VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PARALELNÍ TRÉNOVÁNÍ HLUBOKÝCH NEURONOVÝCH SÍTÍ

PARALLEL DEEP LEARNING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ONDŘEJ ŠLAMPA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MICHAL HRADIŠ, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Šlampa Ondřej, Bc.**
Obor: Inteligentní systémy
Téma: **Paralelní trénování hlubokých neuronových sítí**
Parallel Deep Learning
Kategorie: Zpracování obrazu

Pokyny:

1. Prostudujte základy teorie trénování neuronových sítí a paralelního zpracování na počítačových klastrech.
2. Vytvořte si přehled o současných technologiích vhodných pro klastrové trénování neuronových sítí a zpracování obrazu.
3. Vyberte konkrétní technologii a navrhněte její aplikaci na úlohu trénování neuronových sítí a zpracování obrazu.
4. Implementujte navrženou metodu s využitím vhodných nástrojů a proveďte experimenty výkonnosti systému.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte stručný plakát nebo video prezentující vaši práci, její cíle a výsledky.

Literatura:

- Moritz, Philipp, et al. "SparkNet: Training Deep Networks in Spark." arXiv preprint arXiv:1511.06051 (2015).
- Caffe on Spark by Yahoo, https://github.com/yahoo/CaffeOnSpark
- Krizhevsky, A., Sutskever, I. and Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

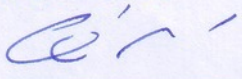
Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hradiš Michal, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2



doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cílem této práce je navrhnou způsob jak zhodnotit výhodnost použití paralelního trénování neuronových sítí. V této práci jsem provedl analýzu paralelního trénování se zaměřením na délku trénování. Vycházím ze sekvenční délky trénování a délky přenosu vah po síti. Výsledkem této práce je návrh vzorců, které slouží k odhadu zrychlení na více výpočetních jednotkách. Tyto vzorce je možné použít na zjištění ideálního počtu pracovních jednotek pro trénování.

Abstract

Aim of this thesis is to propose how to evaluate favourableness of parallel deep learning. In this thesis I analyze parallel deep learning and I focus on its length. I take into account gradient computation length and weight transportation length. Result of this thesis is proposal of equations, which can estimate the speedup on multiple workers. These equations can be used to determine ideal number of workers for training.

Klíčová slova

Neuronové sítě, konvoluční neuronové sítě, trénování, soft computing, odhad délky výpočtu, distribuovaný výpočet, paralení výpočet, počítačové sítě, Tensorflow, Python.

Keywords

Neural networks, convolutional neural networks, training, soft computing, computation length estimation, distributed computing, parallel computing, computer networks, Tensorflow, Python.

Citace

ŠLAMPA, Ondřej. *Paralelní trénování hlubokých neuronových sítí.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hradiš Michal.

Paralelní trénování hlubokých neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Michala Hradiše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Šlampa
23. května 2017

Poděkování

Chtěl bych poděkovat panu Michalu Hradišovi za přínosné a pravidelné konzultace, odborné rady a bezproblémovou komunikaci.

Obsah

Kapitola 1

Úvod

Přestože počátky neuronových sítí sahají až do padesátých let 20. století, v posledních několika letech se jim dostává velmi velké pozornosti. Neuronové sítě jsou výpočetním modelem, který je používaný ve strojovém učení. Základní myšlenkou neuronových sítí je propojení velkého množství malých jednotek do velké struktury. Neuronové sítě jsou použitelné v mnoha odvětvích výpočetních technologií, např. rozpoznávání hlasu, detekce objektů na obrázku či videu nebo klasifikace.

Jednou z nevýhod neuronových sítí je jejich trénování, které může trvat velmi dlouhou dobu v řádu dnů či týdnů. Trénování hlubokých neuronových sítí je obvykle velmi výpočetně náročné. Jedním ze způsobů jak snížit časovou náročnost výpočtu jakéhokoliv druhu je použití distribuovaného výpočtu. Novým problémem při distribuovaném výpočtu je dodatečná komunikace mezi výpočetními jednotkami. Neuronové sítě můžou obsahovat miliony parametrů, jejich přenosy tak mají nezanedbatelný vliv na délku výpočtu.

Cílem této práce je vytvořit způsob, jak ověřit výhodnost distribuovaného trénování pro zadanou neuronovou síť. V této práci provedu analýzu trénování sítě vzhledem k délce natrénování jedné dávky a délce přenosu vah sítě. Z těchto dvou hodnot můžu vypočítat odhad délky celého trénování a zrychlení dosaženého pomocí distribuovaného výpočtu.

První kapitola se po myšlenkové stránce dá rozdělit na dvě části. První tři sekce tvoří první část kapitoly, která se zabývá teoretickým popisem a složením neuronových sítí. Zde se nacházejí definice základních pojmů z oblasti neuronových sítí. Na to navazují poslední čtyři sekce, které popisují konkrétní návrhy neuronových sítí. Tyto sítě se nazývají VGG16E, GoogLeNet, SqueezeNet a Resnet34.

Velmi důležitou oblastí z teorie neuronových sítí je jejich trénování, které popisuje druhá kapitola. Začátek kapitoly slouží k popisu samotných trénovacích algoritmů. Druhá sekce pak popisuje výpočet gradientu, který se používá v algoritmech na začátku kapitoly. Konec této kapitoly je věnován algoritmům pro trénování na více výpočetních jednotkách.

Třetí kapitola popisuje nejdůležitější část této práce. Tou jsou odhady doby paralelního trénování neuronových sítí podle použité techniky trénování. Paralelní trénování je buď synchronní nebo asynchronní, odhady dob trénování jsou ve dvou sekcích této kapitoly. Výpočtu zrychlení trénování sítě je věnována poslední část této kapitoly.

Následující kapitola se zabývá implementačními prostředky, které jsem použil, abych ověřil nebo vyvrátil správnost výpočtů navržených v předchozí kapitole. Pro trénování jsem použil knihovnu Tensorflow, proto zde popisuji její základní principy. Kapitola také obsahuje popis prostření, ve kterém jsem provedl experimenty.

V následující kapitole jsem popsal provedené experimenty a jejich výsledky. Popis samotných experimentů je pochopitelně první částí této kapitoly. Následující tři sekce této

kapitoly se věnují třem způsobům trénování, které jsem popsal v předchozí sekci. V každé sekci je několik grafů, které zobrazují závislost zrychlení trénování na počtu pracovních jednotek pro danou neuronovou síť a způsob distribuovaného trénování. Nové informace odvozené z každého grafu jsem popsal a zhodnotil.

Poslední částí této práce je shrnutí výsledků a poznatků, které jsem zjistil při jejím vypracování. Tato kapitola také obsahuje návrh mechanizmu pro lepší rozdělení komunikace při trénování.

Tato práce navazuje na semestrální projekt. Ze semestrálního projektu jsem převzal kapitolu o neuronových sítích, popis sítě VGG16E a trénovací množiny Caltech256. Text převzatý ze semestrálního projektu je v kapitolách 2, 3 a 5.

Kapitola 2

Neuronové sítě

Práce se zabývá neuronovými sítěmi, proto je nutné vysvětlit principy, na kterých pracují. K tomu slouží tato kapitola. První dvě sekce se zabývají dvěma nejrozšířenějšími typy neuronových sítí: dopřednými plně propojenými a konvolučními. Třetí sekce je věnována těm částem neuronových sítí, které nepoužívají žádné neurony. Poslední čtyři sekce popisují čtyři různé neuronové sítě, se kterými jsem experimentoval v této práci.

Počátky neuronových sítí spadají až do roku 1957, kdy Frank Rosenblatt [20] vytvořil první neuronovou síť – perceptron. Neuronové sítě jsou inspirovány neurony živých organizmů. Historií a teorií se zabývají Ian Goodfellow, Yoshua Bengio a Aaron Courville v knize Deep Learning [6]. Zpočátku měly neuronové sítě poměrně omezené možnosti. To se změnilo v 90. letech, kdy počítače byly dostatečně výkonné na to, aby zvládly natrénovat několikavrstvé dopředné sítě pomocí algoritmu back-propagation. Neuronové sítě se skládají z malých jednotek, které se nazývají neurony. Existuje několik způsobů, jak neurony propojit do struktury a vytvořit z nich tak neuronovou síť. Dva nejčastější způsoby jsou plně propojená síť a konvoluční neuronová síť, které jsou popsané v následujících dvou podsekcích

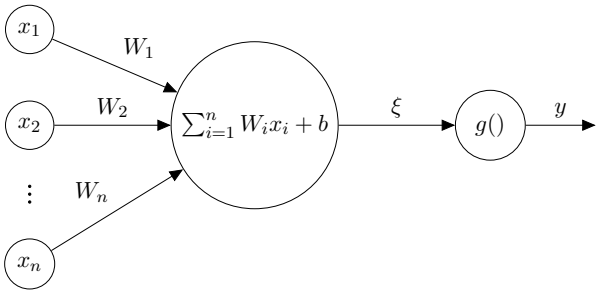
2.1 Dopředné neuronové sítě

Dopředná neuronová síť je tvořena z vrstev neuronů, které jsou na sebe napojeny – výstup jedné vrstvy je napojen na vstup druhé vrstvy. Pokud jsou propojeny všechny výstupy každé vrstvy na všechny vstupy následující vrstvy, pak se taková síť nazývá plně propojená. Je to nejjednodušší architektura neuronové sítě. První vrstva se nazývá vstupní a neprovádí žádný výpočet, pouze přeposílá vstup na následující vrstvu. Poslední vrstva se nazývá výstupní. Ostatní vrstvy se nazývají skryté vrstvy. Diagram 2.4 ukazuje dopřednou neuronovou síť se čtyřmi neurony ve vstupní vrstvě, pěti neurony v jedné skryté vrstvě a třemi neurony ve výstupní vrstvě.

Lineární jednotka. Lineární jednotka je typ neuronu, který provede sumu součinů vstupů neuronu a k nim odpovídajících vah, a který k této sumě přičte bias. Činnost jednotky je popsaná následujícím způsobem

$$\xi = W \cdot X + b, \tag{2.1}$$

kde ξ je výstup, W je vektor vah, X je vstup a b je bias. Nevýhodou lineární jednotky je, že provádí pouze lineární transformace nad vstupem a pro popis složitějších funkcí musí



Obrázek 2.1: Lineární jednotka včetně aktivační funkce $g()$.

být zkombinována s nelineární aktivační funkcí. Diagram 2.1 zobrazuje lineární jednotku s aktivační funkcí $g()$.

Vrstva lineárních jednotek. Z praktického hlediska je velkou výhodou lineární jednotky to, že je možné činnost jedné vrstvy plně propojené neuronové sítě definovat jako maticové násobení mezi vektorem vstupu a transponovanou maticí vah vrstvy. První řádek matice vah vrstvy obsahuje váhy prvního neurony vrstvy atd.. Přičtení biasu odpovídá součtu výsledku maticového násobení a vektoru biasů. Aktivační funkce je pak aplikována na každý prvek výsledku součtu. Činnost takové vrstvy je popsaná jako

$$\Xi = W^T \cdot X + B, \tag{2.2}$$

kde Ξ je výstup, W je matice vah, X je vstup a B je bias.

2.1.1 Aktivační funkce

Neuronová síť vytvořená kombinováním lineárních funkcí je také lineární funkce. To znamená, že taková neuronová síť je nepoužitelná pro problémy, které nejsou lineárně separabilní. Tento problém se řeší použitím nelineární aktivační funkce na výstup každého neuronu. Jelikož taková síť obsahuje nelineární funkce, znamená to, že tato síť je také nelineární funkcí, a tudíž může být použita pro problémy, které jsou nelineárně separabilní. Jako aktivační funkce se obvykle používají různé matematické funkce.

Nejoblíbenější funkcí je usměrněná lineární funkce (rectified linear unit, ReLU) popsaná následujícím způsobem

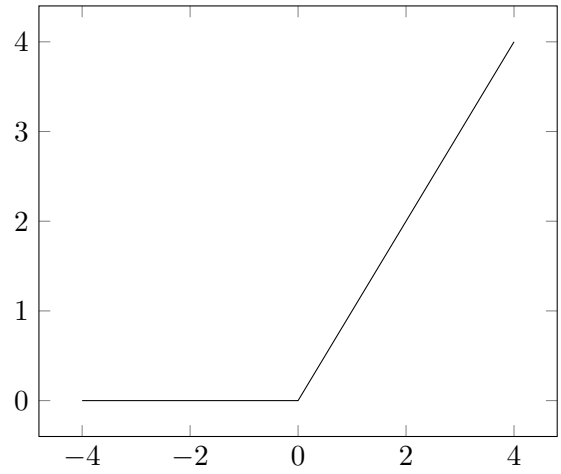
$$y = \begin{cases} x & \text{pokud } x > 0 \\ 0 & \text{pokud } x \leq 0 \end{cases}. \tag{2.3}$$

Grafické znázornění této funkce je na obrázku 2.2. Výhodou této funkce je její nízká výpočetní náročnost. To platí i pro její derivaci. Nevýhodou je, že neexistuje její derivace v bodě 0, a že výstupní hodnoty pro $x > 0$ nejsou nijak omezeny.

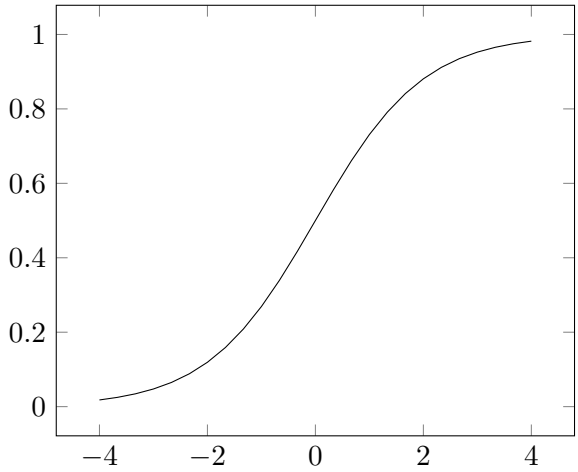
Další oblíbenou aktivační funkcí je logistická funkce, která je definována následující rovnicí

$$y = \frac{1}{1 + e_{-x}}. \tag{2.4}$$

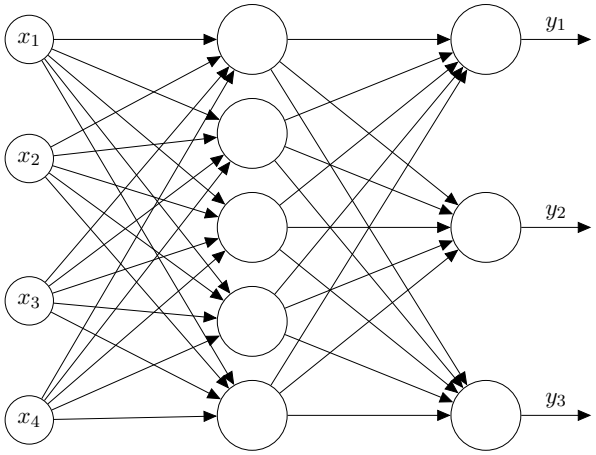
Toto je jedna z verzí logistické funkce, která se nazývá sigmoida. Tato funkce je graficky znázorněna na obrázku 2.3. Výhodou této funkce je, že výstupní hodnoty jsou v omezeném intervalu $y \in (-1, 1)$, a že její derivace je definována ve všech bodech. Nevýhodou je její výpočetní náročnost ve srovnání s ReLU.



Obrázek 2.2: Graf usměrněné lineární aktivační funkce.



Obrázek 2.3: Graf aktivační logistické funkce sigmoida.



Obrázek 2.4: Dopředná plně propojená neuronová síť. Pro zjednodušení diagramu nejsou zakresleny aktivační funkce neuronů.

2.2 Konvoluční neuronové sítě

Konvoluční neuronové sítě (convolutional neural networks, CNN) jsou druhem neuronové sítě, ve které jsou neurony propojeny takovým způsobem, že jedna vrstva sítě počítá konvoluce mezi vstupem do vrstvy a váhami sítě. První konvoluční neuronovou síť představil v roce 1989 Yann LeCun [16], který ji použil k rozpoznávání číslic. V roce 2012 vyhrál Alexandr Krizhevsky soutěž ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [15] s konvoluční neuronovou sítí, to pomohlo popularizaci tohoto druhu neuronové sítě. Tato metoda byla výrazně úspěšnější než její konkurence. Další výhodou konvolučních sítí je velká dostupnost datových sad a grafických karet, které dokážou zpracovat velké množství obrázků za sekundu. Konvoluční neuronová síť se skládá ze dvou typů vrstev: konvoluční vrstvy a podvzorkovací vrstvy.

2.2.1 Konvoluce

Konvoluce je matematická operace nad dvěma funkcemi s argumentem z oblasti reálných čísel, která se značí symbolem $*$. Rovnice

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{2.5}$$

zobrazuje konvoluci nad dvěma funkcemi x a w. Jejím výsledkem je funkce s.

