Tímto přístupem je výpočetní kapacita GPU využita lépe, než pokud by se pro každou Butterfly vrstvu data načítala z a následně ukládala do globální paměti. Sloučení tří Butterfly vrstev je znázorněno na obrázku 2.5, kde každá Butterfly multivrstva (znázorněna stejnou barvou) je složena ze dvou Butterfly bloků. Při aplikaci prvních tří vrstev Butterfly rozkladu na vektor délky



Obrázek 2.4: Ilustrace metody `_logpdf` pro součtově-transformační vrstvu s vyznačenými řádky algoritmu 1.



Obrázek 2.5: Sloučení Butterfly vrstev za účelem využití sdílené paměti.

osm probíhá výpočet pro každý Butterfly blok nezávisle. Výpočet tedy může běžet na dvou různých SM, které spolu nemusí (a kvůli hardwarové architektuře GPU ani nemůžou) komunikovat, a během výpočtu není třeba komunikovat s globální pamětí. Pokud Butterfly rozklad obsahuje více Butterfly vrstev než slučujeme do Butterfly multivrstev, tak mezi Butterfly multivrstvami zbudou jednotlivé Butterfly vrstvy.

Základní smyčka násobení maticemi v Butterfly rozkladu je popsána algoritmem 2. Řádek 3 algo-

Algoritmus 2: Základní smyčka násobení tenzoru X ortogonálními maticemi v Butterfly rozkladu.

```
1 bfmulax(bf, X)
2 for každou Butterfly vrstvu/multivrstvu bfx v bf do
3 |   aplikuj bfx na X
4 end
```

ritmu 2 za běhu programu volá pomocí multiple dispatch buď funkci pro násobení jednoduchou Butterfly vrstvou, nebo pro násobení Butterfly multivrstvou.

Funkci pro násobení jednoduchou Butterfly vrstvou popisuje algoritmus 3. V algoritmu 3 jsou ba-

Algoritmus 3: Násobení tenzoru X jednoduchou Butterfly vrstvou.

1 `bfmulaxl(`bfl, X`)`
2 **for** *každou matici* $X_{\cdot,\cdot,k}$ *v* X **do**
3 | **for** *každou Givensovu rotaci* $g_{m,k}$ *s parametry* $\theta_{m,k}, i_{m,k}, j_{m,k}$ *v* bfl **do**
4 | | **for** *každý vektor* $X_{l,\cdot,k}$ *v* $X_{\cdot,\cdot,k}$ **do**
5 | | | $tmp_{l,i_{m,k},k} \leftarrow \cos(\theta_{m,k}) \cdot X_{l,i_{m,k},k} - \sin(\theta_{m,k}) \cdot X_{l,j_{m,k},k}$
6 | | | $tmp_{l,j_{m,k},k} \leftarrow \sin(\theta_{m,k}) \cdot X_{l,i_{m,k},k} + \cos(\theta_{m,k}) \cdot X_{l,j_{m,k},k}$
7 | | | $X_{l,i_{m,k},k} \leftarrow tmp_{l,i_{m,k},k}$
8 | | | $X_{l,j_{m,k},k} \leftarrow tmp_{l,j_{m,k},k}$
9 | | **end**
10 | **end**
11 **end**

revně zvýrazněné smyčky. Tyto smyčky jsou v implementaci na GPU vypočteny paralelně. Kernel pro funkci `bfmulaxl` je spouštěn v konfiguraci, kde CUDA bloky jsou dvourozměrná pole vláken, a CUDA mřížka je dvourozměrné pole bloků. První blokový index určuje index l vektoru násobeného vláknem. Druhý blokový index určuje číslo rotace g. Velikost CUDA bloku je $(32,8)$, celkem tedy 256 vláken rozdělených do osmi warpů. Jelikož první blokový index určuje, který vektor je násoben, tak vlákna v jednom warpu potřebují načíst prvek na tom samém místě svých vektorů. Tyto prvky jsou v paměti uloženy sekvenčně, jak bylo popsáno dříve, dochází tedy ke sdružování přístupů do paměti. Je-li vektorů více než 32, tak žlutá smyčka není plně paralelizována, provádí se po blocích velikosti 32. První mřížkový index slouží ke spuštění dostatečného množství bloků tak, aby byly pokryty všechny rotace. Druhý mřížkový index určuje index k matice $X_{\cdot,\cdot,k}$.

Funkci pro násobení Butterfly multivrstvou popisuje algoritmus 4. V algoritmu 4 jsou opět barevně zvýrazněné smyčky, které jsou v implementaci na GPU vypočteny paralelně. Kernel pro funkci `bfmulaxml` je spouštěn v konfiguraci, kde CUDA bloky jsou dvourozměrná pole vláken, a CUDA mřížka je trojrozměrné pole bloků. Jeden CUDA blok provádí výpočet jednoho Butterfly bloku pro 32 vektorů. První blokový index určuje index l vektoru násobeného vláknem. Druhý blokový index určuje číslo rotace g. Velikost CUDA bloku je $(32,2)$, celkem tedy 64 vláken rozdělených do dvou warpů. Pokud je

Algoritmus 4: Násobení tenzoru X Butterfly multivrstvou.

```
1  bfmulaxml(bfml, X)
2  for každý Butterfly blok B v bfml do
3  |  for každou matici X_{·,·,k} v X do
4  |  |  načti část X_{·,·,k} příslušnou B do sdílené paměti
5  |  |  for pro každou Butterfly vrstvu rotací g_{·,l,k} v bfml do
6  |  |  |  for každou rotaci g_{m,l,k} s parametry θ_{m,l,k}, i_{m,l,k}, j_{m,l,k} v g_{·,l,k} do
7  |  |  |  |  for každý vektor X_{n,·,k} v X_{·,·,k} do
8  |  |  |  |  |  tmp_{l,i_{m,l,k},k} ← cos(θ_{m,l,k}) · X_{l,i_{m,l,k},k} − sin(θ_{m,l,k}) · X_{l,j_{m,l,k},k}
9  |  |  |  |  |  tmp_{l,j_{m,l,k},k} ← sin(θ_{m,l,k}) · X_{l,i_{m,l,k},k} + cos(θ_{m,l,k}) · X_{l,j_{m,l,k},k}
10 |  |  |  |  |  X_{l,i_{m,l,k},k} ← tmp_{l,i_{m,l,k},k}
11 |  |  |  |  |  X_{l,j_{m,l,k},k} ← tmp_{l,j_{m,l,k},k}
12 |  |  |  |  end
13 |  |  |  end
14 |  |  end
15 |  |  ulož část X_{·,·,k} příslušnou B zpět do globální paměti
16 |  end
17 end
```

v jedné Butterfly vrstvě Butterfly bloku více rotací, než je velikost druhé dimenze CUDA bloku, tak červená smyčka není plně paralelizována, provádí se po blocích velikosti 2 v rámci jednoho Butterfly bloku. Ze stejných důvodů jako u kernelu pro funkci `bfmulaxl` zde dochází ke slučování přístupů do paměti. První mřížkový index slouží ke spuštění dostatečného množství bloků tak, aby byly pokryty všechny vektory. Druhý mřížkový index slouží ke spuštění CUDA bloku pro každý Butterfly blok. Třetí mřížkový index určuje index k matice $X_{\cdot,\cdot,k}$

Dále potřebujeme umět vynásobit tenzor X diagonálními maticemi D. Pseudo-algoritmus tohoto násobení je popsán algoritmem 5. V algoritmu 5 jsou opět barevně zvýrazněné smyčky, které jsou v imple-

Algoritmus 5: Násobení tenzoru X diagonálními maticemi D.

```
1  dmulax(D, X)
2  for každou matici X_{·,·,k} v X do
3  |  for každý vektor X_{i,·,k} v X_{·,·,k} do
4  |  |  for každý prvek X_{i,j,k} vektoru X_{i,·,k} do
5  |  |  |  X_{i,j,k} ← D_{j,k} X_{i,j,k}
6  |  |  end
7  |  end
8  end
```

mentaci na GPU vypočteny paralelně. Pro tento výpočet není třeba psát specializovaný kernel, jelikož se dá snadno zapsat pomocí funkce `broadcast`, která provádí binární operaci prvek po prvku pro dvě pole argumentů, které mají stejný počet dimenzí. Aby bylo možno funkci `broadcast` použít, délky korespondujících dimenzí musí být buď stejné, nebo alespoň pro jedno pole musí mít daná dimenze velikost jedna. Poté se binární operace provede prvek po prvku mezi všemi řezy pole s nejedničkovou dimenzí a polem s dimenzí rovnou jedné. Například při násobení matice o velikosti $(2,2)$ a vektoru délky 2 (in-

terpretovaného jako dvourozměrné pole o rozměrech $(2, 1)$) jsou vynásobeny oba dva sloupce matice prvek po prvku vektorem. Matici obsahující hodnoty diagonálních prvků matic D lze interpretovat jako trojrozměrné pole $\tilde{D}$, které má velikost první dimenze rovnu jedné. Algoritmus 5 lze velmi snadno zapsat jako `X.*D̃`, kde tečka je pouze zjednodušení zápisu funkce `broadcast`. Výpočet se poté díky multiple dispatch a definicím z knihovny `CUDA.jl` specializuje na efektivní metodu pro GPU.