Problémem předchozí definice je to, že pracuje nad spojitými funkcemi. Pro praktické použití je vhodnější konvoluce, která pracuje nad diskrétními funkcemi. Ta je definována jako

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(t-a)h(a). \tag{2.6}$$

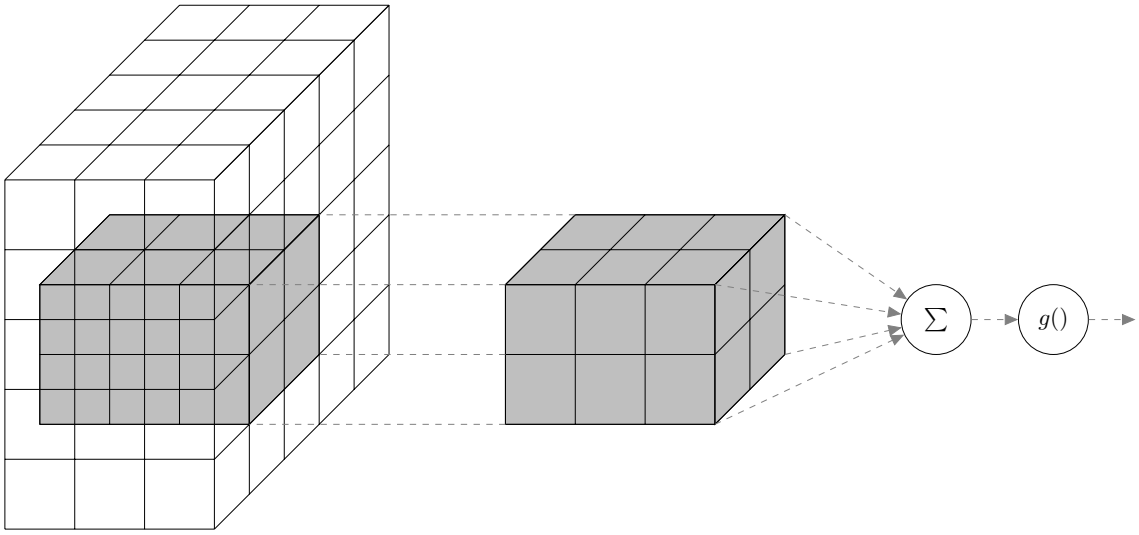
V konvolučních neuronových sítích se obvykle provádí konvoluce mezi dvěma několikarozměrnými poli dat. Jedním z těchto dvou polí je vstup do konvoluční vrstvy a druhým je část trénovatelných vah této vrstvy. Těmto několikarozměrným polím se říká tenzory. Jelikož oba tenzory musí být možné uložit, předpokládá se, že každá funkce se skládá z konečného počtu nenulových bodů, které je možné si uložit, a že funkce je nulová ve všech ostatních bodech. V praxi to znamená, že je možné možné implementovat konvoluci nad poli s konečných počtem prvků. Vzorec

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.7}$$

ukazuje diskrétní konvoluci nad dvěma dvourozměrnými poli I a K.

2.2.2 Konvoluční vrstva

Důležitou částí CNN jsou konvoluční vrstvy. Každý neuron této vrstvy je propojený pouze s několika neurony předchozí vrstvy nebo s body vstupního obrazu, pokud je tato vrstva první. Tyto vstupní neurony nebo body tvoří okolí jednoho neuronu nebo bodu. Neurony konvoluční vrstvy jsou organizovány do rovin. Všechny roviny jsou tvořeny stejným počtem neuronů. Všechny neurony ve stejné rovině mají stejnou množinu vah. To je založeno na úvaze, že všechny neurony v rovině slouží k detekci stejného příznaku. Každý neuron provádí skalární součin mezi jeho vstupem a množinou vah roviny, ve které se nachází, je to tedy lineární jednotka. Výstup všech neuronů v jedné rovině se nazývá mapa příznaků. Jedna



Obrázek 2.5: Jeden neuron konvoluční vrstvy. V levé části diagramu je umístěn vstup do konvoluční vrstvy, které je tento neuron součástí. Šedou barvou je zvýrazněna ta část vstupu, se kterou je neuron propojený. Střední část diagramu obsahuje třírozměrný tensor sdílených vah neuronu/roviny. V pravé části je zobrazeno tělo neuronu označené symbolem $\sum$ a aktivační funkce $g()$.

rovina neuronů tak vlastně provede konvoluci mezi vstupem konvoluční vrstvy a množinou vah jejích neuronů. Výstupy všech rovin neuronů tvoří výstup konvoluční vrstvy, který se označuje jako mapy příznaků. Jeden neuron konvoluční vrstvy zobrazuje diagram 2.5.

Neurony konvoluční vrstvy slouží k detekci příznaků (hrany, rohy, změna barvy atd.), které jsou pak uloženy v mapách příznaků. Následující vrstvy pak kombinují jednotlivé příznaky z několika map na nové příznaky o vyšší úrovni.

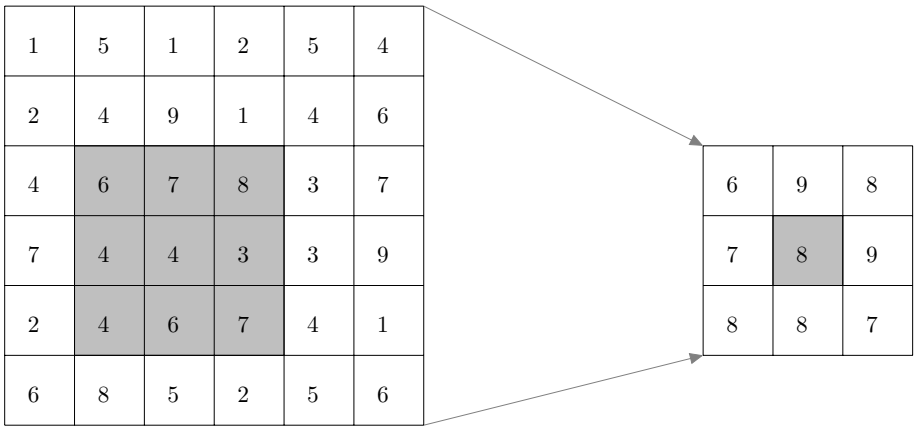
2.3 Vrstvy neobsahující neurony

Důležitou částí neuronových sítí jsou i vrstvy, které nepoužívají vůbec žádné neurony. Tyto vrstvy slouží k úpravě dat, které si mezi sebou předávají dvě jiné vrstvy.

2.3.1 Vrstva maxpooling

Podvzorkování slouží ke snížení velikosti map příznaků, se kterými se v konvolučních neuronových sítích pracuje. Jednou z myšlenek za podvzorkováním je to, že není důležitá přesná poloha příznaku, ale že stačí pouze jeho přibližná poloha vzhledem k ostatním příznakům. Druhou myšlenkou je snížení časové a paměťové náročnosti sítě. To je důležité, protože v sítích dochází ke zvyšování počtu map příznaků. Snížení jejich velikosti by tak mělo omezit narůstání výpočetní složitosti v pozdějších částech sítě. Další výhodou podvzorkování je kontrola přetrénování.

Maxpooling je jedna z možností jak provést podvzorkování. Princip této vrstvy spočívá v tom, že se čtvercové okolí bodu nahradí největší hodnotou z tohoto okolí (včetně bodu samotného). Krok maxpoolingu říká o kolik bodů se musí posunout okolí pro získání dalšího bodu výsledku. Pokud je krok roven jedné, velikost výstupu se rovná velikosti vstupu. Vyšší velikost kroku způsobí snížení velikosti výstupu.



Obrázek 2.6: Ukázka činnosti vrstvy maxpooling používající okolí bodu o velikosti 3×3 a s krokem 2. Mapa příznaků nalevo je vstup do vrstvy a vpravo je výstup z vrstvy. Šedě je vyznačeno okolí jednoho bodu a k němu odpovídající maximum.

2.3.2 Vrstva avgpooling

Tato vrstva byla navržena týmem z univerzity Singapuru v práci [17]. Konvoluční neuronové sítě jsou často ukončeny vektorizací map příznaků, plně propojenou vrstvou a vrstvou softmax. Tento styl architektury je spojením konvoluční a plně propojené neuronové sítě. Konvoluční část sítě tak slouží k extrakci rysů a klasifikace se pak provádí pomocí plně propojené vrstvy. Plně propojené vrstvy jsou ale náchylné k přetrénování.

Jedním způsobem jak tento problém vyřešit je vrstva avgpooling. Předpokládá se, že výstup poslední konvoluční vrstvy má stejný počet map příznaků jako je tříd na výstupu sítě. Následující vrstva avgpool vypočítá průměry všech map příznaků a jejím výstupem tak bude vektor, který má délku rovnající se počtu tříd. Síť bude ukončena vrstvou typu softmax. Tento přístup vynucuje korespondenci mezi mapami příznaků a třídami a používá pouze nástrojů konvolučních neuronových vrstev. Druhou výhodou je, že vrstva avgpool neobsahuje žádné parametry na rozdíl od plně propojené vrstvy. Tím se odstraní nebezpečí přetrénování této vrstvy a zároveň se snižuje počet parametrů celé sítě, které je nutné přenést během paralelního trénování.

2.3.3 Normalizace lokální odezvy

Součástí konvolučních neuronových sítí jsou také normalizační vrstvy, které slouží k úpravě map příznaků, tak aby se ostatním vrstvám s nimi lépe pracovalo. To znamená, že jsou z pohledu paralelního trénování velmi užitečné, protože obvykle neobsahují váhy, které je nutné přenášet.

Normalizace lokální odezvy (Local Response Normalization, LRN) byla vytvořena týmem z univerzity v Torontu a definována v práci [15]. Inspirací bylo chování biologických neuronů, kde excitace jednoho neuronu snižuje aktivitu okolních neuronů. Cílem této normalizace je podpoření soupeření neuronů na stejných souřadnicích v sousedících rovinách ve stejné vrstvě sítě. Rovnice

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^j_{x,y}{}^2) \right)^{\beta} \tag{2.8}$$

udává, jak vypočítat normalizaci pro každý výstup konvoluční vrstvy, kde $a^i_{x,y}$ je výstup neuronu $i$té roviny na souřadnicích (x, y) a $b^i_{x,y}$ je jemu odpovídající výstup z LRN. N je celkový počet rovin vrstvy, n je počet sousedních vrstev, které mají vliv na výstup LRN a α, β a k jsou hyperparametry.

2.3.4 Normalizace dávek

Normalizaci dávek (Batch Normalization, BN) navrhli Sergey Ioffe a Christian Szegedy v práci [13]. Problémem při trénování sítě jsou změny rozložení aktivací neuronů. Toto se obvykle řeší pomocí malého koeficientu učení a velkého počtu kroků trénování. Normalizace dávek omezuje vliv tohoto jevu a umožňuje použít vyšší koeficient učení, který zrychlí trénování. Tato normalizace se provádí pomocí statistických dat jedné dávky při trénování. Vstupem do normalizace je dávka $\mathcal{B} = \{x_1, ..., x_m\}$ a výstupem je normalizovaná dávka $\mathcal{B}_{BN} = \{y_i = BN_{\gamma,\beta}(x_i)\}$. Rovnice 2.9 a 2.10 slouží k výpočtu průměru a směrodatné odchylky dávky. Samotná normalizace je popsána rovnicí 2.11. Jelikož tato normalizace může změnit význam vrstev, tak se k normalizaci ještě přidá škálování a posun hodnot popsané v rovnici 2.12.

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.9}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{2.10}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{2.11}$$

$$y_i = \gamma\hat{x}_i + \beta = BN_{\gamma,\beta}(x_i) \tag{2.12}$$

2.3.5 Funkce softmax

Výstupem neuronové sítě je často zařazení vstupu do jedné z n tříd. Toto zařazení je implementováno pomocí vektoru délky n, kde každý prvek vektoru vyjadřuje příslušnost vstupu k dané třídě. Je možné sestavit neuronovou síť, která na výstupu dá takový vektor. Problém je, že hodnoty ve výstupu se musí chápat v kontextu ostatních hodnot výstupu.

Pro lepší práci s výstupem sítě je tak vhodné provést jeho normalizaci pomocí funkce softmax. Funkce softmax je generalizací logistické funkce pro větší počet tříd. Tato normalizace zaručí, že každý prvek výstupu je v intervalu (0,1) a součet všech prvků výstupu je 1. Funkce softmax je definována nad $j$tým prvkem vektoru z jako

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}. \tag{2.13}$$

2.4 VGG16E

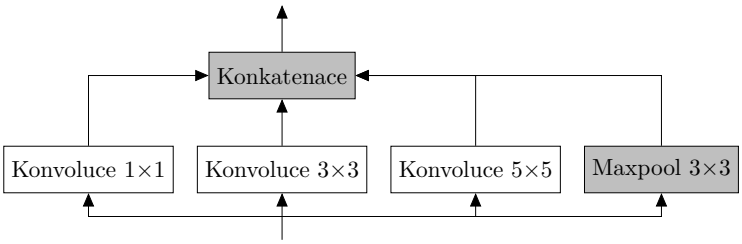
Síť VGG16E (Visual Geometry Group) navrhli Karen Simonyan a Andrew Zisserman v práci [23]. Vstupem sítě je RGB obrázek o velikosti 224×224 pixelů. Obrázek je zpracován šestnácti konvolučními vrstvami, které používají filtry s velmi malou velikostí 3×3. Toto je nejmenší filtr, který dokáže zachytit dojem vpravo, vlevo, dole, nahoře nebo ve středu.

Typ	Velikost vah	Velikost výstupu
konvoluční	3×3×3×64	128×128×64
konvoluční	3×3×64×64	128×128×64
maxpooling		64×64×64
konvoluční	3×3×64×128	64×64×128
konvoluční	3×3×128×128	64×64×128
maxpooling		32×32×128
konvoluční	3×3×128×256	32×32×256
konvoluční	3×3×256×256	32×32×256
konvoluční	3×3×256×256	32×32×256
konvoluční	3×3×256×256	32×32×256
maxpooling		16×16×256
konvoluční	3×3×256×512	16×16×512
konvoluční	3×3×512×512	16×16×512
konvoluční	3×3×512×512	16×16×512
konvoluční	3×3×512×512	16×16×512
maxpooling		8×8×512
konvoluční	3×3×512×512	8×8×512
konvoluční	3×3×512×512	8×8×512
konvoluční	3×3×512×512	8×8×512
konvoluční	3×3×512×512	8×8×512
maxpooling		4×4×512
plně propojená	8192×4096	4096
plně propojená	4096×4096	4096
plně propojená	4096×1000	1000
softmax		

Tabulka 2.1: Struktura sítě VGG16E, která je popsaná v sekci 2.4.

Krok konvoluce je jeden pixel. Padding vstupu je takový, aby výsledek konvoluce měl stejnou velikost jako vstup. Pro filtr o velikosti 3×3 to je jeden pixel. Prostorové podvzorkování je provedeno pomocí pěti vrstev typu maxpooling, které následují některé konvoluční vrstvy (ne všechny konvoluční vrstvy jsou následovány vrstvou maxpooling). Maxpooling používá okno o velikosti 2×2 s krokem 2 pixely. Série konvolučních a maxpooling vrstev je následována třemi plně propojenými vrstvami. První dvě mají 4096 kanálů, třetí provádí klasifikaci do 1000 tříd a tudíž má 1000 kanálů, jeden pro každou třídu. Poslední vrstvou je softmax vrstva. Všechny skryté vrstvy obsahují ReLU nelinearitu. Podrobná architektura sítě je popsána v tabulce 2.1 a diagramu D.1.

Tato konvoluční síť se výrazně liší od ostatních konvolučních sítí. Spíše než relativně větší filtry v počáteční konvolučních vrstvách např. 11×11 s krokem 4 [15] nebo 7×7 s krokem 2 [22] používá tato síť velmi malé 3×3 filtry, které se konvolvují s každým pixelem vstupu (s krokem 1). Je zřejmé, že tři po sobě jdoucí konvoluční vrstvy (tzn není mezi nimi podvzorkování) s 3×3 filtrem mají stejný efektivní dosah jako filtr o velikosti 7×7. Výhodou tohoto přístupu je použití více ReLU nelinearit místo jedné, což způsobí, že rozhodovací funkce bude více diskriminační. Druhou výhodou je snížení počtu parametrů. Předpokládejme, že vstup i výstup 3×3 filtru má C kanálů. Tři po sobě jdoucí konvoluční vrstvy mají $3 \cdot 3^2C^2 = 27C^2$ parametrů. Obdobně jedna vrstva s 7×7 filtrem má $7^2C^2 = 49C^2$ parametrů. Počet parametrů se zvýšil o 81%.



Obrázek 2.7: Diagram naivní verze modulu Inception, který je popsán v podsekci 2.5.1.