Násobení ortogonálními maticemi U_i probíhá stejně jako násobení maticemi V_i. Nyní potřebujeme přičíst vektory b k tenzoru X. Tento výpočet probíhá velmi podobně jako násobení maticemi D a popisuje ho algoritmus 6. Pro tento výpočet opět není třeba psát specializovaný kernel, jelikož se dá snadno zapsat

Algoritmus 6: Přičtení vektorů b k tenzoru X.

```
1 badd(b, X)
2 for každou matici X_{·,·,k} v X do
3 |   for každý vektor X_{i,·,k} v X_{·,·,k} do
4 |   |   for každý prvek X_{i,j,k} vektoru X_{i,·,k} do
5 |   |   |   X_{i,j,k} ← b_{j,k} + X_{i,j,k}
6 |   |   end
7 |   end
8 end
```

pomocí funkce `broadcast`. Matici obsahující vektory b lze interpretovat jako trojrozměrné pole $\tilde{b}$, které má velikost první dimenze rovnu jedné. Algoritmus 6 lze velmi snadno zapsat jako `X.+=b̃`. Výpočet se poté opět díky multiple dispatch a definicím z knihovny `CUDA.jl` specializuje na efektivní metodu pro GPU.

Nyní víme, jak probíhá transformace tenzoru X. Dalším krokem ve výpočtu je spočtení logaritmů věrohodnosti pro jednotlivé vektory v X, o který se postarají metody funkce `_logpdf` pro ostatní typy SPTN vrstev. K těmto logaritmům věrohodnosti jsou poté přičteny logaritmy absolutních hodnot determinantů matic D, a následně jsou logaritmy věrohodnosti váženě sečteny pomocí součtových uzlů. Implementace těchto výpočtů je velmi jednoduchá, jelikož všechny paralelizovatelné operace jsou zapsatelné pomocí funkcí `broadcast` a `mapreduce`. Funkce `mapreduce` nejprve na každý prvek pole aplikuje nějakou funkci, a následně provede redukci v zadaných dimenzích pomocí dané binární asociativní operace. Takto jsme schopni spočíst logaritmy absolutních hodnot determinantů voláním pouze jedné funkce na GPU, a to `mapreduce`, kde mapovaná funkce je $\log \circ \text{abs}$ a redukční operace je $+$, a následně je přičíst pomocí `broadcast`.

Abychom byli schopni provést sečtení součtovými uzly, je třeba implementovat funkci `logsumexp`, což lze udělat způsobem popsaným v algoritmu 7, kde x je pole hodnot, a $dims$ je uspořádaná n-tice určující, přes které dimenze má proběhnout redukce. Přičtení a odečtení xm je nutné z důvodu numerické

Algoritmus 7: Implementace funkce `lse` pomocí `broadcast` a `mapreduce`.

```
1 logsumexp(x, dims)
2 xm ← mapreduce(x, Identita, max, dims)
3 s ← mapreduce(x.-xm, exp, +, dims)
4 vrať log(s).+xm
```

stability. Při aplikaci funkce exp na prvky x by mohlo snadno dojít ke ztrátě přesnosti, pokud by prvky x byly velké. Toto je ošetřeno odečtením největšího prvku x od všech ostatních prvků, a následným přičtením zpět na konci výpočtu, což způsobí, že největší z argumentů exponenciely bude roven nule,

což pro numerickou stabilitu není problematické. Ověříme, že tento postup je matematicky ekvivalentní. Jednoduchými úpravami získáme

$$\begin{aligned}
\operatorname*{lse}_{i=1}^{n}(x_i - \max(x)) &= \log\left(\sum_{i=1}^{n}\exp(x_i - \max(x))\right) \\
&= \log\left(\sum_{i=1}^{n}\exp(x_i)\exp(-\max(x))\right) \\
&= \log\left(\exp(-\max(x))\sum_{i=1}^{n}\exp(x_i)\right) \\
&= \log\left(\sum_{i=1}^{n}\exp(x_i)\right) + \log(\exp(-\max(x))) \\
&= \log\left(\sum_{i=1}^{n}\exp(x_i)\right) - \max(x) \\
&= \operatorname*{lse}_{i=1}^{n}(x_i) - \max(x), \qquad (2.5)
\end{aligned}$$

z čehož lehce vyjádříme

$$\operatorname*{lse}_{i=1}^{n}(x_i) = \operatorname*{lse}_{i=1}^{n}(x_i - \max(x)) + \max(x). \qquad (2.6)$$

Byli jsme tedy schopni zapsat funkci `logsumexp` pomocí funkcí `broadcast` a `mapreduce`, tudíž jsme schopni provést sečtení logaritmů věrohodnosti součtovými uzly. Výpočet se poté opět díky multiple dispatch a definicím z knihovny `CUDA.jl` specializuje na efektivní metody pro GPU.

2.3.2.2 `_logpdf` pro součinovou vrstvu

Pseudo-algoritmus metody `_logpdf` pro součinovou vrstvu je popsán algoritmem 8. Tato metoda implementuje (1.18). Opět není třeba psát kernely, jelikož jak řezy pole, tak sčítání vektorů jsou implementovány v `CUDA.jl`.

Algoritmus 8: Metoda `_logpdf` pro součinovou vrstvu.

1 `_logpdf(`PL, X`)`
2 **for** *každého* L_i *potomka* PL **do**
3 $\quad X_i \leftarrow$ část X příslušná L_i
4 $\quad l \leftarrow l +$ `_logpdf(`L_i, X_i`)`
5 **end**
6 vrať l

2.3.2.3 `_logpdf` pro listovou vrstvu

Pseudo-algoritmus metody `_logpdf` pro listovou vrstvu je popsán algoritmem 9. Tato metoda implementuje (1.23). Výpočet logaritmů věrohodnosti lze zapsat pomocí funkce `mapreduce`, takže opět není třeba psát speciální kernel.

Algoritmus 9: Metoda `_logpdf` pro listovou vrstvu.

```
1 _logpdf(LL, X)
2 l ← mapreduce(X, y → −½ (y² + log (2π)), +, dims=2)
3 vrať l
```

2.3.3 Zpětná propagace

Už víme, jak vyčíslit logaritmus věrohodnosti SPTN pro matici pozorování. Nyní potřebujeme být schopni spočíst derivace všech parametrů sítě, abychom je poté mohli optimalizovat, například pomocí stochastického sestupu po gradientu, nebo algoritmu ADAM [12]. O výpočet derivací se z velké části stará knihovna `Flux.jl`. Výpočet se spoléhá na pravidla pro zpětné derivace, což jsou pouze algoritmické zápisy pravidel pro výpočty derivací a zpětnou propagaci gradientů ze sekce 1.3.2. Tyto pravidla nám umožňuje nezávisle na použité knihovně pro automatické derivace definovat knihovna `ChainRules.jl` pomocí definování nových metod pro funkci `rrule`. Argumenty `rrule` jsou funkce f, pro kterou píšeme pravidlo pro zpětnou derivaci, a dále všechny argumenty f, které označíme $x = (x_1, x_2, \ldots, x_n)$. Jako výstup dostáváme uspořádanou dvojici, kde první prvek je hodnota $f(x)$, a druhý prvek je funkce $\Delta^{(f(x))} \rightarrow \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \ldots, \frac{\partial L}{\partial x_n}\right)$. Při optimalizaci parametrů se tedy místo normálního výpočtu volají funkce `rrule`, jejich první výstup se použije pro vyčíslení hodnoty ztrátové funkce, a zároveň se během výpočtu řetězí funkce z druhého výstupu do jedné velké funkce, která nakonec spočte všechny derivace. Tento přístup také umožňuje uložení mezivýsledků vyčíslování ztrátové funkce, které budou potřeba při výpočtu derivací.

Knihovna `ChainRules.jl` obsahuje definice zpětných pravidel pro mnoho základních funkcí, není tedy třeba vše definovat manuálně. `Flux.jl` nepodporuje automatické derivování funkcí, které modifikují své vstupy in-place. Jelikož transformace pozorování transformačními uzly probíhá in-place, je třeba napsat zpětná pravidla pro algoritmy 3, 4, 5 a 6. Implementace `mapreduce` také používá in-place operace, je tedy třeba napsat zpětná pravidla pro všechny naše funkce, které `mapreduce` využívají, jmenovitě `logsumexp`, výpočet logaritmů věrohodností v listovém uzlu a výpočet logaritmu absolutních hodnot determinantů.

2.3.3.1 Transformace pozorování

Pro výpočet derivací ztrátové funkce vzhledem k parametrům matic v Butterfly rozkladu a parametrům matic D potřebujeme znát mezivýsledky jejího výpočtu. O uložení těchto výsledků se stará knihovna `Flux.jl`, ovšem ukládá je pouze jako ukazatel. Jelikož výpočet transformace probíhá in-place, tak si tyto mezivýsledky přepisujeme. Pole je zkopírováno až na konci transformace, takže pro každou transformaci zůstane uložená hodnota $UDVx+b$. Zde využijeme snadné inverze celé transformace, což nám umožní si mezivýsledky zpětně dopočítat. Tento přístup sice zvýší výpočetní náročnost, ovšem umožní nám ušetřit velké množství paměti, jelikož bude nutné si pamatovat pouze jeden mezivýsledek za jednu celou transformaci, oproti ukládání mezivýsledku po každém násobení Butterfly vrstvou. Každé pravidlo tedy bude začínat inverzí funkce pro přichystání mezivýsledku, následně spočteme derivace vůči parametrům, a nakonec zpětně propagujeme gradient.

Začněme definicí pravidla pro zpětnou derivaci pro jednoduchou Butterfly vrstvu. Nejprve potřebujeme funkci pro inverzi násobení jednoduchou Butterfly vrstvou. Tato funkce je popsána algoritmem 10. Od funkce `bfmulaxl` se liší pouze tím, že se změní znaménko úhlu pro jednotlivé givensovy rotace, tudíž je paralelizována stejným způsobem a její kernel je spouštěn se stejnou konfigurací CUDA bloků a mřížky.

Algoritmus 10: Násobení tenzoru X inverzí jednoduché Butterfly vrstvy.

```
1  bfinvmulaxl(bfl, X)
2  for každou matici X_{·,·,k} v X do
3  │  for každou Givensovu rotaci g_{m,k} s parametry θ_{m,k}, i_{m,k}, j_{m,k} v bfl do
4  │  │  for každý vektor X_{l,·,k} v X_{·,·,k} do
5  │  │  │  tmp_{l,i_{m,k},k} ← cos(−θ_{m,k}) · X_{l,i_{m,k},k} − sin(−θ_{m,k}) · X_{l,j_{m,k},k}
6  │  │  │  tmp_{l,j_{m,k},k} ← sin(−θ_{m,k}) · X_{l,i_{m,k},k} + cos(−θ_{m,k}) · X_{l,j_{m,k},k}
7  │  │  │  X_{l,i_{m,k},k} ← tmp_{l,i_{m,k},k}
8  │  │  │  X_{l,j_{m,k},k} ← tmp_{l,j_{m,k},k}
9  │  │  end
10 │  end
11 end
```

Následně potřebuje být schopni spočíst derivace ztrátové funkce vůči parametrům givensových rotací v Butterfly vrstvě a zpětně propagovat gradient. Výpočet derivací probíhá podle (1.91). Zpětná propagace gradientu probíhá podle (1.92). Jelikož pro ortogonální matici Q platí $Q^{-1} = Q^T$, lze funkci `bfinvmulax` použít i pro zpětnou propagaci gradientu. Celý výpočet pro jednoduchou Butterfly vrstvu je popsán algoritmem 11, kde X je uložený mezivýsledek z vyčíslení ztrátové funkce, $\nabla\theta$ je pole pro uložení derivací vzhledem k úhlům rotací a Δ je příchozí gradient.

Algoritmus 11: Výpočet derivací vůči parametrům a zpětná propagace gradientu pro násobení tenzoru X jednoduchou Butterfly vrstvou.

```
1  ∇bfmulaxl(bfl, X, Δ)
2  inicializuj dvojrozměrné pole ∇θ nulami
3  bfinvmulaxl(bfl, X)
4  for každou matici X_{·,·,k} v X do
5  │  for každou Givensovu rotaci g_{m,k} s parametry θ_{m,k}, i_{m,k}, j_{m,k} v bfl do
6  │  │  for každý vektor X_{l,·,k} v X_{·,·,k} do
7  │  │  │  ∇θ_{m,k} ← ∇θ_{m,k} + Δ_{l,i,k}(− sin(θ_{m,k}) · x_{l,i,k} − cos(θ_{m,k}) · x_{l,i,k})
8  │  │  │  ∇θ_{m,k} ← ∇θ_{m,k} + Δ_{l,j,k}(− sin(θ_{m,k}) · x_{l,j,k} + cos(θ_{m,k}) · x_{l,j,k})
9  │  │  end
10 │  end
11 end
12 bfinvmulaxl(bfl, Δ)
13 vrať (∇θ, Δ)
```

V algoritmu 11 jsou opět barevně zvýrazněné smyčky, které jsou v implementaci na GPU vypočteny paralelně. Funkce je rozdělena na tři GPU volání, dvě pro kernely pro funkci `bfinvmulax`, a jedno pro výpočet derivací smyčkami na řádcích 4 až 11 algoritmu 11. Kernel pro derivace je spouštěn v konfiguraci, kde CUDA bloky jsou dvourozměrná pole vláken, a CUDA mřížka je dvourozměrné pole bloků, podobně jako kernel pro funkci `bfmulaxl`. První blokový index určuje index l vektoru. Druhý blokový index určuje index m rotace. Velikost CUDA bloku je $(32, 4)$, celkem tedy 128 vláken rozdělených do čtyř warpů. Jelikož první blokový index určuje index vektoru l, tak vlákna v jednom warpu potřebují načíst prvek na tom samém místě svých vektorů. Tyto prvky jsou v paměti uloženy sekvenčně, dochází

tedy ke sdružování přístupů do paměti. Je-li vektorů více než 32, tak žlutá smyčka není plně paralelizována, provádí se po blocích velikosti 32. Výpočet probíhá paralelně, a vždy celý warp přičítá na stejné místo v paměti. Pokud by toto nebylo ošetřeno, jednotlivé zápisy do paměti by mezi sebou interferovaly, a výpočet by neproběhl správně. Vlákna ve warpu tedy akumulují hodnotu ve svém registru, a následně atomicky přičtou výsledek do globální paměti. První mřížkový index slouží ke spuštění dostatečného množství bloků tak, aby byly pokryty všechny rotace. Druhý mřížkový index určuje index k matice $X_{\cdot,\cdot,k}$.

Pokračujme definicí pravidla pro zpětnou derivaci pro Butterfly multivrstvu. Protože chceme využít sdílené paměti GPU, není možné výpočet rozdělit do tří kernelů, jako tomu je u `∇bfmulaxl`. Celý výpočet proběhne v jednom kernelu, kde se nejprve načtou data do sdílené paměti, následně se provede výpočet pro všechny Butterfly vrstvy v Butterfly multivrstvě, a nakonec se všechny výsledky uloží zpět do globální paměti. Výpočet je popsán algoritmem 12, kde X je uložený mezivýsledek z vyčíslení ztrátové funkce, $\nabla\theta$ je pole pro uložení gradientu inicializované nulami a Δ je příchozí gradient.

V algoritmu 12 jsou opět barevně zvýrazněné smyčky, které jsou v implementaci na GPU vypočteny paralelně. Kernel pro funkci `bfmulaxml` je spouštěn v konfiguraci, kde CUDA bloky jsou dvourozměrná pole vláken a CUDA mřížka je trojrozměrné pole bloků. Jeden CUDA blok provádí výpočet jednoho Butterfly bloku pro 32 vektorů. První blokový index určuje index m vektoru. Druhý blokový index určuje givensovu rotaci g. Velikost CUDA bloku je $(32, 4)$, celkem tedy 128 vláken rozdělených do čtyř warpů. Pokud je v jedné Butterfly vrstvě Butterfly bloku více rotací než je velikost druhé dimenze CUDA bloku, poté výpočty pro rotace provede CUDA blok sekvenčně. Vlákna opět přistupují k sekvenčně uloženým datům, dochází tedy ke slučování přístupů do paměti. Výpočet probíhá paralelně, a podobně jako u funkce `∇bfmulaxl` vždy celý warp přičítá na stejné místo v paměti. Zde se nejdříve zredukuje výsledek uvnitř warpu, a následně první vlákno warpu atomicky přičte výsledek do globální paměti. První mřížkový index slouží ke spuštění dostatečného množství bloků tak, aby byly pokryty všechny vektory. Druhý mřížkový index slouží ke spuštění CUDA bloku pro každý Butterfly blok. Třetí mřížkový index určuje index k matice X_k.

Dále potřebujeme pravidlo pro zpětnou derivaci pro násobení diagonálními maticemi D. Algoritmus 13 implementuje (1.94) pro výpočet derivace, a (1.95) pro zpětnou propagaci gradientu. Opět není třeba psát kernely, jelikož celá funkce je snadno zapsatelná pomocí funkcí `broadcast` a `mapreduce` způsobem popsaným v algoritmu 14.

Poslední pravidlo pro transorfmaci $UDVx + b$ je pro přičtení vektorů b. Algoritmus 15 implementuje (1.84) pro výpočet derivace, a (1.85) pro zpětnou propagaci gradientu. Celá funkce je opět snadno zapsatelná pomocí funkcí `broadcast` a `mapreduce` způsobem popsaným v algoritmu 16, a tedy není třeba psát kernely.

2.3.3.2 `logsumexp`

Vážený součet logaritmů věrohodnosti jsme byli schopni zapsat pomocí funkcí `logsumexp` jako (2.4). Není tedy třeba implementovat jedno složité pravidlo pro zpětnou derivaci pro celý vážený součet popsané rovnostmi (1.73), (1.74) a (1.75), stačí pouze pravidlo pro funkci `logsumexp` samotnou, o zbytek se postará automatické derivování. Funkce `logsumexp` samotná nemá žádné parametry, potřebujeme tedy pouze umět zpětně propagovat gradient. Odvoďme nyní s využitím (1.70) pravidlo pro zpětnou propagaci gradientu

$$\Delta_j^{(x)} = \frac{\partial L}{\partial x_j} = \frac{\partial L}{\partial \operatorname{lse}_{i=1}^n(x)} \frac{\partial \operatorname{lse}_{i=1}^n(x)}{\partial x_j} = \Delta^{\left(\operatorname{lse}(x)_{i=1}^n\right)} \left(\sum_{i=1}^n \exp(x_i) \right)^{-1} \exp\left(x_j\right), \quad \forall j \in \{1, \ldots, n\}. \tag{2.7}$$

Nyní můžeme zapsat algoritmus 17, implementující (2.7).