2.5 GoogLeNet

Síť GoogLeNet navrhl tým odborníků z univerzit Michiganu a Severní Karolíny a z firem Google a Magic Leap v práci [25]. Nejjednodušším způsobem jak zvýšit výkon sítě je zvětšit její velikost. To je možné provést pomocí zvětšení buď hloubky sítě - počtu vrstev sítě nebo šířky sítě - počtu neuronů v každé vrstvě. Toto je jednoduchý a bezpečný způsob jak vytvořit kvalitnější modely, který má bohužel dvě velké nevýhody. První je zvýšení počtu parametrů, které zvyšuje šanci přetrénování a vyžaduje větší data set pro natrénování. Druhou je velmi výrazné zvýšení výpočetních nároků. Pokud nejsou výpočetní kapacity efektivně využity (například většina vah je blízká nule), pak je zbytečně ztrácen výpočetní výkon. Výpočetní výkon je vždy omezený. Základním způsobem jak toto vyřešit je použití řídce propojených vrstev. V praxi toto řešení naráží na špatnou optimalizaci výpočtů ve srovnání s hustě propojenými vrstvami.

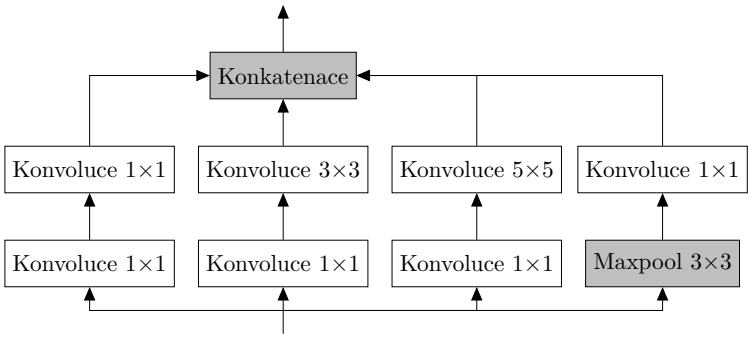
2.5.1 Modul Inception

Architektura Inception byla vytvořena jako výsledek hypotetického algoritmu pro tvorbu konvolučních neuronových sítí, který se snaží vytvořit síť s řídce propojenými vrstvami pomocí plně propojených vrstev. Hlavní myšlenkou architektury Inception je, jak je možné vytvořit optimální lokální řídkou strukturu konvoluční sítě pomocí dostupných hustých komponent. Při tvorbě řídkých konvolučních sítí se vytvářejí shluky neuronů podle korelačních statistik. To je možné provést pro každou vrstvu. Za předpokladu, že existuje vysoká korelace mezi sousedícími neurony, je možné nahradit shluky neuronů konvolucemi. Z praktických důvodů byly použity konvoluce s filtry o velikosti 1×1, 3×3 a 5×5. Dodatečně se ještě provede provede maxpooling nad vstupem modulu Inception. Výsledná architektura je pak kombinací těchto vrstev, proto je jejich výstup konkatenován. Obrázek 2.7 zobrazuje tuto verzi modulu Inception.

Problémem této architektury je vysoká výpočetní náročnost konvolucí s většími filtry. Řešením tohoto problému je snížení dimenzionality před těmito konvolucemi. Redukce dimenzionality je provedena pomocí konvolucí s filtrem 1×1, které snižují počet map příznaků. Výhodou tohoto řešení je zavedení dodatečných ReLU nelinearit po každé redukující konvoluci. Tato verze modulu Inception je zobrazena na obrázku 2.8. Pro snížení paměťové náročnosti síť začíná konvolucemi a maxpoolingem a moduly typu Inception jsou použity až v následujících vrstvách.

2.5.2 Popis sítě

GoogLeNet je konvoluční neuronová síť využívající Inception architekturu. Vstupem sítě je RGB obrázek o velikosti 224×224 pixelů. Podrobná architektura sítě je popsána v ta-



Obrázek 2.8: Diagram modulu Inception s redukcí dimenzionality, který je popsán v podsekci 2.5.1.

Typ	Velikost filtru/krok	Velikost výstupu	1×1	redukce před 3×3	3×3	redukce před 5×5	5×5	redukce před maxpool
konvoluce	7×7/2	112×112×64						
maxpool	3×3/2	56×56×64						
konvoluce	3×3/1	56×56×192		64	192			
maxpool	3×3/2	28×28×192						
inception 3a		28×28×256	64	96	128	16	32	32
inception 3b		28×28×480	128	128	192	32	96	64
maxpool	3×3/2	14×14×480						
inception 4a		14×14×512	192	96	208	16	48	64
inception 4b		14×14×512	160	112	224	24	64	64
inception 4c		14×14×512	128	128	256	24	64	64
inception 4d		14×14×528	112	144	288	32	64	64
inception 4e		14×14×832	256	160	320	32	128	128
maxpool	3×3/2	7×7×832						
inception 5a Z		7×7×832	256	160	320	32	128	128
inception 5b		7×7×1024	384	192	384	48	128	128
avgpool	7×7/2	1×1×1024						
dropout		1×1×1024						
linear		1×1×1000						
softmax		1×1×1000						

Tabulka 2.2: Struktura sítě GoogLeNet, která je popsaná v podsekci 2.5.2.

bulce 2.2. Čísla ve sloupcích „redukce před 3×3“, „redukce před 5×5“ a „redukce před maxpool“ udávají na kolik map příznaků bude redukován vstup do modulu Inception před konvolucí s filtrem o velikosti 3×3, 5×5 a vrstvou maxpool. Sloupce „1×1“, „3×3“ a „5×5“ udávají počet map příznaků, které jsou výsledkem odpovídajících konvolucí. Všechny konvoluce (včetně těch v Inception modulech) jsou následovány ReLU nelinearitou. Síť obsahuje jednu plně propojenou vrstvu, která slouží k její jednoduché adaptaci na různé trénovací množiny. Síť GoogLeNet je tvořena 22 vrstvami s parametry a 5 vrstvami, které provádějí podvzorkování. Síť byla navržena tak, aby byla použitelná v praxi, a aby dopředný průchod sítí mohl být proveden i na zařízeních s omezeným výkonem.

Obavou při vytvoření této sítě byl problém mizejícího gradientu, který je popsaný v podsekci 3.2.2. Mělké sítě si s tímto problémem dokážou velmi dobře poradit. Řešením tohoto problému jsou postranní klasifikátory napojené na vrstvy ve střední části sítě. Architektura postranního klasifikátoru umístěného na výstup Inception modulu 4a je popsána v tabulce 2.3. Druhý postranní klasifikátor je umístěn na výstup Inception modulu 4d a liší pouze

Typ	Velikost filtru/krok	Velikost výstupu
avgpooling	5×5/3	4×4×512
konvoluční	1×1×128/1	4×4×128
plně propojená	2048×1024	1024
ReLU		
dropout 70%		1000
plně propojená	1024×1000	1000
softmax		

Tabulka 2.3: Struktura postranního klasifikátoru sítě GoogLeNet, který je popsán v podsekci 2.5.2.

v počtu map příznaků na vstupu prvních dvou vrstev. Síť je zakončena softmax vrstvou. Kompletní strukturu sítě GoogLeNet zobrazuje diagram D.2.

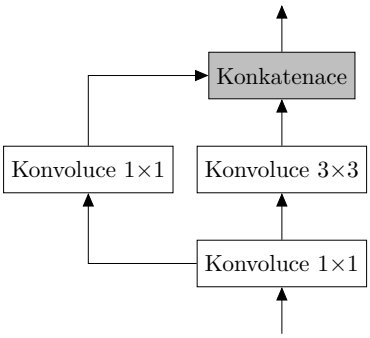
2.6 SqueezeNet

Síť SqueezeNet navrh tým výzkumníků z firmy DeepScale a Stanfordovy a Kalifornské univerzity v Berkeley[12]. Cílem většiny výzkumu v oblasti konvolučních neuronových sítí bylo zvýšení přesnosti. Kromě přesnosti je ale důležitou vlastností sítě počet jejích parametrů. Síť s nižším počtem parametrů se efektivněji trénuje na více počítačích. Délka komunikace mezi počítači je přímo úměrná počtu parametrů sítě, proto se sítě s nižším počtem parametrů trénují kratší dobu. Tato vlastnost se promítne i mimo trénováni sítě, kdy menší síť je možné rychleji nahrát do zařízení, které ji používá. Je tak možné provádět častější aktualizace vah sítě v zařízení. Další výhodou menších sítí je možnost jejich implementace v FPGA a použití ve vestavěných zařízeních. SqueezeNet je síť, která dosahuje podobné přesnosti jako AlexNet a přitom má padesátkrát méně parametrů.

2.6.1 Modul Fire

Tento modul je založený na následujících třech principech. Prvním je použití konvolucí s filtrem o velikosti 1×1 místo větších filtrů o velikosti 3×3. Tyto filtry mají devětkrát méně parametrů. Druhou strategií je snížení počtu map příznaků před 3×3 konvolucemi. Počet parametrů 3×3 konvoluce je závislý také na počtu map příznaků, které jsou vstupem a výstupem konvoluce. Snížení se provádí pomocí konvoluce, která má na výstupu méně map příznaků než na vstupu. Posledním principem je snižování velikosti map příznaků v pozdní části sítě. Velikost map příznaků sítě je závislá na velikosti vstupu a umístění vrstev pro podvzorkování. Pokud je podvzorkování umístěno v pozdní části sítě, pracuje síť s většími mapami příznaků. To zvyšuje přesnost sítě. První dva principy mají za úkol snížit počet parametrů sítě, třetí princip slouží k maximálnímu využití malého počtu parametrů sítě.

Modul Fire je definován pomocí dvou vrstev. První vrstva slouží ke snížení počtu map příznaků a je tvořena jednou konvolucí s filtrem o velikosti 1×1. Druhá vrstva se skládá ze dvou konvolucí, první je konvoluce s filtrem o velikosti 1×1, druhá je konvoluce s filtrem o velikosti 3×3. Vstupem obou konvolucí je výstup z první vrstvy. Výstupem celého modulu je konkatenace výstupů konvolucí ve druhé vrstvě. Struktura modulu Fire je zobrazena na diagramu 2.9.



Obrázek 2.9: Diagram modulu Fire sítě SqueezeNet, který je popsán v podsekci 2.6.1.

Typ	Velikost filtru/krok	Velikostvýstupu	Squeeze	1×1	3×3
konvoluce	7×7/2	112×112×96			
maxpool	3×3/2	56×56×96			
fire 1		56×56×128	16	64	64
fire 2		56×56×128	16	64	64
fire 3		56×56×256	32	128	128
maxpool	3×3/2	28×28×256			
fire 4		28×28×256	32	128	128
fire 5		28×28×384	48	192	192
fire 6		28×28×384	48	192	192
fire 7		28×28×512	64	256	256
maxpool	3×3/2	14×14×512			
fire 8		14×14×512	64	256	256
dropout		14×14×512			
konvoluce	1×1/1	14×14×1000			
avgpool	14×14/1	1×1×1000			

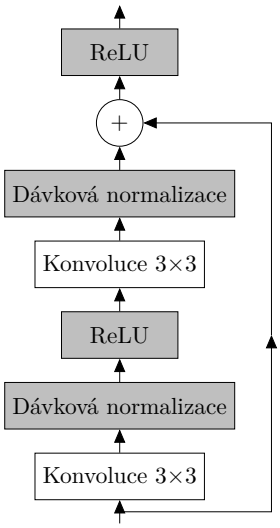
Tabulka 2.4: Struktura sítě SqueezeNet, která je popsaná v podsekci 2.6.2.

2.6.2 Popis sítě

SqueezeNet je konvoluční neuronová síť využívající modul Fire. Vstupem sítě je RGB obrázek o velikosti 224×224 pixelů. Podrobná architektura sítě je popsána v tabulce 2.4 a diagramu D.3. Čísla ve sloupci „Squeeze“ udávají na kolik map příznaků bude redukován vstup do modulu Fire. Sloupce 1×1 a 3×3 udávají počet map příznaků, které jsou výsledkem odpovídajících konvolucí. Všechny konvoluce (včetně těch ve Fire modulech) jsou následovány ReLU nelinearitou. SqueezeNet je tvořena 20 vrstvami s parametry a čtyřmi vrstvami, které provádějí podvzorkování. Poslední vrstvou je softmax vrstva.

2.7 Resnet34

Síť Resnet34 navrhl tým odborníků z firmy Microsoft v práci [9]. Hloubka konvoluční neuronové sítě je její velmi důležitou vlastností. Problémem při trénování velmi hlubokých sítí je problém mizejícího gradientu, který je popsaný v podsekci 3.2.2. Tento problém je možné obejít pomocí normalizované inicializace a normalizačních vrstev, takže váhy sítě začnou konvergovat. Druhým problémem je degradace při trénování. Když se zvyšuje počet vrstev sítě, přesnost se nezvyšuje a následně klesá. Nečekaně toto není způsobeno přetrénováním a přidání více vrstev zvýší chybovost modelu. Tato degradace ukazuje, že ne všechny modely se stejně dobře trénují.



Obrázek 2.10: Diagram jednoduchého reziduálního bloku sítě Resnet34, který je popsán v podsekci 2.7.1.

2.7.1 Reziduální blok

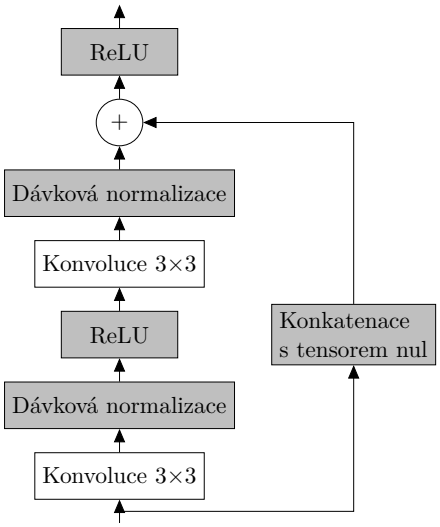
Nechť $\mathcal{H}(x)$ je funkce, kterou je možné aproximovat pomocí několika vrstev sítě (nemusí to být celá síť), kde x je vstup do první vrstvy sítě. Za předpokladu, že vícevrstvé sítě dokážou aproximovat složité funkce, tak můžou aproximovat i zbytkové funkce jako například $\mathcal{H}(x) - x$. Předpokládá se, že vstup a výstup funkce má stejné rozměry. Než se snažit natrénovat síť na aproximaci $\mathcal{H}(x)$, je lepší síť natrénovat na aproximaci zbytkové funkce $\mathcal{F}(x) = \mathcal{H}(x) - x$. Původní funkci je tak možné vyjádřit jako $\mathcal{H}(x) = \mathcal{F}(x) + x$. Přestože oba zápisy funkce $\mathcal{H}(x)$ je možné aproximovat pomocí neuronové sítě, jednoduchost trénování může být odlišná. Zápis funkce $\mathcal{H}(x) = \mathcal{F}(x) + x$ je možné realizovat jako neuronovou síť s dopřednými spojeními.

Experimenty ukázaly, že síť sestavená z reziduálních bloků, které jsou tvořeny ze dvou nebo více konvolucí, dosahuje lepších výsledků než nereziduální síť nebo reziduální síť sestavená z reziduálních bloků, které jsou tvořeny jednou konvolucí, s podobnou strukturou a stejným počtem parametrů. V každém reziduálním bloku byla za každou konvolucí umístěna dávková normalizace a za první normalizaci a sumu byla umístěna jednotka ReLU. Diagram 2.10 ukazuje jeden reziduální blok.

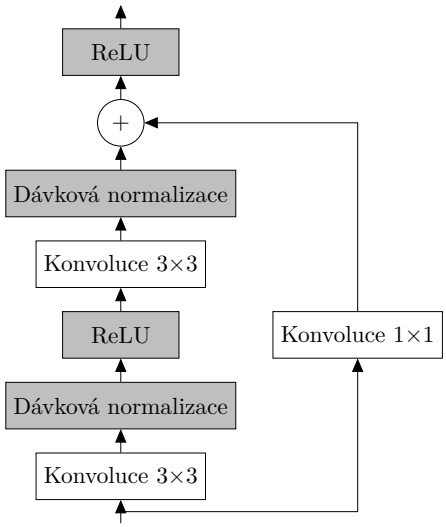
2.7.2 Popis sítě

Resnet34 je konvoluční neuronová síť využívající reziduální bloky popsané v předchozí podkapitole. Podrobná architektura sítě je popsána v tabulce 2.5. Celkem se skládá z 34 vrstev obsahujících parametry, z toho 32 vrstev tvoří 16 reziduálních jednotek. Resnet34 je ukončena vrstvou avgpool, plně propojenou vrstvou a vrstvou softmax.

Důležitou částí této sítě je podvzorkování, které se provádí v reziduálních jednotkách pomocí kroku první konvoluce o velikosti 2. Současně s podvzorkováním je také provedeno zdvojnásobení počtu map příznaků, aby všechny bloky měly podobnou výpočetní náročnost. Zde nastává problém, protože výstup konvoluce má jiné rozměry než vstup do reziduálního jádra. Řešením je zvýšení počtu map příznaků ve vstupu před jeho sečtením s výstupem z konvoluce. Existují dva přístupy: konkatenace s tensorem nul a konvoluce s jádrem o ve-



Obrázek 2.11: Diagram reziduálního bloku sítě Resnet34 s konkatenací s tensorem nul pro zvýšení počtu map příznaků, který je popsán v podsekci 2.7.1.