Algoritmus 12: Výpočet derivací vůči parametrům a zpětná propagace gradientu pro násobení tenzoru X Butterfly multivrstvou.

```
1  ∇bfmulaxml(bfml, X, ∇θ, Δ)
2  inicializuj trojrozměrné pole ∇θ nulami
3  for každý Butterfly blok B v bfml do
4      for každou matici X_{·,·,k} v X do
5          načti část X_{·,·,k} příslušnou B do sdílené paměti
6          načti část Δ_{·,·,k} příslušnou B do sdílené paměti
7          for pro každou Butterfly vrstvu rotací g_{·,l,k} v bfml do
8              for každou rotaci g_{m,l,k} s parametry θ_{m,l,k}, i_{m,l,k}, j_{m,l,k} v g_{·,l,k} do
9                  for každý vektor X_{n,·,k} v X_{·,·,k} do
                       // spočti sin a cos úhlu
10                     s_{m,l,k} ← sin(θ_{m,l,k})
11                     c_{m,l,k} ← cos(θ_{m,l,k})
                       // přichystej mezivýsledek
12                     tmp_{n,i_{m,l,k},k} ← c_{m,l,k} · X_{n,i_{m,l,k},k} + s_{m,l,k} · X_{n,j_{m,l,k},k}
13                     tmp_{n,j_{m,l,k},k} ← −s_{m,l,k} · X_{n,i_{m,l,k},k} + c_{m,l,k} · X_{n,j_{m,l,k},k}
14                     X_{n,i_{m,l,k},k} ← tmp_{n,i_{m,l,k},k}
15                     X_{n,j_{m,l,k},k} ← tmp_{n,j_{m,l,k},k}
                       // spočti derivace
16                     ∇θ_{m,l,k} ← ∇θ_{m,l,k} + Δ_{n,i_{m,l,k},k}(−s_{m,l,k} · X_{n,i_{m,l,k},k} − c_{m,l,k} · X_{n,j_{m,l,k},k})
17                     ∇θ_{m,l,k} ← ∇θ_{m,l,k} + Δ_{n,j_{m,l,k},k}(−s_{m,l,k} · X_{n,j_{m,l,k},k} + c_{m,l,k} · X_{n,i_{m,l,k},k})
                       // zpětně propaguj Δ
18                     tmp_{n,i_{m,l,k},k} ← c_{m,l,k} · D_{n,i_{m,l,k},k} + s_{m,l,k} · D_{n,i_{m,l,k},k}
19                     tmp_{n,j_{m,l,k},k} ← −s_{m,l,k} · D_{n,i_{m,l,k},k} + c_{m,l,k} · D_{n,i_{m,l,k},k}
20                     Δ_{n,i_{m,l,k},k} ← tmp_{n,i_{m,l,k},k}
21                     Δ_{n,j_{m,l,k},k} ← tmp_{n,j_{m,l,k},k}
22                 end
23             end
24         end
25         ulož část X_{·,·,k} příslušnou B zpět do sdílené paměti
26         ulož část Δ_{·,·,k} příslušnou B zpět do sdílené paměti
27     end
28 end
29 vrať (∇θ, Δ)
```

Algoritmus 13: Výpočet derivací vůči parametrům a zpětná propagace gradientu pro násobení tenzoru X diagonálními maticemi D.

1 ∇`dmulax`(D, X, Δ)
2 inicializuj dvojrozměrné pole ∇D nulami
3 **for** *každou matici* $X_{\cdot,\cdot,k}$ *v* X **do**
4   **for** *každý vektor* $X_{i,\cdot,k}$ *v* $X_{\cdot,\cdot,k}$ **do**
5   **for** *každý prvek* $X_{i,j,k}$ *vektoru* $X_{i,\cdot,k}$ **do**
6             $X_{i,j,k} \leftarrow \frac{X_{i,j,k}}{D_{j,k}}$
7             $\nabla D_{j,k} \leftarrow \nabla D_{j,k} + \Delta_{i,j,k} X_{i,j,k}$
8             $\Delta_{i,j,k} \leftarrow D_{j,k} \Delta_{i,j,k}$
9   **end**
10   **end**
11 **end**
12 vrať $(\nabla D, \Delta)$

Algoritmus 14: Algoritmus 13 zapsaný pomocí `broadcast` a `mapreduce`.

1 ∇`dmulax`(D, X, Δ)
2 inicializuj dvojrozměrné pole ∇D nulami
3 $\tilde{D} \leftarrow$ matice D interpretovaná jako 3D pole s velikostí první dimenze rovnou jedné
4 $\nabla \tilde{D} \leftarrow$ matice ∇D interpretovaná jako 3D pole s velikostí první dimenze rovnou jedné
5 $X \leftarrow X./\tilde{D}$
  `// označme sum`(x, $dims$) `= mapreduce`(x, $identita$, $+$, $dims$)
6 $\nabla \tilde{D} \leftarrow$ `sum`($\Delta.*X$, $dims{=}1$)
7 $\Delta \leftarrow \tilde{D}.*\Delta$
8 $\nabla D \leftarrow$ matice $\nabla \tilde{D}$ interpretovaná jako 2D pole vynecháním první dimenze
9 vrať $(\nabla D, \Delta)$

Algoritmus 15: Výpočet derivací vůči parametrům a zpětná propagace gradientu pro přičtení vektorů b k tenzoru X.

1 ∇`badd`(b, X, Δ)
2 inicializuj dvojrozměrné pole ∇b nulami
3 **for** *každou matici* $X_{\cdot,\cdot,k}$ *v* X **do**
4   **for** *každý vektor* $X_{i,\cdot,k}$ *v* $X_{\cdot,\cdot,k}$ **do**
5   **for** *každý prvek* $X_{i,j,k}$ *vektoru* $X_{i,\cdot,k}$ **do**
6             $X_{i,j,k} \leftarrow X_{i,j,k} - b_{j,k}$
7             $\nabla b_{j,k} \leftarrow \nabla b_{j,k} + \Delta_{i,j,k}$
8   **end**
9   **end**
10 **end**
11 vrať $(\nabla b, \Delta)$

Algoritmus 16: Algoritmus 15 zapsaný pomocí `broadcast` a `mapreduce`.

1 ∇`dmulax`(b, X, ∇b, Δ)
2 inicializuj dvojrozměrné pole ∇b nulami
3 $\tilde{b} \leftarrow$ matice b interpretovaná jako 3D pole s velikostí první dimenze rovnou jedné
4 $\nabla\tilde{b} \leftarrow$ matice ∇b interpretovaná jako 3D pole s velikostí první dimenze rovnou jedné
5 $X \leftarrow X.\text{-}\tilde{b}$
6 $\nabla\tilde{b} \leftarrow$ `sum`(Δ, *dims=1*)
7 $\nabla b \leftarrow$ matice $\nabla\tilde{b}$ interpretovaná jako 2D pole vynecháním první dimenze
8 vrať $(\nabla b, \Delta)$

Algoritmus 17: Zpětná propagace gradientu pro funkci `logsumexp`.

1 ∇`logsumexp`(x, *dims*, Δ)
2 $xm \leftarrow$ `mapreduce`(x, *Identita*, max, *dims*)
3 exn = `exp.`(x.-xm)
4 $\Delta \leftarrow \Delta.{*}exn./$`sum`($exn$, *dims*)
5 vrať Δ

2.3.3.3 Listové uzly

Listové uzly nemají žádné parametry, je tedy třeba pouze zpětně propagovat gradient podle (1.67), což popisuje algoritmus 18.

Algoritmus 18: Zpětná propagace gradientu pro funkci `_logpdf` pro listový uzel.

1 ∇`_logpdf`(X, Δ)
2 vrať $-\Delta.{*}X$

2.3.3.4 Logaritmus absolutních hodnot determinantů

Pravidlo pro zpětnou derivaci funkce `lad` implementuje (1.81), popsáno algoritmem 19.

2.4 Mapování stromů

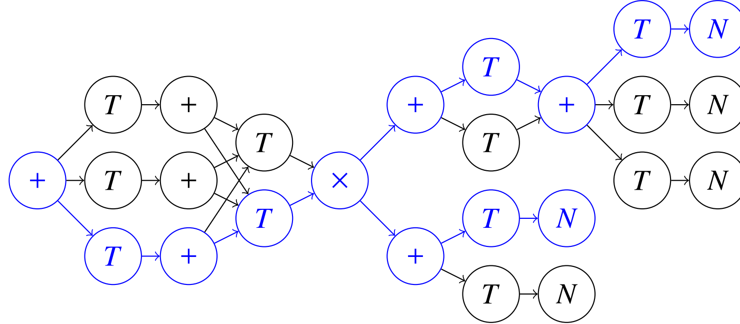
SPTN reprezentuje směs pravděpodobnostních veličin. Každá z komponent směsi je reprezentována stromem, což je znázorněno na obrázku 2.6. Abychom mohli počítat věrohodnost pozorování vzhledem ke komponentě a generovat náhodné nové pozorování, musíme umět tyto stromy nějak popsat.

Stromy budeme popisovat pomocí vnořených uspořádaných n-tic, kde každá n-tice bude odpovídat jedné z SPTN vrstev. Listová vrstva bude reprezentována prázdnou n-ticí. Součinová vrstva bude reprezentována n-ticí, jejíž prvky budou n-tice reprezentující její potomky. Součtově-transformační vrstva bude reprezentována n-ticí, jejíž první prvek bude určovat komponentu součtového uzlu, a druhý prvek bude n-tice reprezentující jejího potomka.

Algoritmus je strukturován podobně jako funkce `_logpdf`. Skládá se z funkce `maptree`, která má stejnou implementaci pro všechny typy SPTN vrstev, a provádí přípravu pro rekurzivní výpočet, který provede funkce `_maptree`. Jelikož nepotřebujeme jenom grafový popis stromu, ale i pravděpodobnost komponenty jím reprezentované, tak algoritmus během mapování počítá i pravděpodobnost. Součtové

Algoritmus 19: Pravidlo pro zpětnou derivaci funkce `lad`.

1 ∇`lad`(D, Δ)
2 vrať $-\Delta./D$



Obrázek 2.6: SPTN s modře vyznačeným stromem.

vrstvy ovšem neukládají pravděpodobnosti svých jednotlivých komponent přímo, ale pouze hodnoty, na které je třeba použít funkci `softmax` pro získání pravděpodobností. Tyto hodnoty také mohou být uloženy v paměti GPU, kam má CPU velmi pomalý přístup. Z tohoto důvodu jsou na začátku výpočtu parametry zkopírovány do RAM a přepočteny na pravděpodobnosti pomocí funkce `cpuweights`, která má tři metody, jednu pro každý typ SPTN vrstvy. Abychom zachovali strukturu sítě, tak pravděpodobnosti komponent uložíme do vnořených n-tic stejné struktury, jako pro popis stromů. Metoda pro součtově-transformační vrstvu je popsána algoritmem 20. Metoda pro součinovou vrstvu je popsána algoritmem 21. Metoda pro listovou vrstvu je popsána algoritmem 22.

Algoritmus 20: Metoda funkce `cpuweights` pro součtově-transformační vrstvu.

1 `cpuweights`(ST)
2 $x \leftarrow$ kopie vah komponent ST uložená v RAM
3 $ps \leftarrow$ `softmax`$(x, dims{=}2)$
4 $w_c \leftarrow$ `cpuweights`$(potomek\ ST)$
5 vrať (ps, w_c)

Připravme si ještě jednu pomocnou funkci, a to `treenum`, která bude mít za argument vnořenou n-tici získanou funkcí `cpuweights`, a bude vracet počet stromů v síti reprezentované touto n-ticí. Metoda pro součtově-transformační vrstvu je popsána algoritmem 23. Metoda pro součinovou vrstvu je popsána algoritmem 24. Metoda pro listovou vrstvu je popsána algoritmem 25.

Nyní přejděme k samotným funkcím `maptree` a `_maptree`. Funkce `maptree` pomocí `cpuweights` vytvoří reprezentaci sítě s pravděpodobnostmi komponent a pomocí `treenum` spočte celkový počet komponent n. Následně vygeneruje náhodné celé číslo $i \in \{1, \ldots, n\}$, a poté zavolá funkci `_maptree`, která rekurzivně vytvoří n-tici popisující i-tý strom v síti. Funkce `maptree` je popsána algoritmem 26.

Dále popišme rekurzivní funkci `_maptree`. Tato funkce má 4 argumenty. První argument je počet stromů v síti popsané třetím argumentem. Druhý argument je index stromu v rámci sítě popsané třetím argumentem. Třetí argument je vnořená n-tice získaná funkcí `cpuweights`. Poslední argument je index součtového uzlu pro následující součtově-transformační vrstvu. Návratová hodnota je uspořádaná dvojice, jejíž první prvek je pravděpodobnost komponenty reprezentované druhým prvkem.

Algoritmus 21: Metoda funkce `cpuweights` pro součinovou vrstvu.

1 `cpuweights(`*PL*`)`
2 vrať `map(cpuweights,` *potomci L*`)`

Algoritmus 22: Metoda funkce `cpuweights` pro listovou vrstvu.

1 `cpuweights(`*LL*`)`
2 vrať prázdnou uspořádanou n-tici

Algoritmus 23: Metoda funkce `treenum` pro součtově-transformační vrstvu.

1 `treenum(`w`)`
2 $n \leftarrow$ velikost druhé dimenze pole w_1
3 vrať n*`treenum(`w_2`)`

Algoritmus 24: Metoda funkce `treenum` pro součinovou vrstvu.

1 `treenum(`w`)`
2 $n \leftarrow$ počet prvků w
3 vrať $\prod_{i=1}^{n}$ `treenum(`w_i`)`

Algoritmus 25: Metoda funkce `treenum` pro listovou vrstvu.

1 `treenum(`w`)`
2 vrať 1

Algoritmus 26: Popis funkce `maptree`.

1 `maptree(`L`)`
2 $w \leftarrow$ `cpuweights(`L`)`
3 $n \leftarrow$ `treenum(`w`)`
4 $i \leftarrow$ náhodné číslo $\in \{1, \ldots, n\}$
5 $(p, t) \leftarrow$ `_maptree(`*n, i, w, 1*`)`
6 vrať (p, t)

Funkce `_maptree` je popsána algoritmem 27 pro součtově-transformační vrstvu. Pokud součtově-transformační vrstva sdílí parametry součtových uzlů, je třeba poslední argument změnit na hodnotu 1. Abychom určili index podstromu vzhledem k podsítí, využijeme funkci `CartesianIndices`. Tato funkce má za argument uspořádanou n-tici přirozených čísel n, a vrací pole uspořádaných n-tic, kde každá n-tice odpovídá indexu vícerozměrného pole s dimenzemi určenými n. Toto pole můžeme lineárně indexovat, a snadno tak index i rozložit na hodnotu určující transformační uzel v součtově-transformační vrstvě, a na index podstromu v podsíti. Poté už snadno pomocí rekurze zmapujeme podstrom, a získanou uspořádanou n-tici následně rozšíříme o index transformace aktuální vrstvy.

Algoritmus 27: Metoda funkce `_maptree` pro součtově-transformační vrstvu.

1 `_maptree(`n, i, w, sid`)`
2 **if** *součtově-transformační vrstva sdílí parametry součtových uzlů* **then**
3 | $sid \leftarrow 1$
4 **end**
5 $tn \leftarrow$ počet transformačních uzlů v součtově-transformační vrstvě
6 $m \leftarrow \frac{n}{tn}$ `// počet stromů v potomkovi aktuální součtově-transformační vrstvy`
7 $idxs \leftarrow$ `CartesianIndices(`m, tn`)`
8 $idx \leftarrow idxs_i$
9 $p, t_c \leftarrow$ `_maptree(`m, idx_1, w_2, idx_2`)`
10 $t \leftarrow (idx_2, t_c)$
11 $p \leftarrow p*(w_1)_{sid,idx_2}$
12 vrať (p, t)

Funkci `_maptree` pro součinovou vrstvu popisuje algoritmus 28. Indexy podstromů vzhledem k jednotlivým potomkům nalezneme opět pomocí funkce `CartesianIndices`, jejímž argumentem bude tentokrát uspořádaná n-tice obsahující počty stromů v potomcích součinové vrstvy.

Algoritmus 28: Metoda funkce `_maptree` pro součinovou vrstvu.

1 `_maptree(`n, i, w, sid`)`
2 $ms \leftarrow$ `map(`$y \rightarrow$ `treenum(`y`)`, w`)`
3 $idxs \leftarrow$ `CartesianIndices(`ms`)`
4 $idx \leftarrow idxs_i$
5 **for** *každý prvek w_j ve w* **do**
6 | $p_j, t_j \leftarrow$ `_maptree(`ms_j, idx_j, w_j, sid`)`
7 **end**
8 $t \leftarrow (t_1, t_2, \dots, t_n)$
9 $k \leftarrow$ počet potomků aktuální součinové vrstvy
10 $p \leftarrow \prod_{j=1}^{k} p_j$
11 vrať (p, t)

Funkci `_maptree` pro listovou vrstvu popisuje algoritmus 29.

Algoritmus 29: Metoda funkce `_maptree` pro listovou vrstvu.

1 `_maptree(`n, i, w, sid`)`
2 vrať $(1, \text{prázdná n-tice})$

Nyní jsme tedy schopni popsat libovolný strom reprezentující jednu komponentu v směsi pravděpodobnostních veličin reprezentovaných SPTN, a znát pravděpodobnost, že náhodné pozorování bude pocházet z této komponenty. Implementována je i funkce `maptrees`, která zmapuje všechny stromy, uloží je do pole, a to následně seřadí podle pravděpodobnosti jednotlivých komponent.

2.5 Inference vzhledem ke stromu

Víme-li, ze které komponenty pozorování pochází, můžeme počítat logaritmus věrohodnosti vzhledem ke stromu popisujícímu tuto komponentu. Ve výpočtu se pouze mění funkce součtově-transformační vrstvy, která nyní provádí jednu transformaci určenou stromem, a nikoliv všechny transformace, které by potom sečetla součtovými uzly. Známe-li tedy strom, vůči kterému logaritmus věrohodnosti počítáme, mění se (2.4) na

$$st(x) = \log\left(p_i(U_i D_i V_i x + b_i)\right) + \operatorname{lad}\left(D_i\right), \tag{2.8}$$

kde i je index transformačního uzlu určený stromem.

Výpočet je implementován podobně jako výpočet plného logaritmu věrohodnosti popsaný v sekci 2.3. Funkce `treelogpdf` připraví výpočet na rekurzi zkopírováním dat na GPU a transpozicí pozorování, a následně zavolá funkci `_treelogpdf`, která se postará o samotný rekurzivní výpočet. Metoda funkce `_treelogpdf` pro součtově-transformační vrstvu je popsána algoritmem 30. Oproti `_logpdf` má navíc argument specifikující strom, který jsme získali pomocí funkce `maptree`. Transformace probíhá pomocí

Algoritmus 30: Metoda `_treelogpdf` pro součtově-transformační vrstvu.