Obrázek 2.12: Diagram reziduálního bloku sítě Resnet34 s konvolucí pro zvýšení počtu map příznaků, který je popsán v podsekci 2.7.1.

Typ	Velikost filtru/krok	Velikost výstupu	Počet	Změna velikosti a počtu map příznaků
konvoluce	7×7/2	112×112×64	1	
maxpool	3×3/2	56×56×64	1	
residualní		56×56×64	3	NE
residualní		28×28×128	1	ANO
residualní		28×28×128	3	NE
residualní		14×14×256	1	ANO
residualní		14×14×256	5	NE
residualní		7×7×512	1	ANO
residualní		7×7×512	2	NE
avgpool	7×7/1	1×1×512	1	
plně propojená softmax	1000	1000	1	

Tabulka 2.5: Struktura sítě Resnet34, která je popsaná v podsekci 2.7.2.

likosti 1×1. Oba způsoby jsou zobrazeny na diagramech 2.11 a 2.12. Konvoluce dosahuje mírně lepších výsledků, protože zvyšuje počet parametrů bloku. Podrobná architektura sítě je popsána v tabulce 2.5 a diagramu D.4.

Kapitola 3

Trénování sítí

Velmi důležitou oblastí z teorie neuronových sítí je jejich trénování. V této kapitole jsou popsány způsoby trénování. Samotné trénovací algoritmy jsou popsány v první sekci. Sekce 3.2 pak popisuje výpočet gradientu, který se používá v algoritmech z předchozí sekce. Konec této kapitoly je věnován algoritmům pro trénování na více výpočetních jednotkách.

Trénováním sítí se zabývá Raúl Rojas v knize Neural Networks: A Systematic Introduction [5]. Trénování sítě je vhodné nastavení vah sítě tak, aby síť pracovala z co nejlépe. Existuje několik algoritmů pro trénování, ale nejpoužívanější jsou algoritmy založené na zpětné propagaci chyby sítě.

3.1 Trénovací algoritmy

Předpokládejme, že dopředná neuronová síť má n vstupních a m výstupních jednotek. Může se skládat z libovolného počtu skrytých jednotek a může být propojena jakýmkoliv dopředným způsobem. Trénovací sada $\{(x_1, t_1), ..., (x_q, t_q)\}$ této sítě se skládá z q dvojic $n$rozměrného a $m$rozměrného vektoru, které nazývají vstupní a výstupní vzor. Když je vektor x_p dán na vstup sítě, vyprodukuje síť výstup y_p, který se obecně liší od vzoru t_p. Trénování slouží ke snížení rozdílu mezi y_p a t_p pro všechna $p \in \langle 1, q \rangle$. Přesněji řečeno: cílem trénování sítě je minimalizace chybové funkce sítě E_p definované následujícím způsobem

$$E_p = \frac{1}{2} \sum_{i=1}^{m} (y_{p,i} - t_{p,i})^2 \tag{3.1}$$

pro všechny vzory (x_p, t_p).

Po natrénování bude síť provádět interpolaci, to znamená, že pro nové vstupní vzory podobné těm, které byly použity při trénování, bude dávat podobné výstupy. Algoritmus zpětné propagace se používá pro nalezení lokálního minima chybové funkce sítě. Váhy sítě jsou inicializovány náhodnými hodnotami.

3.1.1 Stochastický gradientní sestup

Tento algoritmus je nejjednodušší algoritmus pro trénování hlubokých neuronových sítí. Stochastický gradientní sestup (Stochastic gradient descent, SGD) je inkrementální algoritmus založený na úpravě vah sítě podle jejích gradientů. Vzorec

$$w_{i,j} = w_{i,j} - \eta \nabla_p w_{i,j} \tag{3.2}$$

slouží k výpočtu velikosti váhy $w_{i,j}$ po natrénování vzoru p, kde $\nabla_p w_{i,j}$ je gradient sítě definovaný v sekci 3.2 vzorcem 3.8 . Gradient říká jakým způsobem upravit velikost váhy, to znamená, že určuje jestli se velikost váhy má snížit nebo zvýšit. Symbol η značí trénovací koeficient, který ovlivňuje rychlost trénování sítě díky tomu, že ovlivňuje velikosti změny vah sítě. Vyšší trénovací koeficient zrychluje učení, protože změny vah budou větší, ale nižší zvyšuje přesnost učení, protože malé změny vah dokážou lépe aproximovat ideální hodnotu vah. Hlavní výhodou tohoto algoritmu je jeho jednoduchá implementace a s tím související nízká výpočetní náročnost. Činnost tohoto algoritmu je popsána algoritmem 3.1, který popisuje natrénování sítě pomocí N epoch a trénovací množiny, která se skládá z P dvojic vzorů.

Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, učící koeficient η.
Output: Natrénované váhy W neuronové sítě.
inicializace vah sítě W_0
$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    $t \leftarrow t + 1$
    for $w_{i,j}^t \in W_t$ **do**
      $w_{i,j}^t \leftarrow w_{i,j}^{t-1} - \eta \nabla_{n,p} w_{i,j}^{t-1}$
    end
  end
end
return W_t

Algoritmus 3.1: Trénovací algoritmus stochastický gradientní sestup.

3.1.2 Algoritmus Adam

Tento algoritmus navrhli Diederik Kingma a Jimmy Lei Ba ve článku [14]. Tento způsob trénování je detailně popsán algoritmem 3.2. Kde β_1^t a β_1^t značí umocnění každého prvku matic β_1 a β_1 na $t$tou, g_t^2 značí umocnění každého prvku matice g_t na druhou, $\beta_1 m_{t-1}$ je vynásobení matice m_{t-1} skalárem β_1.

Adam si zaznamenává exponenciální pohybující se průměry změny vah m_t podle algoritmu SGD a rozptyl těchto změn v_t, kde hyperparametry $\beta_1, \beta_2 \in \langle 0, 1)$ jsou váhy. které kontrolují jakým způsobem se aktualizuje hodnota průměru vzhledem k nejnovější hodnotě. Tyto průměry jsou odhadem prvního a druhého momentu změny váhy. Protože jsou oba průměry inicializovány na hodnotu nula, jsou tyto odhady vychýlené k nule. To je obzvláště výrazné během počátečních kroků algoritmu, a když se hodnoty β_1 a β_2 blíží jedničce. Řešením tohoto problému je použití upravených hodnot $\hat{m}_t$ a $\hat{v}_t$. Doporučenými hodnotami pro parametry algoritmu jsou $\alpha = 0,001$, $\beta_1 = 0,9$, $\beta_2 = 0,999$ a $\epsilon = 10^{-8}$.

3.2 Výpočet gradientu sítě

Vzorce pro výpočet gradientu sítě odvodili D. Rumelhart, G. Hinton, a R. Williams v článku [21]. Jako první se vypočítá výstup sítě pro zadaný vstup. To znamená, že dojde k propagaci vstupu na výstup. Zpětnou propagací výstupu sítě skrz síť za pomoci požadovaného výstupu

Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, učící koeficient η.
Output: Natrénované váhy W neuronové sítě.
inicializace vah sítě W_0
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \eta \nabla_{n,p} W$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    $W_t \leftarrow W_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
  end
end
return W_t

Algoritmus 3.2: Trénovací algoritmus Adam.

se vypočítávají rozdíly mezi reálnými výstupy a požadovanými výstupy všech neuronů sítě. Z rozdílů výstupů jsou vypočítány gradienty vah sítě, které určují jak se mají upravit jednotlivé váhy sítě, aby došlo ke snížení chyby sítě. Pro výpočet gradientů sítě se používá vzorec

$$\nabla_p w_{i,j} = \delta_{p,j} y_{p,i}, \tag{3.3}$$

kde $\delta_{p,j}$ je chybový signál, který je dostupný $j$tému neuronu vrstvy, a $y_{p,i}$ je výstup $i$tého neuronu předchozí vrstvy nebo $i$tá hodnota vstupu do neuronové sítě, pokud je tato vrstva první vrstvou sítě. Obě veličiny jsou vzhledem k $p$tému vzoru.

Následující dva vzorce definují hodnotu chybového signálu $\delta_{p,j}$ podle toho, ve které vrstvě se trénovaný neuron nachází. První je vzorec

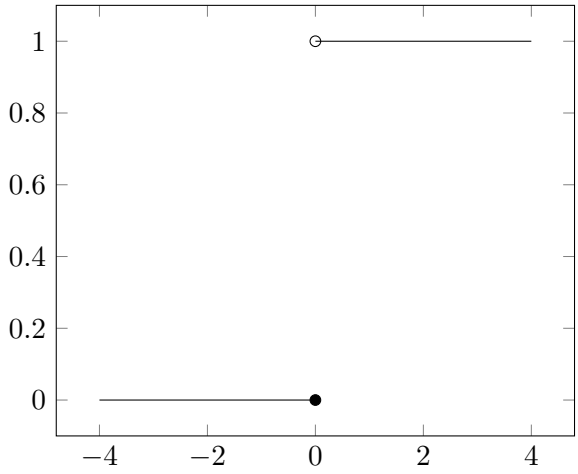
$$\delta_{p,j} = (y_{p,j} - t_{p,j}) f'(\xi_{p,j}), \tag{3.4}$$

který určuje hodnotu chybového signálu pro neuron, který je ve výstupní vrstvě. V tomto vzorci $y_{p,j}$ představuje výstup $j$tého neuronu výstupní vrstvy, $t_{p,j}$ je odpovídající vzor této výstupní hodnoty z trénovací množiny, f' je derivace aktivační funkce neuronu a $\xi_{p,j}$ je výstup lineární jednotky $j$tého neuronu výstupní vrstvy před aplikací aktivační funkce (viz sekce 2.1 a vzorec 2.1).

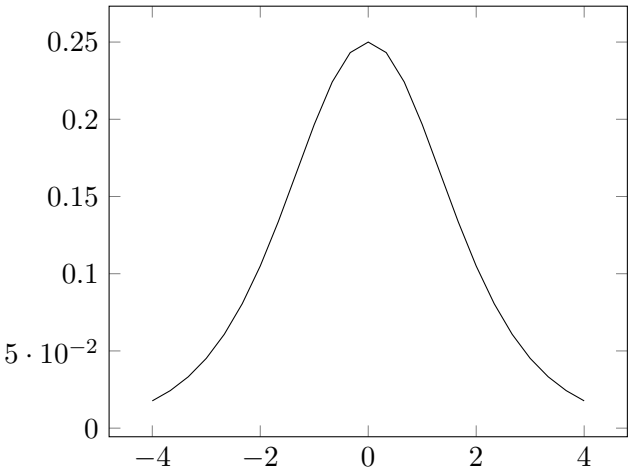
Pokud se neuron nachází v jiné než výstupní vrstvě je nutné použít vzorec

$$\delta_{p,j} = \sum_k (\delta_{p,k} w_{j,k}) f'(\xi_{p,j}) \tag{3.5}$$

pro výpočet chybového signálu $\delta_{p,j}$. Tento vzorec je rekurentní, protože pro výpočet hodnot chybového signálu $\delta_{p,j}$ ve skryté vrstvě jsou potřeba hodnoty chybového signálu $\delta_{p,k}$ v následujících vrstvách. Obdobně jako v předchozích vzorcích $w_{j,k}$ je váha mezi $j$tým neuronem skryté vrstvy a $k$tým neuronem následující vrstvy.



Obrázek 3.1: Graf derivace aktivační usměrněné lineární funkce.



Obrázek 3.2: Graf derivace aktivační logistické funkce sigmoida.

Ve vzorcích 3.4 a 3.5 se používá funkce f', která je derivací aktivační funkce neuronu. Velmi často se jako aktivační funkce používá usměrněná lineární funkce popsaná rovnicí 2.3. Vzorec

$$y' = \begin{cases} 1 & \text{pokud } x > 0 \\ 0 & \text{pokud } x \leq 0 \end{cases} \tag{3.6}$$

se používá jako derivace této funkce. Graf 3.1 ilustruje průběh této funkce. Tento vzorec obchází problém ReLU, kterým je, že neexistuje její derivace v bodě 0, tak, že tuto hodnotu definuje jako 0.

Druhou velmi používanou aktivační funkcí je logistické funkce sigmoida, která je popsaná vzorcem 2.4. Derivace sigmoidy je zapsána ve vzorci

$$y' = \frac{e^{-x}}{(1 + e^{-x})^2}, \tag{3.7}$$

který je graficky znázorněn pomocí grafu 3.2. Grafy derivací dobře ilustrují hlavní výhodu sigmoidy jako aktivační funkce, kterou je, že její derivace je definována pro všechny body.

3.2.1 Odvození vzorců pro zpětnou propagaci

Odvození vzorců prezentovaných v na začátku sekce 3.2 provedli D. Rumelhart, G. Hinton, a R. Williams v článku [21]. Gradient vyjadřuje rychlost, jakou se veličina zvyšuje nebo snižuje v poměru ke změnám dané proměnné. Zpětná propagace je gradientní metoda, to znamená, že je založená na výpočtu gradientu sítě vzhledem k jednotlivým vahám sítě. Tento gradient se vypočítá jako parciální derivace chybové funkce vzhledem k jedné z vah sítě. Vzorec

$$\nabla_p w_{j,i} = \frac{\partial E_p}{\partial w_{j,i}} \tag{3.8}$$

je zápis této derivace pro váhu $w_{j,i}$.

Řetízkové pravidlo derivací. Toto pravidlo slouží ke zjednodušení výpočtu derivace funkce $F(x)$ podle x. Pravidlo bylo použito ve vzorcích 3.10 a 3.11. Pokud $F(x) = f(g(x))$ potom platí

$$\frac{\partial F}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}. \tag{3.9}$$

Vzorec 3.10 je vzorec výpočtu změny váhy $w_{j,i}$ výstupní vrstvy sítě pro vzor p trénovací množiny. Kde E_p je chybová funkce pro vzor p, $w_{j,i}$ je váha mezi $i$tým vstupem do vrstvy a $j$tým neuronem výstupní vrstvy, $\xi_{p,j}$ je výstup lineární jednotky $j$tého neuronu před aplikací aktivační funkce (viz sekce 2.1 a vzorec 2.1), $w_{k,j}$ je váha mezi $k$tým vstupem do vrstvy a $j$tým neuronem výstupní vrstvy, $y_{p,j}$ je výstup $j$tého neuronu. Index p značí, že se jedná o hodnotu vypočítanou vzhledem ke vzoru p z trénovací množiny.

$$\begin{aligned}
\nabla_p w_{j,i} &= \frac{\partial E_p}{\partial w_{j,i}} = \frac{\partial E_p}{\partial \xi_{p,j}}\frac{\partial \xi_{p,j}}{\partial w_{j,i}} = \frac{\partial E_p}{\partial \xi_{p,j}}\frac{\partial \sum_k w_{j,k} y_{p,k}}{\partial w_{j,i}} = \\
&= \frac{\partial E_p}{\partial \xi_{p,j}} y_{p,i} = \frac{\partial E_p}{\partial y_{p,j}}\frac{\partial y_{p,j}}{\partial \xi_{p,j}} y_{p,i} = (y_{p,j} - t_{p,j}) g'(\xi_{p,j}) y_{p,i} = \\
&= \delta_{p,j} y_{p,i}
\end{aligned} \tag{3.10}$$

Vzorec 3.11 slouží k výpočtu změny váhy $w_{j,i}$ skryté vrstvy sítě pro vzor p trénovací množiny. Kde E_p je chybová funkce pro vzor p, $w_{j,i}$ je váha mezi $i$tým vstupem do vrstvy a $j$tým neuronem skryté vrstvy, $\xi_{p,j}$ je výstup lineární jednotky $j$tého neuronu před aplikací aktivační funkce (viz sekce 2.1 a vzorec 2.1), $w_{k,j}$ je váha mezi $j$tým neuronem skryté vrstvy a $k$tým neuronem následující vrstvy, $y_{p,i}$ a $y_{p,j}$ jsou výstupy $i$tého a $j$tého neuronu. Index p značí, že se jedná o hodnotu vypočítanou vzhledem ke vzoru p z trénovací množiny.

$$\begin{aligned}
\nabla_p w_{j,i} &= \frac{\partial E_p}{\partial w_{j,i}} = \frac{\partial E_p}{\partial \xi_{p,j}}\frac{\partial \xi_{p,j}}{\partial w_{j,i}} = \frac{\partial E_p}{\partial \xi_{p,j}} y_{p,i} = \frac{\partial E_p}{\partial y_{p,j}}\frac{\partial y_{p,j}}{\partial \xi_{p,j}} y_{p,i} = \\
&= \frac{\partial E_p}{\partial y_{p,j}} g'(\xi_{p,j}) y_{p,i} = \sum_k \left(\frac{\partial E_p}{\partial \xi_{p,k}}\frac{\partial \xi_{p,k}}{\partial y_{p,j}} \right) g'(\xi_{p,j}) y_{p,i} = \\
&= \sum_k \left(\frac{\partial E_p}{\partial \xi_{p,k}}\frac{\partial \sum_l w_{k,l} y_{p,l}}{\partial y_{p,j}} \right) g'(\xi_{p,j}) y_{p,i} = \sum_k \left(\frac{\partial E_p}{\partial \xi_{p,k}} w_{k,j} \right) g'(\xi_{p,j}) y_{p,i} = \\
&= \sum_k (\delta_{p,k} w_{k,j}) g'(\xi_{p,j}) y_{p,i} = \sum_k (\delta_{p,k} w_{k,j}) g'(\xi_{p,j}) y_{p,i} = \\
&= \delta_{i,j} y_{p,i}
\end{aligned} \tag{3.11}$$

3.2.2 Problém mizejícího gradientu

Sepp Hochreiter ve své diplomové práci [10] v roce 1991 formálně popsal problém mizejícího gradientu. Odvození tohoto problému následně také popsal v práci [11]. Tento problém nastává při trénování neuronových sítí pomocí algoritmů založených na gradientu sítě popsaném v sekci 3.2.