1 `_treelogpdf(`ST, X, t`)`
2 transformuj tenzor X transformačním uzlem s indexem t_1
3 $l \leftarrow$ `_treelogpdf(`*potomek* ST, X, t_2`)`
4 přičti logaritmus absolutní hodnoty determinantu transformačního uzlu s indexem t_1 k l
5 vrať l

funkcí popsaných algoritmy 2, 3, 4, 5 a 6, které nyní neobsahují žlutě zvýrazněnou smyčku přes všechny matice $X_{\cdot,\cdot,k}$ v X, jelikož X je nyní pouze jedna matice, a nikoliv tenzor. Index transformačního uzlu k, který tato smyčka určovala, je nyní předán funkcím jako parametr.

Metoda funkce `_treelogpdf` pro součinovou vrstvu je popsána algoritmem 31. Tato metoda je skoro identická metodě `_logpdf` pro součinovou vrstvu, pouze navíc rozdistribuuje podstromy svým potomkům.

Algoritmus 31: Metoda `_treelogpdf` pro součinovou vrstvu.

1 `_treelogpdf(`PL, X, t`)`
2 **for** *každého* L_i *potomka* PL **do**
3     $X_i \leftarrow$ část X příslušná L_i
4     $l \leftarrow l+$ `_treelogpdf(`L_i, X_i, t_i`)`
5 **end**
6 vrať l

Metoda funkce `_treelogpdf` pro listovou vrstvu je identická metodě `_logpdf`, a je popsána algoritmem 32.

Algoritmus 32: Metoda `_treelogpdf` pro listovou vrstvu.

1 `_treelogpdf`(LL, X, t)
2 $l \leftarrow$ `mapreduce`(X, $y \to -\frac{1}{2}\left(y^2 + \log(2\pi)\right)$, +, *dims*=2)
3 vrať l

2.6 Samplování

Z SPTN lze generovat i nová náhodná pozorování. Nejdříve je třeba náhodně vygenerovat nebo specifikovat komponentu, ze které generované pozorování pochází. K tomuto lze použít výstup funkce `maptrees`, která nám zmapuje všechny stromy odpovídající komponentám a spočte jejich pravděpodobnosti. Jelikož listové vrstvy implementují pouze vícerozměrné normální náhodné rozdělení, lze snadno vygenerovat náhodné vektory. Tyto vektory je poté třeba transformovat inverzními transformacemi v transformačních uzlech dané komponenty, a pospojovat pomocí součinových uzlů.

Pro samplování byla přidána metoda pro funkci `rand`. Tato metoda má za argument SPTN, a volitelně lze specifikovat strom reprezentující komponentu, ze které mají náhodné pozorování pocházet, a počet pozorování, který chceme samplovat. Pokud volitelné argumenty nejsou specifikovány, tak je vygenerováno jedno pozorování z náhodné komponenty.

O samotný rekurzivní výpočet se stará funkce `_rand`. Tato funkce má opět tři metody pro jednotlivé typy SPTN vrstev. Metodu pro součtově-transformační vrstvu popisuje algoritmus 33. Zde potřebujeme

Algoritmus 33: Metoda `_rand` pro součtově-transformační vrstvu.

1 `_rand`(*ST, tree*)
2 $x \leftarrow$ `_rand`(*potomek ST*, t_2)
3 transformuj vektor x inverzí transformačního uzlu s indexem t_1
4 vrať x

inverzní transformaci jedním transformačním uzlem v součtově-transformační vrstvě. Tyto funkce jsou jednoduché modifikace funkcí popsaných algoritmy 2, 3, 4, 5 a 6, kde inverze je zajištěna negací úhlu pro Butterfly vrstvy a multivrstvy, dělením místo násobení pro diagonální matice D a odečítáním místo přičítání pro biasy b. Index transformace je specifikován parametrem, a tudíž není prováděna žlutě zvýrazněná smyčka, podobně jak bylo popsáno v sekci 2.5.

Metodu pro součinovou vrstvu popisuje algoritmus 34.

Algoritmus 34: Metoda `_rand` pro součinovou vrstvu.

1 `_rand`(*PL, tree*)
2 **for** *každého L_i potomka PL* **do**
3 | $x_i \leftarrow$ `_rand`(L_i, t_i)
4 **end**
5 spoj vektory x_i do jednoho vektoru x
6 vrať x

Metodu pro listovou vrstvu popisuje algoritmus 35.

Algoritmus 35: Metoda `_rand` pro listovou vrstvu.

1 `_rand`(*LL, tree*)
2 $x \leftarrow$ vektor obsahující náhodné pozorování se standardním gaussovským rozdělením
3 vrať x

2.7 Marginalizace

Marginalizace umožňuje počítat věrohodnost pro neúplná data. Pro získání marginálního rozdělení ovšem není vhodná struktura dat v SPTN. Využijeme zde faktu, že naše regularizovaná SPTN reprezentuje směs vícerozměrných normálních rozdělení. Pro vícerozměrné normální rozdělení je marginalizace jednoduchá. Mějme $\mathcal{N}(\mu, \Sigma)$, kde $\mu \in \mathbb{R}^k$ je vektor střední hodnoty a $\Sigma \in \mathbb{R}^{k,k}$ je kovarianční matice. Nechť $I \subset \{1, \ldots, k\}$ je množina indexů. Poté marginalizace vícerozměrného normálního rozdělení $\mathcal{N}(\mu, \Sigma)$ je $\mathcal{N}(\mu_I, \Sigma_{I,I})$, kde μ_I vznikne z vektoru μ ponecháním pouze těch prvků, jejichž indexy jsou v I. Matice $\Sigma_{I,I}$ vznikne z matice Σ podobným způsobem, ponecháním pouze těch prvků, jejichž indexy jak řádku tak sloupce jsou v I.

Abychom tohoto mohli využít, potřebujeme umět vyjádřit parametry μ, Σ komponenty reprezentované nějakým stromem. Logaritmus věrohodnosti pro vícerozměrné normální rozdělení má tvar

$$\log\left(f_{\mathcal{N}(\mu,\Sigma)}(x)\right) = -\frac{1}{2}\left((x-\mu)^T \Sigma^{-1} (x-\mu) + k\log(2\pi) + \log(|\Sigma|)\right). \tag{2.9}$$

Výpočet parametrů normálního rozdělení komponenty SPTN bude probíhat rekurzivně. Potřebujeme tedy odvodit pravidla pro rekurzivní výpočet přes transformační uzel, součinový uzel a listový uzel.

Začněme transformačním uzlem. Předpokládejme, že známe parametry μ a Σ_1 pro rozdělení potomka transformačního uzlu. Odvodíme parametry ν a Σ_2 pro rozdělení transformačního uzlu. Zaměřme se na první člen uvnitř závorky v (2.9)

$$(y-\mu)^T \Sigma_1^{-1} (y-\mu), \tag{2.10}$$

kde y je vektor pozorování vstupující do potomka aktuálního transformačního uzlu. Tento vektor jsme získali transformací $y = UDVx+b$. Pro přehlednost označme lineární část afinní transformace $T = UDV$, a poté dosaďme do (2.10) a upravme

$$(Tx+b-\mu)^T \Sigma^{-1} (Tx+b-\mu) = \left(x+T^{-1}(b-\mu)\right)^T T^T\Sigma_1^{-1}T\left(x+T^{-1}(b-\mu)\right). \tag{2.11}$$

Dosazením

$$\Sigma_2^{-1} = T^T\Sigma_1^{-1}T, \tag{2.12}$$

$$\nu = T^{-1}(\mu - b) \tag{2.13}$$

do (2.14) získáme

$$(x-\nu)^T \Sigma_2^{-1} (x-\nu), \tag{2.14}$$

což je stejného tvaru jako (2.10), takže pomocí (2.12) a (2.13) jsme schopni rekurzivně vypočítávat parametry normálního rozdělení.

Pokračujme součinovým uzlem. Předpokládejme, že známe parametry μ_i a Σ_i pro $\forall i \in \{1, \ldots, n\}$ indexy potomků součinového uzlu. Odvodíme parametry μ a Σ pro rozdělení součinového uzlu. Logaritmus věrohodnosti součinového uzlu je dle (1.18) a (2.9) roven

$$\log(p(x)) = -\frac{1}{2}\sum_{i=1}^{n}\left((x_i-\mu_i)^T \Sigma_i^{-1}(x_i-\mu_i) + k_i\log(2\pi) + \log(|\Sigma_i|)\right). \tag{2.15}$$

Označením

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mu = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \Sigma_1 & & & \\ & \Sigma_2 & & \\ & & \ddots & \\ & & & \Sigma_n \end{pmatrix}, \quad k = \sum_{i=1}^{n} k_i \tag{2.16}$$

lze (2.15) přepsat na

$$\log(p(x)) = -\frac{1}{2}\left((x-\mu)^T \Sigma^{-1} (x-\mu) + k \log(2\pi) + \log(|\Sigma|)\right), \tag{2.17}$$

což je ve tvaru logaritmu věrohodnosti normálního rozdělení, takže jsme schopni rekurzivně vypočítávat parametry pro součinové uzly.

Listovým uzlem končíme rekurzi, jelikož má známé parametry $\mu = 0$ a $\Sigma^{-1} = I$.

Přejděme nyní k implementaci těchto vzorců. Výpočet je opět rozdělen do dvou částí, první provede přípravu na rekurzi, a druhá provede samotný rekurzivní výpočet. Přípravná funkce je pojmenována `cache_tree`, a je popsána algoritmem 36. Jako argumenty má kořen sítě a uspořádanou dvojici obsahující strom popisující jednu komponentu a pravděpodobnost této komponenty. Návratové hodnoty jsou $Q = \Sigma^{-1}, \mu$ a pravděpodobnost komponenty. Ve výpočtu je využita optimalizace výpočtu Q. Abychom spočetli Q, opakujeme operaci (2.12) pro každou součtově-transformační vrstvu, přičemž začínáme s maticí identity. Lze počítat pouze výraz $U = IT_1T_2 \ldots T_n$, kde matice T_i jsou lineární části afinních transformací příslušející jednotlivým součtově-transformačním vrstvám, a následně získat matici Q pomocí $Q = U^T U$.

Algoritmus 36: Funkce `cache_tree`.

1 `cache_tree`$(L, (p, t))$
2 $U \leftarrow$ inicializuj matici identity
3 $\mu \leftarrow$ inicializuj vektor nul
4 `_cache_tree`(L, U, μ, t)
5 $Q = U^T U$
6 vrať (Q, μ, p)

Metoda `_cache_tree` pro součtově-transformační vrstvu implementující (2.12) a (2.13) je popsána algoritmem 37.

Algoritmus 37: Metoda funkce `_cache_tree` pro součtově-transformační vrstvu.

1 `_cache_tree`(ST, U, μ, t)
2 `_cache_tree`(*potomek* ST, U, μ, t_2)
3 aplikuj lineární část transformačního uzlu s indexem t_1 na U
4 aplikuj inverzi transformačního uzlu s indexem t_1 na μ

Metoda `_cache_tree` pro součinovou vrstvu implementující (2.16) je popsána algoritmem 38.

Metoda `_cache_tree` pro listovou vrstvu je prázdná funkce, a slouží pouze pro ukončení rekurze.

Nyní jsme schopni spočíst matici Q a vektor středních hodnot μ pro jednu komponentu SPTN. Pomocí funkce `cache_sptn` lze spočíst matice Q a vektory μ pro všechny komponenty v SPTN. Výsledek je poté uložen jako struktura obsahující pole matic Q, pole vektorů μ a pole pravděpodobností jednotlivých komponent. Jelikož původní SPTN může sdílet parametry, tak tímto výpočtem se mohou značně zvýšit paměťové nároky. Funkce `cache_sptn` umožňuje nastavit maximální počet komponent, případně

Algoritmus 38: Metoda funkce `_cache_tree` pro součinovou vrstvu.

```
1 _cache_tree(PL, U, μ, t)
2 for Každého potomka L_i SPTN vrstvy PL do
3     U_i ← čtvercová část matice U příslušející L_i
4     μ_i ← část vektoru μ příslušející L_i
5     _cache_tree(L_i, U_i, μ_i, t_i)
6 end
```

minimální pravděpodobnost komponent, které budou započítány. Při nastavení obou parametrů má přednost maximální počet komponent.

Abychom byli schopni spočíst logaritmus věrohodnosti, využijeme faktu, že síť můžeme nyní reprezentovat jako jediný součtový uzel, který má za potomky listové uzly, které nyní implementují obecné vícerozměrné normální rozdělení, nikoliv standardní. Můžeme tedy použít (1.16) a (2.9), což je velmi přímočaře implementováno v metodě funkce `logpdf` pro strukturu získanou funkcí `cache_sptn`. Tato metoda je popsána algoritmem 39. Funkce má za argumenty pole matic Qs, pole vektorů μs, pole pravděpodobností jednotlivých komponent ps, vektor pozorování x, a abychom mohli počítat logaritmus hustoty pravděpodobnosti pro marginální rozdělení, tak posledním argumentem je I pole pravdivostních hodnot určující indexy pro marginalizaci.

Algoritmus 39: Výpočet logaritmu věrohodnosti pro marginalizovanou SPTN.

1 `_cache_tree`(Qs, μs, ps, x, I)
2 **for** *Každé parametry i-tého normálního rozdělení* Q, μ **do**
3 $\quad l_i \leftarrow -\frac{1}{2}\left((x-\mu_I)^T Q_{I,I}(x-\mu_I) + k\log(2\pi) + \log\left(\left|Q_{I,I}\right|\right)\right)$
4 **end**
5 $l \leftarrow l + \log(ps)$
6 vrať `logsumexp`(l)

Kapitola 3

Výsledky

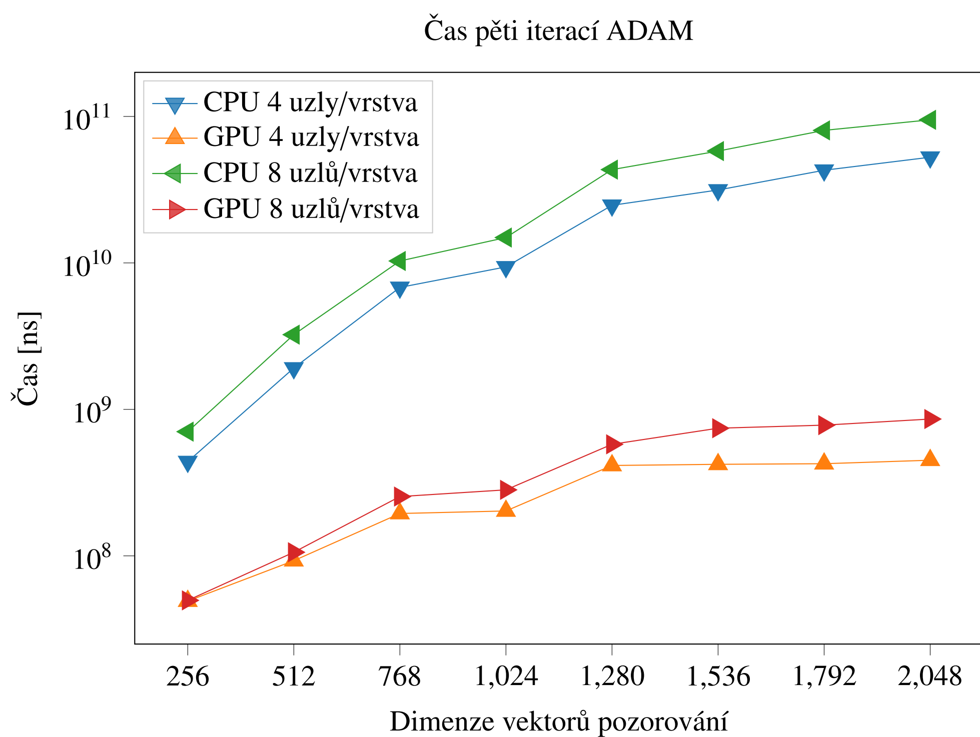
3.1 Paralelní urychlení

Pro měření paralelního urychlení bylo použito šestnácti-jádrové CPU AMD EPYC 7281@2.1GHz a GPU NVIDIA TESLA V100 (Volta) s 16GB HBM2 RAM. Urychlení bylo měřeno oproti čistě sekvenční implementaci běžící na jednom jádře procesoru. Jelikož urychlení závisí na velikosti sítě, bylo zvoleno několik konfigurací. SPTN byla vždy složena ze tří součtově-transformačních vrstev zakončených listovou vrstvou. V polovině případů měly všechny vrstvy 4 transformační uzly, a ve druhé polovině případů 8 transformačních uzlů. Parametry součtových uzlů sdíleny nebyly. Dimenze vektorů pozorování byla volena z množiny $\{256, 512, 768, 1024, 1280, 1536, 1792, 2048\}$. Data pro učení byla vygenerována náhodně. Byl měřen čas pěti iterací algoritmu ADAM [12], kde během jedné iterace výpočet probíhal s dvěma vektory pozorování. Naměřené hodnoty jsou zapsány v tabulce 3.1, spolu s vypočítaným paralelním urychlením. Časy z tabulky 3.1 jsou znázorněny na obrázku 3.1 a urychlení na obrázku 3.2.

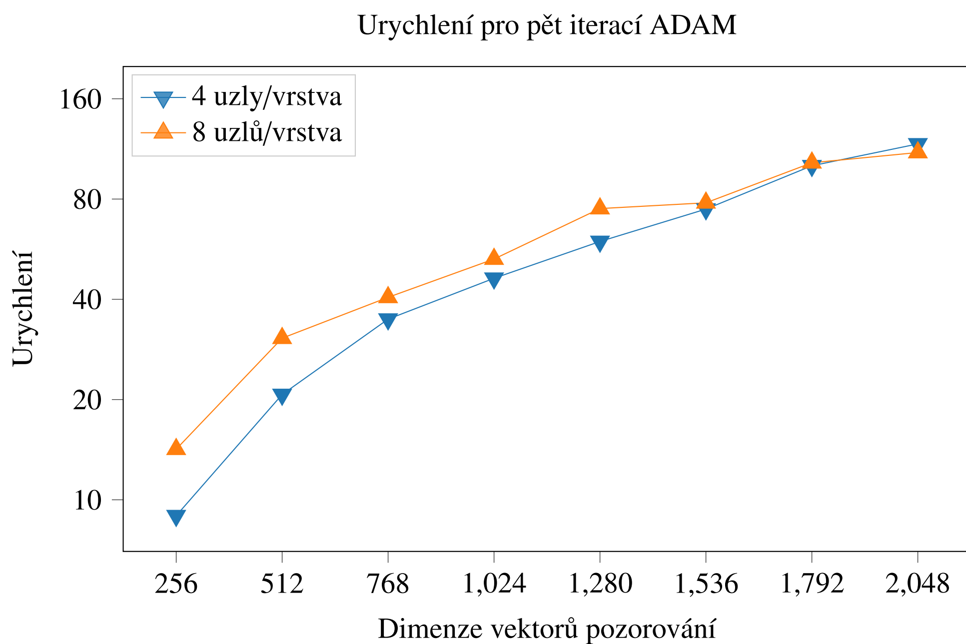
Naměřené urychlení se pohybuje mezi jedním a dvěma řády a roste s velikostí výpočtu, přičemž nejvyššího urychlení 117.11 bylo dosaženo pro pozorování o dimenzi 2048. Urychlení pro síť se čtyřmi uzly na vrstvu dosahovalo menších hodnot než pro síť s osmi uzly na vrstvu, ovšem s rostoucí dimenzí pozorování byl tento rozdíl stále méně významný.