Tyto algoritmy používají změnu vah, která je závislá na velikosti chybového signálu a současné velikosti váhy. Toto je popsané rovnicí 3.8. Velikost chybového signálu je přímo úměrná velikosti derivace aktivační funkce. Derivace běžné aktivační funkce sigmoidy má výstup v intervalu $(-1, 1)$. Zpětná propagace vypočítává gradient pomocí řetízkového pravidla. To znamená, že pro výpočet gradientu váhy neuronu v první vrstvě $n$vrstvé sítě je potřeba vynásobit chybový signál $n$krát. Výsledkem tohoto jevu je, že velikost chybového signálu se exponenciálně snižuje při propagaci chyb od výstupní vrstvy ke vstupní vrstvě. Následkem toho jsou malé změny vah ve vrstvách blízkých vstupu a pomalé trénování těchto vrstev.

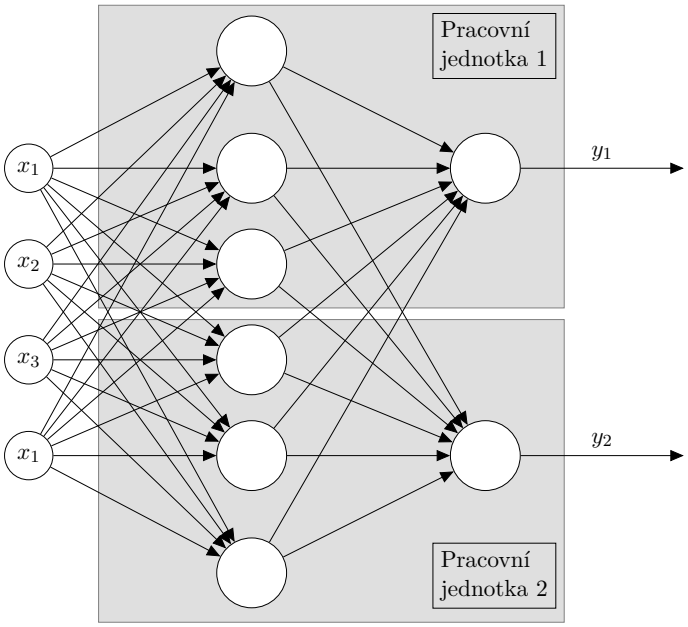
Tento problém postihuje hlavně sítě s velkým počtem vrstev. Opakem tohoto problému je problém explodujícího gradientu, kdy je velikost výstupu derivace aktivační funkce mimo interval $(-1, 1)$. V tomto případě může docházet k exponenciálnímu nárůstu velikosti chybového signálu. To způsobí oscilaci vah, kdy je velikost vah upravována velmi velkými změnami, které nesnižují hodnotu chybové funkce. Neuronové sítě problém mizejícího gradientu řeší různými způsoby. Například síť GoogLeNet používá postranní klasifikátory, které zvyšují hodnotu gradientu díky tomu, že mají nižší hloubku než zbytek sítě. Chybový signál vypočtený z těchto klasifikátorů je tak méně ovlivněný průchodem vrstvami. Síť ResNet využívá reziduálních spojení, které „přeskakují“ některé vrstvy. To způsobuje, že chybový signál je ovlivněn i hlubšími vrstvami. Tyto sítě jsou popsané v sekcích 2.5 a 2.7.

3.3 Distribuované trénování

Autoři systému Apache Singa [18] popisují dva přístupy pro distribuované trénování: modelový paralelizmus a datový paralelizmus.

V modelovém paralelizmu je model rozdělen na několik částí, které jsou umístěny na několik výpočetních jednotek, které počítají pouze část změn vah sítě. Výhodou tohoto přístupu je to, že umožňuje trénování velkých sítí, které jsou rozdělené na menší podsítě. V modelovém paralelizmu se ale musí přenášet mezivýsledky trénování mezi výpočetními jednotkami. Na obrázku 3.3 je zobrazen modelový paralelizmus na dvou výpočetních jednotkách. K přenosům mezivýsledků dochází na hranách grafu, které vedou mezi oblastmi označenými jako „Pracovní jednotka 1“ a „Pracovní jednotka 2“.

Datový paralelizmus je situace, kdy trénovací data jsou rozdělena na několik částí a každá část je natrénována na jiné výpočetní jednotce. Hlavní výhodou datového paralelizmu je zrychlení natrénování sítě trénovací sadou. Nevýhodou je to, že se celá neuronová síť musí uložit do paměti výpočetní jednotky, to klade omezení na její velikost. Druhou nevýhodou je to, že se na každé výpočetní jednotce zároveň počítají různé hodnoty gradientů podle různých dat. Je tak nutné vyřešit jak podle nich aktualizovat hodnoty vah. Dalším problém je to, že každá jednotka potřebuje všechny váhy sítě. Je tak potřeba zajistit sdílení vah mezi jednotkami. To znamená, že budou potřeba přenosy dat a gradientů mezi jednotkami. Tyto problémy řeší algoritmy pro synchronní a asynchronní trénování. Obrázek 3.4 ilustruje datový paralelizmus na dvou výpočetních jednotkách.



Obrázek 3.3: Neuronová síť rozdělená na dvě části, které se trénují na různých výpočetních jednotkách.

Je také možné oba přístupy zkombinovat. Tento přístup v sobě spojuje výhody i nevýhody modelového a datového paralelizmu. Díky tomu je možné rychleji trénovat větší sítě, ale cenou za to jsou přenosy mezivýsledků, vah i gradientů.

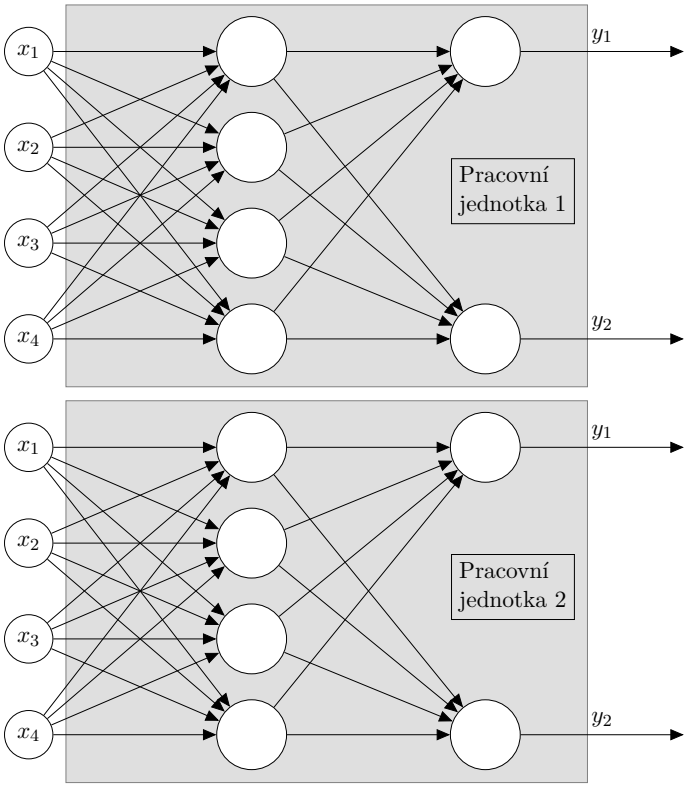
3.3.1 Synchronní trénování

Synchronní trénování použili Suyog Gupta, Wei Zhang a Josh Milthorpe v práci [8]. Algoritmus popsaný v této práci se nazývá Hardsync. Bližší popis tohoto algoritmu je v práci [26] od pracovníků z laboratoří firmy Yahoo!.

Při synchronním trénování jsou na všech výpočetních jednotkách spočítány gradienty pomocí stejných vah sítě. Aktualizace vah sítě je provedena pomocí gradientů ze všech výpočetních jednotek. Synchronizace spočívá v tom, že jednotky, které ukončí výpočet gradientů dříve, čekají než je ukončen výpočet na všech ostatních jednotkách. Následně je provedena aktualizace vah sítě pomocí agregace změn vah sítě. Po aktualizaci si všechny jednotky stáhnou nové hodnoty vah. Výhodou tohoto přístupu je to, že je založený sekvenční implementaci trénování. Agregace dávek je založena na agregaci změn vah sítě při trénování dávkami z trénovací množiny a je definována následujícím způsobem

$$\nabla W^k = \frac{1}{N} \sum_{i=1}^{N} \nabla W_i^k, \tag{3.12}$$

kde ∇W^k je agregovaný gradient pro $k$tou dávku, ∇W_i^k je vypočtený gradient pro $k$tou dávku z $i$té pracovní jednotky z celkového počtu N. Proto je možné pro synchronní trénování říci následující. Pokud n pracovních jednotek provede trénování sítě dávkou o velikosti d pomocí synchronního trénování, odpovídá to tomu, jako kdyby jedna pracovní jednotka provedla trénování sítě jednou dávkou o velikosti nd. Činnost jedné pracovní jednotky je popsána algoritmem 3.3, činnost serveru 3.4.



Obrázek 3.4: Dvě kopie neuronové sítě, které se trénují na různých výpočetních jednotkách.

Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, server s parametry s.

$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    $t \leftarrow t + 1$
    $B_t \leftarrow$ vyber_příklad()
    $W_t \leftarrow$ přijmi_nebo_čekej_na_váhy_od_serveru(s, t)
    $\Delta W_t \leftarrow$ vypočítej_změny_vah(W_t, B_t)
    odešli_změny_vah_na_server$(s, \Delta W_t)$
  end
end

Algoritmus 3.3: Činnost jedné pracovní jednotky při trénování pomocí algoritmu Hard-sync.

Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, množina pracovních jednotek M.
Output: Natrénované váhy W neuronové sítě.
inicializace vah sítě W_0
$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    $t \leftarrow t + 1$
    pošli_všem_pracovníkům_váhy(M, W_{t-1}, t)
    $\{\Delta W_t\} \leftarrow$ přijmi_změny_vah_od_všech_pracovníků(M)
    $\Delta W_t \leftarrow$ agreguj_změny_vah$(\{\Delta W_t\})$
    $W_t \leftarrow$ aktualizuj_váhy_sítě$(W_{t-1}, \Delta W_t)$
  end
end
return W_t

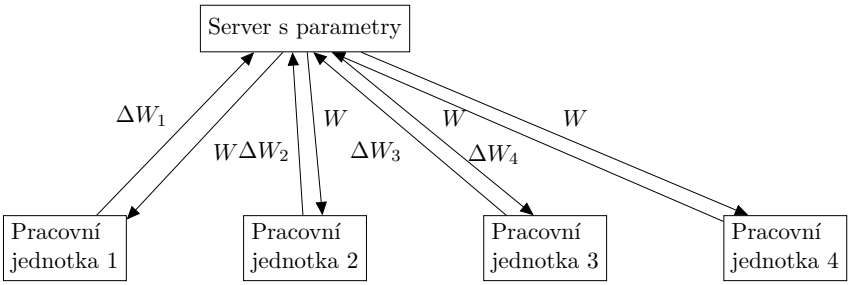
Algoritmus 3.4: Činnost serveru s parametry při trénování pomocí algoritmu Hardsync.

3.3.2 Asynchronní trénování

Metoda SGD popsaná v podsekci 3.1.1 je nejčastěji používaný optimalizační algoritmus pro trénování neuronových sítí. Bohužel formulace tohoto algoritmu je ve své podstatě sekvenční. To znamená, že je nepraktický pro trénování při použití velmi velké trénovací množiny, kdy je potřeba několikrát natrénovat všechny vzory v množině. Sekvenční trénování takové množiny bude velmi časově náročné, proto může být výhodnější použít paralelní trénování na více pracovních jednotkách.

V práci [3] je předveden algoritmus Downpour SGD, který je variací na asynchronní SGD, který používá několik replik jedné neuronové sítě. Tento algoritmus pracuje následujícím způsobem. Jako první se rozdělí trénovací data na několik podmnožin a každá tato podmnožina se použije na natrénování repliky sítě na různých výpočetních jednotkách. Všechny repliky komunikují pomocí centralizovaného serveru s parametry, který uchovává současný stav všech vah sítě sdílený přes všechny výpočetní jednotky. Je také možné použít více než jeden server pro uložení parametrů. V tom případě pak každý server má na starosti sdílení a aktualizace pouze části vah sítě. Tento přístup je asynchronní, protože každá výpočetní jednotka pracuje nezávisle na ostatních výpočetních jednotkách, a protože každý server s parametry pracuje nezávisle na ostatních serverech s parametry. V Nejjednodušší implementaci algoritmu se před výpočtem změn vah sítě přenesou nejnovější hodnoty vah sítě ze serveru s parametry na výpočetní jednotku, aby se následující změny vah nepočítaly pomocí zastaralých hodnot. Po obdržení nejnovějších hodnot vah, jsou na výpočetní jednotce vypočítány změny vah sítě, které se následně odešlou na server s parametry. Server provede aktualizaci vah sítě podle těchto změn. Činnost jedné pracovní jednotky je popsána algoritmem 3.5, činnost serveru 3.6.

Výhodou algoritmu Downpour SGD je jeho vysoká robustnost. Když při synchronním trénování dojde k nečekanému ukončení činnosti jedné výpočetní jednotky, trénování se zastaví. V případě asynchronního trénování ostatní jednotky budou dále pokračovat v trénování, protože jsou na sobě nezávislé. Na druhou stranu současné asynchronní trénování pomocí algoritmu Downpour SGD na více výpočetních jednotkách zavádí dodatečnou náhodnost do optimalizačního procesu. Výpočetní jednotky počítají změny vah sítě na základě mírně zastaralých hodnot vah, protože během tohoto výpočtu určitě došlo k aktualizaci



Obrázek 3.5: Komunikační diagram zobrazující interakce mezi čtyřmi pracovními jednotkami a jedním serverem s parametry při asynchronním trénování.

hodnot vah sítě na serveru s parametry. Při aktualizaci hodnot vah sítě na serveru se tak nejnovější hodnoty vah aktualizují pomocí hodnot změn, které byly vypočítány pomocí jiných starších hodnot vah. Pokud jsou hodnoty uloženy na více serverech, může nastat situace, kdy hodnoty vah na jednom serveru jsou starší než hodnoty na jiném serveru, nebo kdy jsou aktualizace vah provedeny na různých serverech v různém pořadí. Výpočetní jednotka si takové váhy může stáhnout a trénovat tak pomocí vah, které byly aktualizovány různým počtem změn vah v různém pořadí. Příklad komunikace při asynchronním trénování je zobrazen na obrázku 3.5.

Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, neuronová síť, server s parametry s.
$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    $t \leftarrow t + 1$
    $B_t \leftarrow$ vyber_příklad()
    $W_t \leftarrow$ přijmi_váhy_od_serveru(s)
    $\Delta W_t \leftarrow$ vypočítej_změny_vah(W_t, B_t)
    odešli_změny_vah_na_server($s, \Delta W_t$)
  end
end

Algoritmus 3.5: Činnost jedné pracovní jednotky při trénování pomocí algoritmu Downpour SGD.

3.3.3 Distribuovaný server

Nikko Strom ve své práci [24] představil velmi zajímavou architekturu pro paralelní trénování neuronových sítí. Hlavním rysem této architektury je to, že nepoužívá centrální server, na kterém jsou uložené parametry. Místo toho jsou parametry uložené na všech jednotkách a tyto jednotky si mezi sebou posílají gradienty. Druhým rysem této architektury je vysoká míra kvantizace a využití řídkých gradientů, pro snížení komunikační zátěže. V práci [3] je prezentována myšlenka použití více serverů s parametry, kde každý server uchovává pouze část vah sítě.

Distribuovaný server je kombinací myšlenek uvedených v těchto dvou pracích. Před započtením výpočtu jsou vahám přiřazeny jednotky, na kterých budou uložené. Algoritmus 3.7 popisuje jak jsou váhám přiřazeny jednotky. Každé váze je přiřazena právě jedna jednotka.