	4 transformační uzly na vrstvu			8 transformačních uzlů na vrstvu		
dim	CPU	GPU	Urychlení	CPU	GPU	Urychlení
256	4.40×10^8	4.91×10^7	8.96	7.05×10^8	4.97×10^7	14.19
512	1.92×10^9	9.27×10^7	20.71	3.24×10^9	1.06×10^8	30.57
768	6.80×10^9	1.95×10^8	34.87	1.03×10^{10}	2.54×10^8	40.55
1024	9.39×10^9	2.03×10^8	46.26	1.49×10^{10}	2.82×10^8	52.84
1280	2.47×10^{10}	4.14×10^8	59.66	4.33×10^{10}	5.78×10^8	74.91
1536	3.15×10^{10}	4.22×10^8	74.64	5.80×10^{10}	7.44×10^8	77.96
1792	4.29×10^{10}	4.26×10^8	100.70	8.03×10^{10}	7.81×10^8	102.82
2048	5.27×10^{10}	4.50×10^8	117.11	9.48×10^{10}	8.59×10^8	110.36

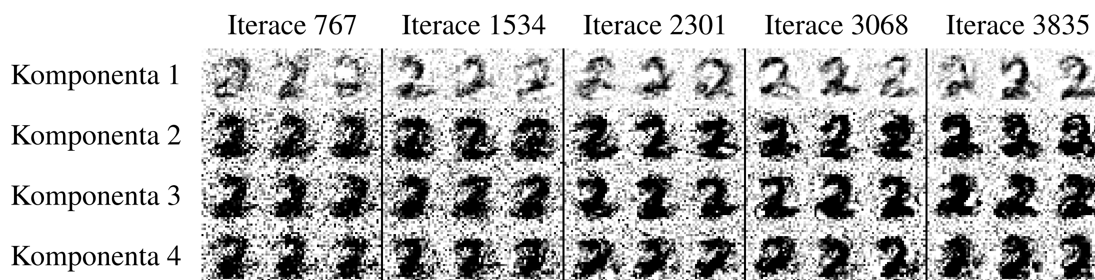
Tabulka 3.1: Časy a urychlení pro 5 iterací algoritmu ADAM s dvěma vektory pozorování pro SPTN o třech součtově-transformačních vrstvách.



Obrázek 3.1: Časy pro 5 iterací algoritmu ADAM s dvěma vektory pozorování pro SPTN o třech součtově-transformačních vrstvách.



Obrázek 3.2: Urychlení pro 5 iterací algoritmu ADAM s dvěma vektory pozorování pro SPTN o třech součtově-transformačních vrstvách.



Obrázek 3.3: Náhodné vzorky vygenerované z náhodného rozdělení reprezentující číslici 2 z datasetu MNIST pro 4 nejpravděpodobnější komponenty v různých fázích učení.

3.2 Samplování

Jelikož SPTN implementuje pravděpodobnostní rozdělení, lze z něho generovat náhodné vzorky. Pro demonstraci byl využit dataset MNIST [13], který obsahuje ručně psané číslice, konkrétně byla vybrána číslice 2. Konfigurace sítě byla dvě součtově-transformační vrstvy zakončené listovou vrstvou. Obě součtově-transformační vrstvy obsahovaly 4 transformační uzly a nesdílely parametry součtových uzlů. Pro učení byl použit algoritmus ADAM, kde v každé iteraci výpočet probíhal s osmi vektory pozorování. Těchto iterací bylo provedeno celkem 3835, každý vzorek byl pro učení použit průměrně pětkrát. Během učení byl vždy uložen průběžný stav po 767 iteracích. Po skončení učení byly vybrány 4 komponenty sítě s největší pravděpodobností, a z těchto komponent a každého uloženého stavu byly vygenerovány tři náhodné vzorky, vyobrazené na obrázku 3.3. Jednotlivé hodnoty ve vygenerovaných vzorcích byly následně omezeny na interval $[0, 1]$.

Závěr

Tato práce se zabývá sítěmi typu Sum-Product-Transform a jejich implementací na architektuře typu GPU. Náplní kapitoly 1 je obecný popis sítí typu SPTN, regularizace těchto sítí pro implementaci na GPU a odvození matematické formulace učení sítě. Regularizace sítě se skládá ze čtyř kroků. Prvním je omezení typu pravděpodobnostních distribucí v listových uzlech na standardní normální rozdělení. Druhým krokem je omezení typu transformací v transformačních uzlech na afinní transformace, které mají matici reprezentující lineární část transformace uloženou v SVD rozkladu, což umožňuje efektivní inverzi transformace a výpočet determinantu jejího jakobiánu a derivací vůči němu. Za těchto omezení existují struktury sítě se stejnou schopností reprezentace distribuce pravděpodobnosti, ale rozdílnou výpočetní náročností. Třetí krok tedy omezuje struktury sítě tak, aby ze všech sítí se stejnou schopností reprezentace distribuce pravděpodobnosti mohla být použita pouze ta s nejmenší výpočetní náročností. Čtvrtý krok popisuje modifikaci transformace v transformačních uzlech, které mají za potomka listový uzel. Tato modifikace umožňuje snížit výpočetní náročnost bez ztráty schopnosti reprezentace pravděpodobnostní distribuce. Pro učení sítě je použita zpětná propagace.

Kapitola 2 obsahuje stručný úvod do architektury GPU, CUDA a programovacího jazyka Julia. Dále obsahuje popis implementace SPTN na GPU, která se skládá z algoritmů pro inferenci, učení, samplování a marginalizaci. Pro reprezentaci ortogonálních matic v SVD rozkladu matic reprezentujících transformace v transformačních uzlech byl zvolen Butterfly rozklad [16], který umožňuje paralelizaci výpočtu násobení touto maticí a výpočtu derivací vůči parametrům.

Obsahem kapitoly 3 je porovnání rychlosti učení sítě pomocí čistě sekvenční implementace oproti implementaci pro architekturu GPU, a demonstrace generování náhodných vzorků z již naučené sítě. Bylo dosaženo urychlení od jednoho do dvou řádů, v závislosti na velikosti sítě. Pro demonstraci samplování byla zvolena číslice 2 z datasetu MNIST [13], pro niž byla natrénována pravděpodobnostní distribuce, a následně z ní byly vygenerovány vzorky.

Na tuto práci by bylo možné navázat například rozšířením implementace pro sítě s méně restrikcemi, případně přidáním podpory pro využití více než jedné GPU.

Bibliografie

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018, s. 497. ISBN: 978-3-319-94462-3.

[2] Mathieu Besançon et al. 'Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem'. In: *arXiv e-prints*, arXiv:1907.08611 (čvc. 2019), arXiv:1907.08611. arXiv: 1907.08611 [stat.CO].

[3] Tim Besard, Christophe Foket a Bjorn De Sutter. 'Effective Extensible Programming: Unleashing Julia on GPUs'. In: *IEEE Transactions on Parallel and Distributed Systems* (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[4] Tim Besard et al. 'Rapid software prototyping for heterogeneous and distributed platforms'. In: *Advances in Engineering Software* 132 (2019), s. 29–46.

[5] *ChainRules.jl package readme*. 2021. URL: https://github.com/JuliaDiff/ChainRules.jl (cit. 03. 04. 2021).

[6] NVIDIA Corporation. *CUDA Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (cit. 06. 08. 2020).

[7] Gene H. Golub a Charles F. Van Loan. *Matrix computations*. The John Hopkins University Press, 2013.

[8] Michael Innes et al. 'Fashionable Modelling with Flux'. In: *CoRR* abs/1811.01457 (2018). arXiv: 1811.01457. URL: https://arxiv.org/abs/1811.01457.

[9] Mike Innes. 'Flux: Elegant Machine Learning with Julia'. In: *Journal of Open Source Software* (2018). DOI: 10.21105/joss.00602.

[10] Ty McKercher John Cheng Max Grossman. *Professional CUDA C programming*. John Wiley & Sons, Inc., 2014.

[11] *Julia 1.5 Documentation*. 2020. URL: https://docs.julialang.org/en/v1/ (cit. 12. 08. 2020).

[12] Diederik P. Kingma a Jimmy Ba. 'Adam: A Method for Stochastic Optimization'. In: *arXiv e-prints*, arXiv:1412.6980 (pros. 2014), arXiv:1412.6980. arXiv: 1412.6980 [cs.LG].

[13] Yann LeCun a Corinna Cortes. 'MNIST handwritten digit database'. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[14] Dahua Lin et al. *JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions*. Čvc. 2019. DOI: 10.5281/zenodo.2647458. URL: https://doi.org/10.5281/zenodo.2647458.

[15] Tomas Pevny et al. 'Sum-Product-Transform Networks: Exploiting Symmetries using Invertible Transformations'. In: *arXiv e-prints*, arXiv:2005.01297 (květ. 2020), arXiv:2005.01297. arXiv: 2005.01297 [stat.ML].

[16] John Polcari. ‘Butterfly Decompositions for Arbitrary Unitary Matrices - Rev 2’. In: (ún. 2014).