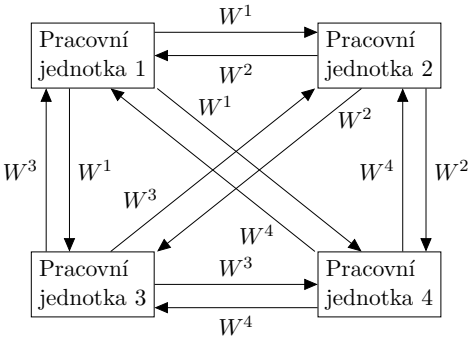
Input: Trénovací sada obsahující P dvojic vzorů, počet epoch N, neuronová síť, množina pracovních jednotek M.
Output: Natrénované váhy W neuronové sítě.
inicializace vah sítě W_0
$t \leftarrow 0$
for $n \in \langle 1, N \rangle$ **do**
  for $p \in \langle 1, P \rangle$ **do**
    for $m \in M$ **do in parallel**
      $t \leftarrow t + 1$
      pošli_pracovníkovi_váhy(m, W_{t-1})
      $\Delta W_t \leftarrow$ přijmi_změny_vah_od_pracovníka(m)
      $W_t \leftarrow$ aktualizuj_váhy_sítě$(W_{t-1}, \Delta W_t)$
    end
  end
end
return W_t

Algoritmus 3.6: Činnost serveru s parametry při trénování pomocí algoritmu Downpour SGD.

Na této jednotce vždy bude uložena nejnovější hodnota této váhy. Ostatní jednotky budou odesílat gradienty váhy, té jednotce, která je váze přiřazená. Tato jednotka tudíž provede aktualizaci hodnoty této váhy a také odesílá nejnovější hodnotu váhy ostatním jednotkám. Jednotky se tak chovají jako výpočetní jednotky i servery s parametry. Cílem distribuovaného serveru je lepší využití duplexních vlastností jednotek. Na obrázku 3.6 demonstruji rozeslání nejnovějších hodnot vah všem jednotkám.

Input: Seznam tenzorů vah W, seznam jednotek pro uložení vah N.
Output: Relace WN přidělující každému tenzoru vah jednotku.
nastav ohodnocení NS všech jednotek na 0
for $w \in W$ **do**
  $s \leftarrow$ vyber_jednotku_s_nejnižším_ohodnocením(N, NS)
  $NS(s) \leftarrow NS(s) + velikost(w)$
  $(w, s) \in WN$
end
return WN

Algoritmus 3.7: Algoritmus pro rozdělení vah uložených jako seznam tenzorů W na N jednotek.



Obrázek 3.6: Komunikační diagram zobrazující interakce mezi čtyřmi pracovními jednotkami používajícími distribuovaný server s parametry, kde $W^i \subset W$ je množina vah, které byly přiděleny na jednotku i.

Kapitola 4

Odhad zrychlení

V této kapitole se budu zabývat popisem nejdůležitější části mé práce. Tou jsou odhady doby paralelního trénování neuronových sítí podle použité techniky trénování. Paralelní trénování je buď synchronní nebo asynchronní, odhady dob trénování jsou v sekcích 4.1 a 4.2. Zrychlení trénování sítě je věnována poslední část této kapitoly.

Každý algoritmus trénování popsaný v kapitole 3.1, se skládá z přípravy na trénování, samotného trénování a ukončení trénování. Celkovou délku výpočtu tak můžu zapsat jako

$$t_{comp} = t_{prepare} + t_{train} + t_{finish}. \tag{4.1}$$

Předpokládám, že délky $t_{prepare}$ a t_{finish} jsou velmi krátké ve srovnání s t_{train}. Ve vzorcích v této kapitole předpokládám, že všechny pracovní jednotky mají stejnou rychlost komunikace se severem t_{comm} a délka výpočtu gradientů na pracovní jednotce t_{grad} je také stejná.

4.1 Synchronní trénování

Synchronní trénování bylo popsáno v podsekci 3.3.1. Důležitou vlastností tohoto způsobu trénování je to, že je možné ho rozdělit na iterace. Ty jsou od sebe oddělené a jediným způsobem komunikace mezi nimi je přenos hodnot vah neuronové sítě na serveru s parametry. Díky tomu můžu provést odhad celkové délky trénování t_{train} jako

$$t_{train} = It_{iter}, \tag{4.2}$$

kde I je počet po sobě jdoucích iterací synchronního trénování a t_{iter} je délka výpočtu jedné iterace trénování. Počet iterací je obvykle zadaný, proto se zaměřím na odhad délky trénování jedné iterace algoritmu.

4.1.1 Pomocí centrálního serveru

Při tomto způsobu trénování se používá jeden centrální server, na kterém jsou uložené nejnovější hodnoty vah trénované sítě. Centrální server je popsaný v podsekci 3.3.1. Při trénování pomocí centrálního serveru se používají dva typy jednotek: výpočetní jednotky a server s parametry. Tyto provádějí odlišné činnosti trvající odlišnou dobu, a proto je nutné vzít oba druhy jednotek v potaz. Délka iterace se tak rovná největší délce trénování z délek trénování na všech jednotkách. Vzorcem to zapíši následujícím způsobem

$$t_{iter} = max(t_{iter,ps}, t_{iter,1}, ..., t_{iter,N}), \tag{4.3}$$

kde $t_{iter,ps}$ je délka trénování na serveru s parametry, $t_{iter,i}$ je délka trénování na $i$té pracovní jednotce.

Trénování na pracovní jednotce je možné rozdělit na tři části: přijmutí nejnovějších hodnot vah od serveru s parametry, výpočet gradientů vah sítě a odeslání gradientů na server. Dobu potřebnou pro aktualizaci vah zanedbávám. Tyto činnosti je nutné provést jednu za druhou. Není možné vypočíst gradient bez přijetí nejnovějších hodnot vah sítě nebo odeslat gradient před jeho výpočtem. Trénování na serveru s parametry je možné rozdělit na tři části: odeslání nejnovějších hodnot vah pracovní jednotce, čekání na dokončení výpočtu gradientů vah sítě a přijmutí gradientů. Toto se provádí pro každou pracovní jednotku. Zde ale nastávají dva problémy. Server i pracovní jednotky používají duplexní spojení, to znamená, že server může odesílat první pracovní jednotce nejnovější hodnoty vah a zároveň přijímat vypočtené gradienty od druhé pracovní jednotky a také čekat na dokončení výpočtu na třetí pracovní jednotce. Je tedy vhodné maximalizovat překrývání těchto činností. Druhým problémem je to, že server může odesílat váhy dvou nebo více pracovním jednotkám současně. Pokud to dělá, tak je rychlost komunikace snížená a její délka zvýšená. Proto jsem se rozhodl, že budu uvažovat dva případy: nejlepší a nejhorší rozložení komunikace a výpočtu.

V nejlepším případě je ideální rozložení komunikace a výpočtů. To znamená, že server vždy posílá váhy pouze jedné výpočetní jednotce a přijímá gradienty od jedné pracovní jednotky. To zaručí, že komunikace s jednou pracovní jednotkou bude nejrychlejší a nejkratší možná. Cílem je snížit dobu, po kterou pracovní jednotka komunikuje se serverem, a zajistit, aby výpočet gradientů začal co nejdříve a překrýval se s komunikací serveru a jiné pracovní jednotky. Pro lepší pochopení je tento případ ilustrován na sekvenčním diagramu 4.1. Délku tohoto případu trénování na pracovní jednotce je tak možné zapsat jako

$$t_{iter,N}^{BEST} = (N-1)t_{comm} + t_{comm} + t_{grad} + t_{comm} = (N+1)t_{comm} + t_{grad}. \tag{4.4}$$

Tento vzorec jsem odvodil podle doby trénování pracovní jednotky, která si jako poslední stáhne váhy ze serveru. Tato jednotka bude čekat než si předchozích $N-1$ pracovních jednotek stáhne váhy a následně si je stáhne sama, provede výpočet a odešle gradienty. Předchozí jednotky dokončí své trénování před touto jednotkou, a proto nejsou pro výpočet celkové délky trénování relevantní. Na serveru s parametry má trénování délku

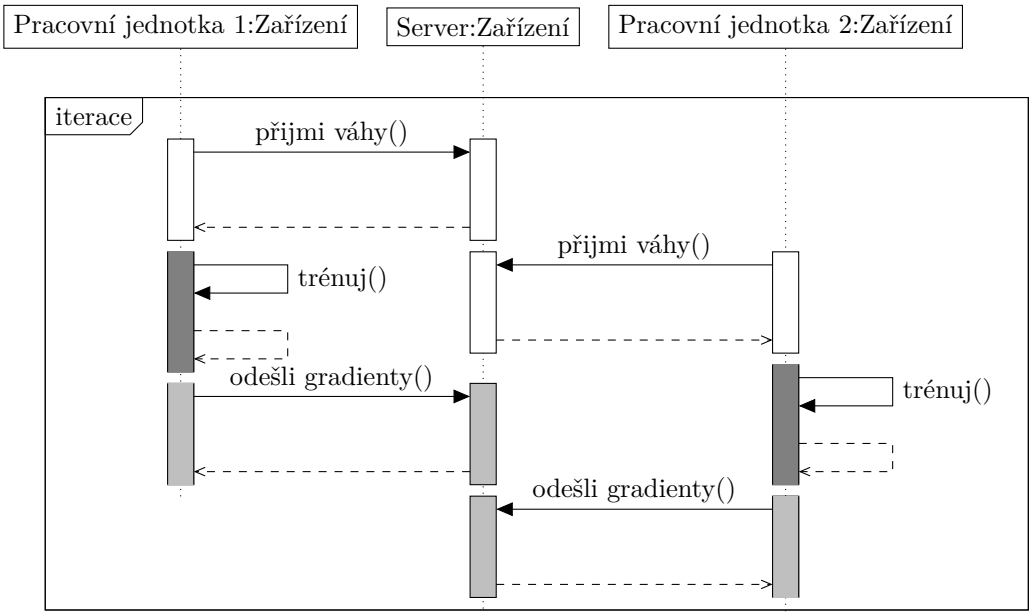
$$t_{iter,ps}^{BEST} = Nt_{comm} + t_{grad} + t_{comm} = (N+1)t_{comm} + t_{grad}. \tag{4.5}$$

Tento vzorec jsem odvodil tak, že server musí odeslat váhy N jednotkám a následně čekat, než poslední jednotka, které poslal váhy, dokončí výpočet a odešle gradienty. Výpočty a příjmy gradientů od ostatních jednotek provede server paralelně s odesíláním vah a výpočtem gradientů na poslední jednotce. Ze vzorců 4.3, 4.4 a 4.5 vyplývá, že

$$t_{iter}^{BEST} = t_{iter,ps}^{BEST} = t_{iter,N}^{BEST}. \tag{4.6}$$

V nejhorším případě je paralelní rozložení komunikace a sekvenční rozložení výpočtů. To znamená, že server vždy posílá váhy všem výpočetním jednotkám zároveň a také zároveň přijímá gradienty od všech pracovních jednotkek. To způsobí, že komunikace s jednou pracovní jednotkou bude nejpomalejší a nejdelší možná. Důsledkem toho je, že nedojde k využití duplexních vlastností propojení serveru a pracovních jednotek, a že server musí čekat na dokončení výpočtu gradientů. Předpokládám, že rychlost komunikace je nepřímo úměrná počtu komunikujících jednotek a délka přenosu je tak přímo úměrná počtu těchto jednotek. Sekvenční diagram 4.2 ilustruje tento případ. Délku tohoto případů trénování na jakékoliv pracovní jednotce je tak možné zapsat jako

$$t_{iter,i}^{WORST} = Nt_{comm} + t_{grad} + Nt_{comm} = 2Nt_{comm} + t_{grad}. \tag{4.7}$$



Obrázek 4.1: Sekvenční diagram jedné iterace distribuovaného synchronního trénování s centrálním serverem za předpokladu nejlepšího rozložení komunikace.

Tento vzorec jsem odvodil z toho, že N pracovních jednotek přijímá od serveru nejnovější hodnoty vah sítě. Z toho plyne, že komunikace se serverem bude Nkrát pomalejší a Nkrát delší. Odeslání gradientů po jejich výpočtu se provádí obdobným způsobem. Délku tohoto případů trénování na serveru s parametry je tak možné zapsat jako

$$t_{iter,ps}^{WORST} = Nt_{comm} + t_{grad} + Nt_{comm} = 2Nt_{comm} + t_{grad}. \tag{4.8}$$

Odvození tohoto vzorce je založené na tom, že server posílá N pracovním jednotkám nejnovější hodnoty vah sítě. Následně čeká na dokončení výpočtu na pracovních jednotkách. Příjem gradientů server provede podobným způsobem. Ze vzorců 4.3, 4.7 a 4.8 vyplývá, že

$$t_{iter}^{WORST} = t_{iter,ps}^{WORST} = t_{iter,N}^{WORST}. \tag{4.9}$$

Pro reálnou hodnotu délky iterace výpočtu t_{iter} musí platit

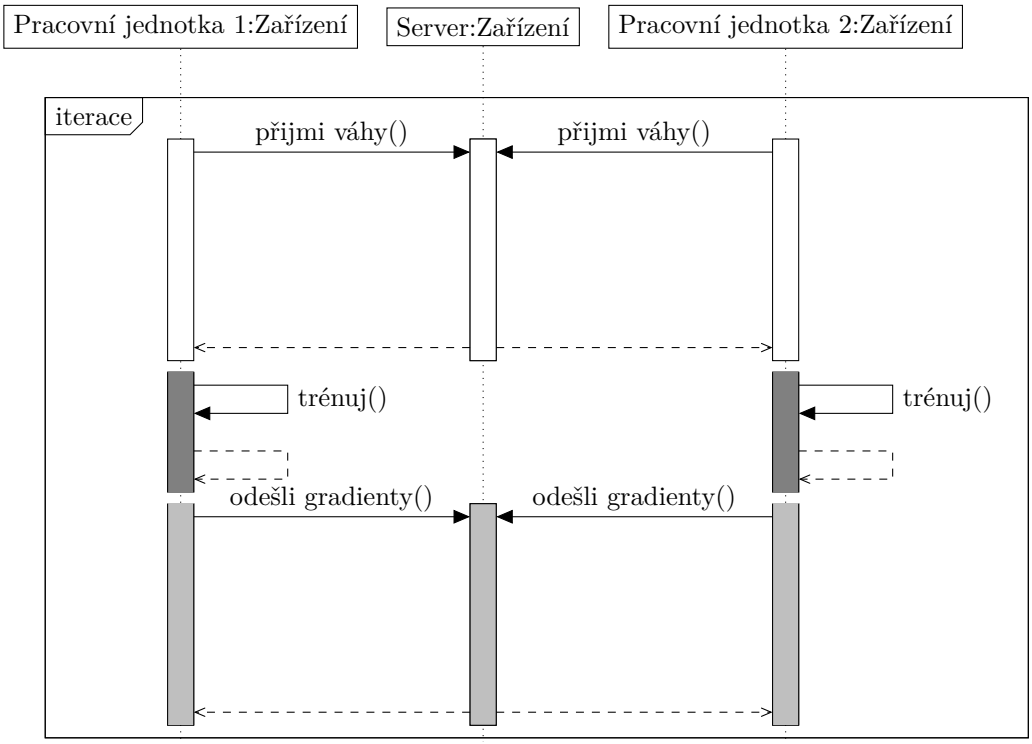
$$t_{iter} \in \langle t_{iter}^{BEST}, t_{iter}^{WORST} \rangle. \tag{4.10}$$

Pro výpočet odhadu zrychlení tak použiji průměr mezi nejlepší a nejhorší délkou iterace

$$t_{iter} = \frac{t_{iter}^{BEST} + t_{iter}^{WORST}}{2}. \tag{4.11}$$

4.1.2 Pomocí distribuovaného serveru

Hlavní vlastností trénování, které používá distribuovaný server je to, že neobsahuje centrální server, který by zpomalovat komunikaci. To znamená, že hodnoty vah sítě jsou rozděleny na několik částí, které jsou rozmístěny na výpočetních jednotkách. Výpočetní jednotky tak můžou stahovat nejnovější hodnoty vah umístěných na ostatních jednotkách a zároveň odesílat své váhy ostatním jednotkám. To je velmi dobré využití duplexních vlastností



Obrázek 4.2: Sekvenční diagram jedné iterace distribuovaného synchronního trénování s centrálním serverem za předpokladu nejhoršího rozložení komunikace.

pracovních jednotek. Průběh trénování je podobný jako při trénování pomocí centrálního serveru popsaného v podsekci 4.1.1.

Trénování na pracovní jednotce se skládá ze z pěti částí: příjem vah od ostatních jednotek, odeslání svých vah ostatním jednotkám, výpočet gradientů, odeslání svých gradientů ostatním jednotkám a příjem gradientů od ostatních jednotek. Dobu potřebnou pro aktualizaci vah zanedbávám. Příjem a odeslání hodnot vah je možné provést současně. To stejné platí pro příjem a odeslání gradientů. Před výpočtem, ale server musí přijmou váhy, protože jsou potřebné pro výpočet. Pro odeslání gradientů je potřeba je před tím vypočítat.

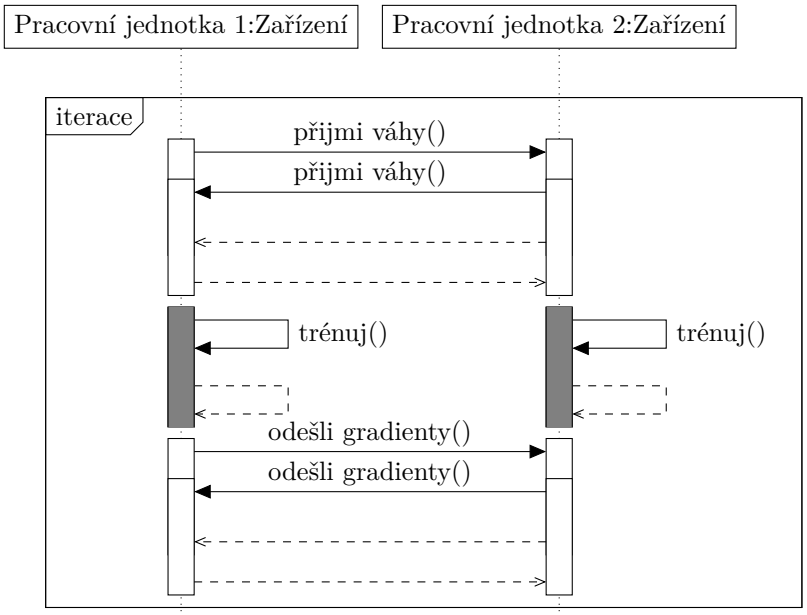
Ideální je situace, kdy pro N pracovních jednotek jsou váhy rozděleny na N stejně velkých částí, které jsou rozmístěné jedna na každé jednotce. Předpokládám, že rychlosti odesílání a přijímání jsou stejné na všech jednotkách. Při odesílaná vah jednotka odešle jednu Ntinu vah každé ze zbývajících $N-1$ jednotek. Při přijímaní vah jednotka přijme jednu Ntinu vah od každé ze zbývajících $N-1$ jednotek. Obě tyto komunikace trvají stejnou dobu a to $t_{comm}\frac{N-1}{N}$. Odesílání a přijímání gradientů probíhá analogickým způsobem. Potom délku tohoto trénování je možné zapsat jako

$$t_{iter} = 2t_{comm}\frac{N-1}{N} + t_{grad}. \tag{4.12}$$

Pro lepší pochopení je tento případ ilustrován na sekvenčním diagramu 4.3.

4.2 Asynchronní trénování

Asynchronní trénování bylo popsáno v podsekci 3.3.2. Hlavní vlastností tohoto způsobu trénování je to, že ho není možné rozdělit na iterace. To znamená, že musím provést odhad



Obrázek 4.3: Sekvenční diagram jedné iterace distribuovaného synchronního trénování s distribuovaným serverem.

doby celého trénování. Předpokládám, že D je počet dávek, které se mají natrénovat, a N je počet pracovních jednotek. Pokud každá jednotka pracuje stejnou rychlostí, stejně rychle odesílá a přijímá data a D je dělitelné N, pak každá jednotka zpracuje $\frac{D}{N}$ dávek při trénování.

4.2.1 Pomocí centrálního serveru

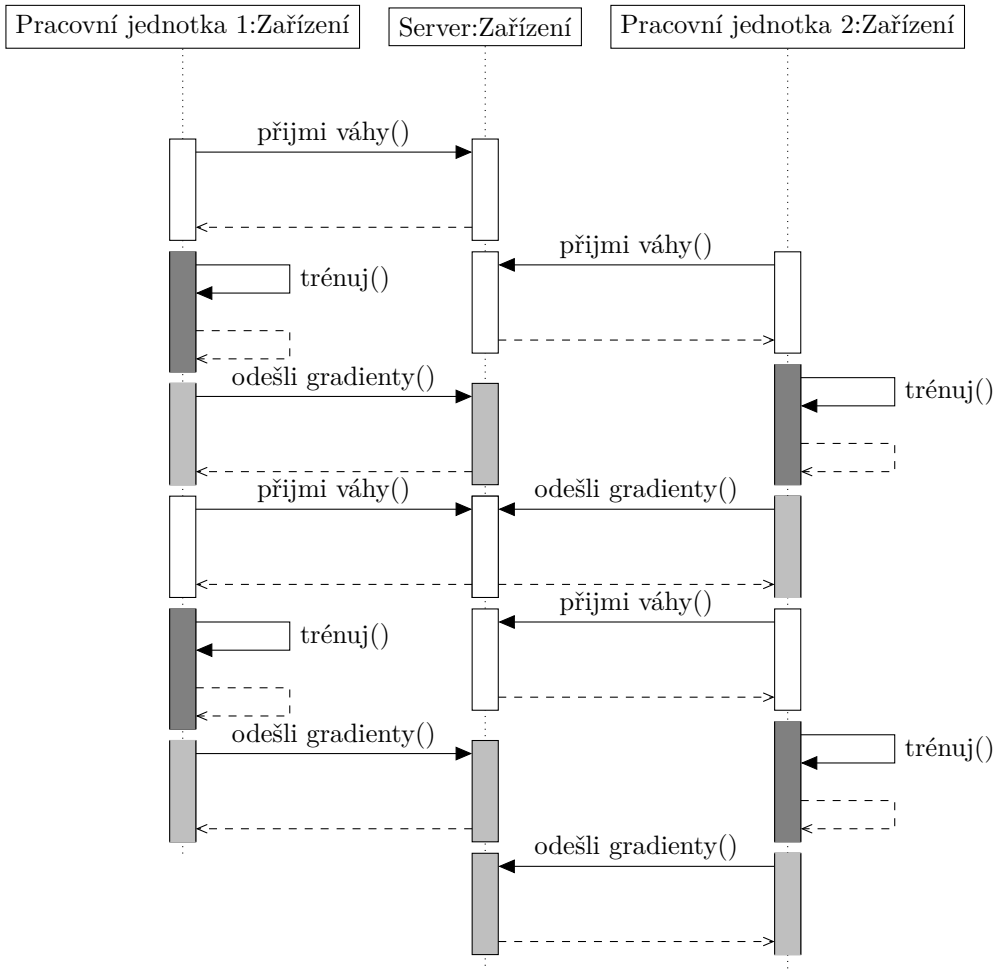
Centrální server je popsaný v podsekci 3.3.1. V podsekci 4.1.1 jsem popsal vlastnosti centrálního serveru vzhledem k odhadu doby trénování. Stejně jako při synchronním trénování budu uvažovat nejlepší a nejhorší rozložení komunikace mezi pracovními jednotkami a serverem s parametry.

V nejlepším případě je ideální rozložení komunikace a výpočtů. Tento případ je ilustrován na sekvenčním diagramu 4.4. Hlavním rozdílem oproti synchronnímu trénování je to, že výpočetní jednotky nemusí čekat na ostatní jednotky. Díky tomu může jednotka okamžitě po odeslání gradientů začít stahovat nejnovější hodnoty vah sítě. Délku tohoto případu trénování na pracovní jednotce je tak možné zapsat jako

$$t^{BEST}_{train,N} = (N-1)t_{comm} + \frac{D}{N}(2t_{comm} + t_{grad}). \tag{4.13}$$

Tento vzorec jsem odvodil podle doby trénování pracovní jednotky, která si jako poslední stáhne váhy ze serveru. Tato jednotka bude čekat než si předchozích $N-1$ pracovních jednotek stáhne váhy a následně $\frac{D}{N}$krát provede stažení nejnovějších hodnot vah serveru, výpočet gradientů a odeslání gradientů na server. Předchozí jednotky dokončí své trénování před touto jednotkou, a proto nejsou pro výpočet celkové délky trénování relevantní. Na serveru s parametry má trénování délku

$$t^{BEST}_{train,ps} = N\frac{D}{N}t_{comm} + t_{grad} + t_{comm} = (D+1)t_{comm} + t_{grad}. \tag{4.14}$$



Obrázek 4.4: Sekvenční diagram distribuovaného asynchronního trénování s centrálním serverem za předpokladu nejlepšího rozložení komunikace.

Tento vzorec jsem odvodil tak, že server musí $\frac{D}{N}$krát odeslat váhy N jednotkám a následně čekat, než poslední jednotka, které poslal váhy, dokončí výpočet a odešle gradienty. Výpočty a příjmy gradientů od ostatních jednotek provede server paralelně s odesíláním vah a výpočtem gradientů na poslední jednotce. Ze vzorců 4.3, 4.13 a 4.14 vyplývá, že

$$t_{train}^{BEST} = max(t_{train,ps}^{BEST}, t_{train,N}^{BEST}). \tag{4.15}$$

V nejhorším případě je paralelní rozložení komunikace a sekvenční rozložení výpočtů. To znamená, že server vždy posílá váhy všem N výpočetním jednotkám a také přijímá gradienty od všech N pracovních jednotek. To způsobí, že komunikace s jednou pracovní jednotkou bude Nkrát nejpomalejší a nejdelší možná. Sekvenční diagram 4.5 ilustruje tento případ. Délku tohoto případů trénování na jakékoliv pracovní jednotce je tak možné zapsat jako

$$t_{train,i}^{WORST} = \frac{D}{N}(Nt_{comm} + t_{grad} + Nt_{comm}) = \frac{D}{N}(2Nt_{comm} + t_{grad}). \tag{4.16}$$

Tento vzorec jsem odvodil z toho, že N pracovních jednotek přijímá od serveru nejnovější hodnoty vah sítě. Odeslání gradientů po jejich výpočtu se provádí obdobným způsobem. To

se provede $\frac{D}{N}$krát na každé jednotce. Délku tohoto případu trénování na serveru s parametry je tak možné zapsat jako

$$t_{train,ps}^{WORST} = \frac{D}{N}(Nt_{comm} + t_{grad} + Nt_{comm}) = \frac{D}{N}(2Nt_{comm} + t_{grad}). \tag{4.17}$$

Odvození tohoto vzorce je založené na tom, že server posílá N pracovním jednotkám nejnovější hodnoty vah sítě. Následně čeká na dokončení výpočtu na pracovních jednotkách. Příjem gradientů server provede podobným způsobem. To se opakuje $\frac{D}{N}$krát. Ze vzorců 4.3, 4.16 a 4.17 vyplývá, že

$$t_{train}^{WORST} = t_{train,ps}^{WORST} = t_{train,N}^{WORST}. \tag{4.18}$$

Pro reálnou hodnotu délky trénování t_{iter} musí platit

$$t_{train} \in \langle t_{train}^{BEST}, t_{train}^{WORST} \rangle. \tag{4.19}$$

Pro výpočet odhadu zrychlení tak použiji průměr mezi nejlepší a nejhorší délkou trénování

$$t_{train} = \frac{t_{train}^{BEST} + t_{train}^{WORST}}{2}. \tag{4.20}$$

4.2.2 Pomocí distribuovaného serveru

Distribuovaný server je popsaný v podsekci 3.3.3. V podsekci 4.1.2 jsem popsal vlastnosti distribuovaného serveru vzhledem k odhadu doby trénování.

Při asynchronním trénování také předpokládám, že pro N pracovních jednotek jsou váhy rozděleny na N stejně velkých částí, které jsou rozmístěné jedna na každé jednotce. Dále že rychlosti odesílání a přijímání jsou stejné na všech jednotkách. Při komunikaci jednotka odešle nebo přijme jednu Ntinu vah nebo gradientů. Obě tyto komunikace trvají stejnou dobu a to $t_{comm}\frac{N-1}{N}$. Potom je délku tohoto trénování možné zapsat jako

$$t_{train} = \frac{D}{N}(2t_{comm}\frac{N-1}{N} + t_{grad}). \tag{4.21}$$

Sekvenční diagram 4.6 zobrazuje takové trénování.

4.3 Zrychlení

Zrychlením při paralelních výpočtech se zabývá Peter Pacheco v knize [19]. Při paralelním trénování se rozděluje práce na N pracovních jednotek. V ideálním případě by doba trénování byla Nkrát menší. To se nazývá lineární zrychlení a obvykle program dosahuje horšího zrychlení, protože paralelní program vyžaduje přenosy dat a komunikaci mezi pracovními jednotkami. Zrychlení je definováno následujícím způsobem

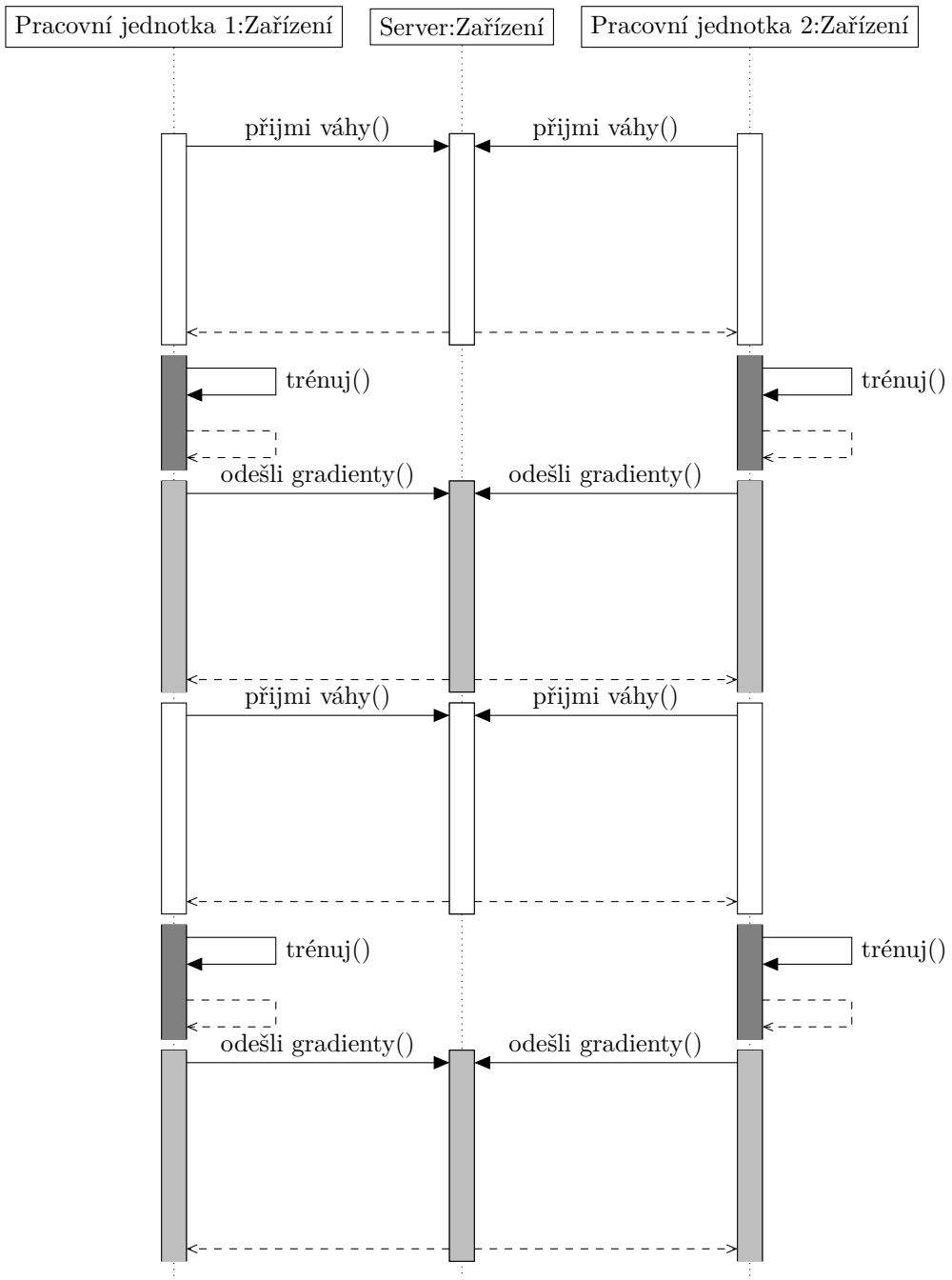
$$S = \frac{t_{serial}}{t_{parallel}}, \tag{4.22}$$

kde t_{serial} je délka výpočtu na jedné výpočetní jednotce a $t_{parallel}$ na N jednotkách. Pokud je zrychlení lineární pak $S = N$.

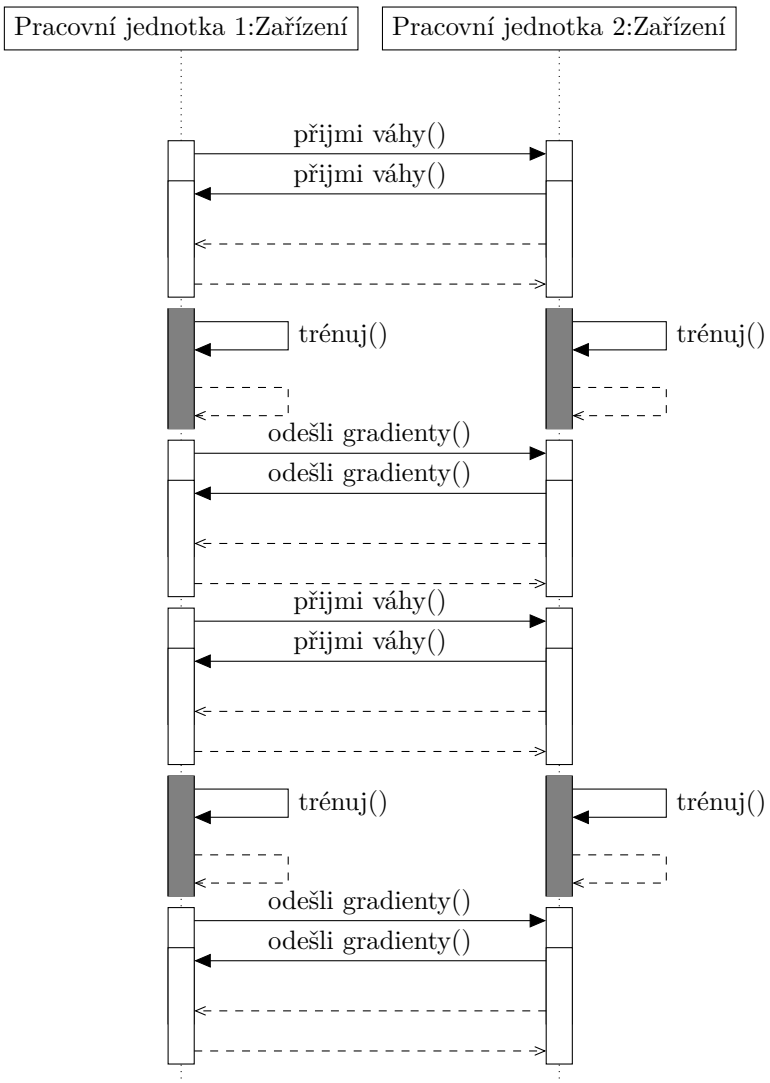
V mém případě budu používat vzorec

$$S = \frac{Dt_{grad}}{t_{parallel}}, \tag{4.23}$$

kde D je počet dávek, které se mají natrénovat, t_{grad} je délka výpočtu gradientů a t_{train} je délka trénování. Aktualizace vah zanedbávám v sériovém i paralelním trénování. Doba strávená přípravou výpočtu a ukončením výpočtu není důležitá a proto ji také zanedbávám.



Obrázek 4.5: Sekvenční diagram distribuovaného asynchronního trénování s centrálním serverem za předpokladu nejhoršího rozložení komunikace.



Obrázek 4.6: Sekvenční diagram distribuovaného asynchronního trénování s distribuovaným serverem.

Kapitola 5

Implementační prostředky

Tato kapitola slouží k seznámení s prostředky, které jsem použil při implementaci experimentů, které jsem provedl, abych ověřil nebo vyvrátil správnost výpočtů navržených v předchozí kapitole. První sekce je věnována knihovně Tensorflow, pomocí kfvteré jsem implementoval trénování sítí. Vysvětluji zde základní principy, na kterých je tato knihovna postavena. Následující část obsahuje popis prostření, ve kterém jsem provedl experimenty. To se skládá ze dvou částí: pracovní jednotky a trénovací sady.

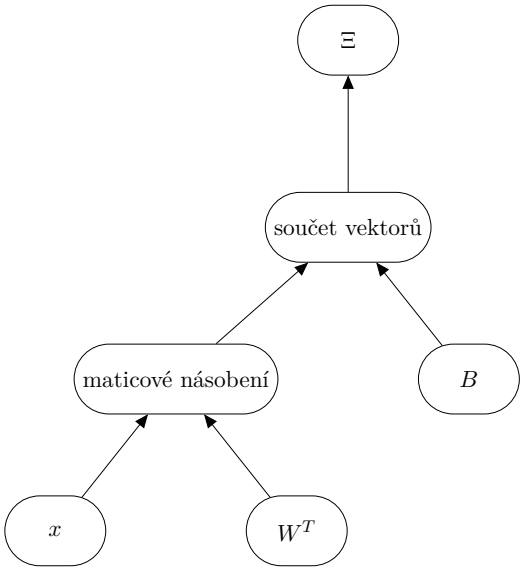
5.1 Tensorflow

Tensorflow je knihovna popsaná v práci [1] pro tvorbu programů, které využívají principů strojového učení, mezi které patří i neuronové sítě popsané v kapitole 2. Výpočty vyjádřené pomocí knihovny Tensorflow je možné spustit na mnoha různých zařízeních pouze s minimálními změnami jako například na mobilních telefonech, tabletech, osobních počítačích, ale i na distribuovaných systémech skládajících se z velkého množství propojených procesorů nebo grafických karet. Tato knihovna byla použita pro implementaci trénování neuronových sítí v mnoha oborech jako jsou rozpoznávání hlasu, počítačové vidění a robotika. Tato sekce se zbývá knihovnou Tensorflow.

5.1.1 Základní principy knihovny Tensorflow

Základní principy knihovny Tensorflow byly definovány pracovníky z výzkumného týmu Google Brain v práci [2]. Tensorflow byl vytvořen na základě dřívější knihovny DistBelief za účelem jejího nahrazení knihovnou, která je více flexibilní.

Pro tvorbu programů se používá abstrakce založená na grafech toku dat. To zvyšuje přenositelnost programů napsaných pomocí této knihovny. Graf je sestaven pomocí vysokoúrovňového skriptovacího jazyka, který umožňuje programátorům pracovat s různými architekturami a optimalizačními algoritmy bez potřeby modifikace podprogramu popisujícího nízkoúrovňovou funkcionalitu. Obě knihovny používají grafy toků dat pro reprezentaci svých modelů. Hlavním rozdílem mezi knihovnami Tensorflow a DistBelief je úroveň popisu, které používají. Zatímco DistBelief používá pouze několik komplexních vrstev, Tensorflow používá velké množství jednodušších vrstev implementující jednotlivé matematické operace (maticové násobením, konvoluce, ...) jako vrcholy v grafu toku dat. Tento přístup umožňuje programátorům definování vlastních vrstev pomocí vysokoúrovňového skriptovacího jazyka. Mnoho optimalizačních algoritmů potřebuje definované gradienty vrstev, tvoření vrstev z jednoduchých operátorů umožňuje automatický výpočet těchto gradientů. Vrcholy



Obrázek 5.1: Výpočetní graf znázorňující činnost jedné plně propojené vrstvy popsané rovnicí 2.2, kde W^T je transponovaná matice vah, X je vektor vstupů do vrstvy, B je bias a Ξ je výstup z vrstvy.

grafu můžou také reprezentovat měnitelný stav a operace na jeho změnu, to umožňuje experimentaci s různými trénovacími algoritmy.

Typický program používající knihovnu Tensorflow se skládá ze dvou částí: definice programu, neuronové sítě a pravidel pro její trénování jako grafu toků dat a interpretace tohoto grafu na jednom nebo více zařízeních. Protože interpretace grafu požaduje, aby byl graf kompletní a během ní se neměnil, může Tensorflow provést optimalizace založené na informacích o celém grafu. Například je možné rozložit zátěž na všechna jádra GPU pomocí toho, že graf obsahuje informace o tom, které operace si vzájemně předcházejí, a které je možné provést zároveň. Je tak možné přesně naplánovat a optimalizovat celý výpočet ještě před tím než začne.

5.1.2 Výpočetní graf

Výpočetní graf je reprezentací neuronové sítě a algoritmu jejího trénování. Je to orientovaný graf, ve kterém vrcholy reprezentují operace a hrany přenosy dat mezi těmito operacemi, směr hran určuje směr přenosu dat. Pokud je proměnná C výsledkem binární operace mezi proměnnými A a B, potom hrany vedou z A do operace, z B do operace a z operace do C. Na obrázku 5.1 je zobrazena jedna plně propojená vrstva sítě jako výpočetní graf. Následující odstavce popisují jednotlivé části výpočetního grafu.

Operace. Velkou výhodou reprezentace výpočtu jako výpočetního grafu je intuitivní a vizuální zobrazení vzájemných závislostí operací. V Tensorflow uzly grafu reprezentují operace, hrany vedoucí do nich vstupy těchto operací a hrany vedoucí z nich výstupy operací. Uzly v grafu reprezentují operace a celý graf tak reprezentuje transformaci vstupních dat na výstupní. Každá operace má libovolný počet vstupů a výstupů. Operace může reprezentovat matematickou operaci, konstantu, proměnnou, kontrolu toku, operaci s diskem nebo i komunikaci po síti. Konstanty je možné si definovat jako operace, které mají jeden výstup

a žádné vstupy, a které vždy vracejí na výstup stejnou hodnotu. Proměnné jsou analogicky operace, které mají jeden výstup a žádné vstupy, a které vracejí na výstup aktuální hodnotu proměnné. Každá operace musí mít odpovídající implementaci. Těmto implementacím se říká kernely. Každý kernel je implementací pouze pro jeden typ zařízení (procesor, grafická karta, ...).

Tenzory. V Tensorflow hrany grafu reprezentují toky dat mezi operacemi. Tenzor je vícerozměrná matice hodnot stejného typu s konstantní velikostí. Důležitou vlastností tenzoru je řád, který určuje počet rozměrů tenzoru. Tvar tenzoru je definován jako ntice, popisující počet složek tensoru v odpovídajícím rozměru. Velikost tvaru tenzoru se rovná řádu tenzoru. Z matematického pohledu je tenzor generalizací matice, vektoru a skaláru, který je tenzorem o řádu nula. Z informatického hlediska jsou tenzory vstupy a výstupy operací. Tenzory v Tensorflow jsou symbolické reprezentace dat, které neukládají žádná data a slouží pouze k jejich adresaci. Při tvorbě výpočetního grafu jsou tenzory výstupy operací jako například $A + B$. Výsledkem je tenzor, který je možné použít jako vstup do další operace. Takový tenzor se tak vlastně používá jako reprezentace spojení mezi dvěma operacemi (hrana ve výpočetním grafu). Speciálním typem tenzoru je řídký tenzor, který je možné použít pro snížení paměťové náročnosti v některých případech

Proměnné. Výpočetní graf je obvykle při trénování pomocí algoritmů z kapitoly 3 vícekrát interpretován. Mezi dvěma interpretacemi je většina tenzorů odstraněna. Je ale nutné uchovávat některé hodnoty mezi dvěma různými interpretacemi grafu: váhy a parametry sítě. Proto existují speciální operace, kterým se říká proměnné, a které slouží k uložení a nahrání perzistentních dat, které je možné použít k uložení dat tenzorů mezi dvěma různými interpretacemi grafu. Proměnné jsou definované typem dat a tvarem. Tensorflow používá několik operací, které slouží k práci s proměnnými. Při vytvoření proměnné je nutné zadat tenzor sloužící k inicializaci proměnné při interpretaci grafu. Tvar a typ dat je odvozen z tohoto tenzoru.

Sezení. Interpretace výpočetního grafu se v Tensorflow provádí pomocí prostředí, kterému se říká sezení. Sezení je zodpovědné za alokaci a správu prostředků pro interpretaci jako je například paměť pro uložení proměnných a dat tenzorů. Dále sezení poskytuje rutiny pro interpretaci samotnou. Tato metoda dostane na vstupu, které hodnoty uzlů má vypočítat. Dále se stará o nahrazení některých uzlů grafu reálnými hodnotami poskytnutými programátorem při jejím spuštění. Při invokaci Tensorflow začne od hodnot požadovaných uzlů a zpětně prochází graf a zkoumá závislosti a plánuje výpočet požadovaných hodnot. Výpočty jsou umísťovány na jednu nebo více pracovních jednotek.

Zařízení. Zařízení se starají o výpočetní část knihovny Tensorflow. Každá pracovní jednotka se může starat o jedno nebo více zařízení. Každé zařízení má typ a jméno. Jména zařízení se skládají z typu zařízení a indexu zařízení na této pracovní jednotce. V případě distribuovaného zpracování pak jméno zařízení také obsahuje i identifikaci výpočetní jednotky. Příkladem jmen jsou `/job:localhost/device:cpu:0`, které identifikuje první procesor na lokální výpočetní jednotce, nebo `/job:worker/task:8/device:gpu:1`, které identifikuje druhou grafickou kartu na pracovní jednotce 8. Každý objekt zařízení má na starosti správu paměti na odpovídajícím zařízení a provedení výpočtů na zařízení, které požadují jiné části knihovny. Pro daný výpočetní graf je hlavní činností knihovny Tensorflow umístit výpočty a proměnné na zařízení. Umístění výpočtů a proměnných je možné

Součástka	Typ součástky
Základní deska	MB GA-H97-D3H3i
Procesor	Intel Core i5-4460
Paměť	8 GB RAM
Grafická karta	NVidia GTX970 4GB
Disk	disk 250 GB
Ethernetové rozhraní	1Gbps duplexní

Tabulka 5.1: Součásti jedné výpočetní jednotky, na které byly provedeny experimenty.

nechat provést automaticky pomocí algoritmu, který rozděluje práci v závislosti na odhadu délky jejího trvání a závislostech ve výpočetním grafu. Manuální umístění se provádí pomocí funkce `device()`.

Distribuovaný výpočet. Poté co byly výpočty a proměnné umístěny na zařízení, je graf rozdělen na podgrafy, které odpovídají jednotlivým zařízením. Každá hrana grafu vedoucí z a na zařízení A do b na jiném zařízení B je nahrazena hranou vedoucí z a do nového uzlu *send* na zařízení A, hranou vedoucí z nového uzlu *recv* na zařízení B do b a hranou vedoucí z uzlu *send* do *recv*. Uzly *send* a *recv* jsou speciální uzly, které slouží k přenosu dat z jednoho zařízení na druhé. Obrázek 5.2 ilustruje výsledek takové úpravy provedené na výpočetním grafu jedné plně propojené vrstvy z obrázku 5.1. Toto izoluje všechny implementace přenosů dat pouze do dvou typů uzlů sítě. Výhodou tohoto přístupu je zjednodušeni knihovny. Poslední částí je optimalizace, kdy se všechny uzly typu *send* a *recv* na stejném zařízení sloučí do jednoho uzlu typu *send* a jednoho uzlu typu *recv*. Cílem této optimalizace je snížení paměťových nároků komunikace, protože sloučené uzly používají jeden vstupní nebo výstupní buffer. To je pochopitelně výhodnější než velké množství bufferů pro mnoho uzlů typu *send* a *recv*.

5.2 Trénovací prostředí

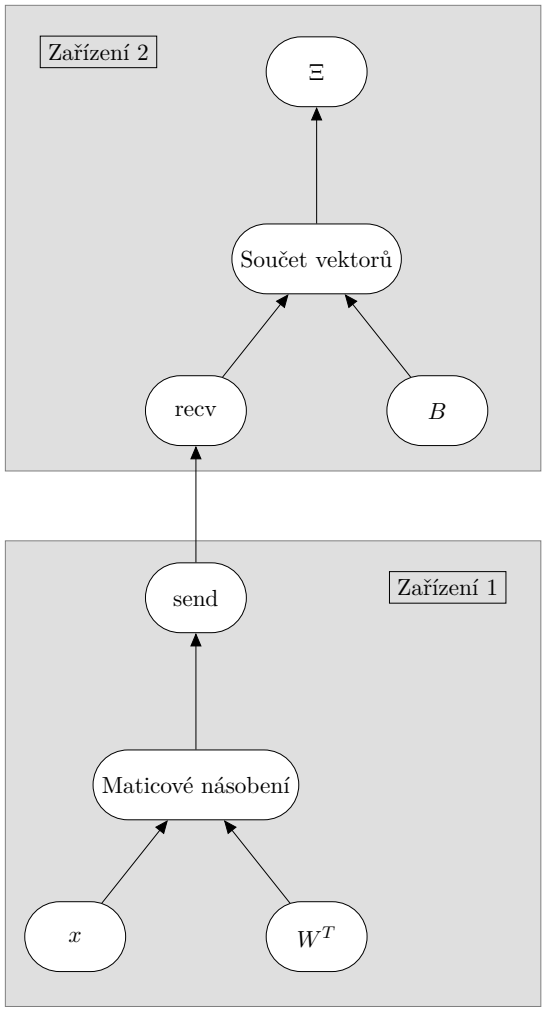
Tato sekce popisuje prostředí, ve kterém jsem provedl experimenty. Hlavní částí trénovacího prostředí jsou pracovní jednotky, které jsem použil v experimentech. Popis jedné jednotky je v první podsekci této sekce. Při trénování jsem použil trénovací množinu Caltech256, která je popsaná ve druhé podsekci této sekce.

5.2.1 Pracovní jednotka

Experimenty jsem provedl v učebně O204 Fakulty informačních technologií Vysokého učení technického v Brně. Podrobný popis složení jedné výpočetní jednotky je v tabulce 5.1. Na výpočetních jednotkách je nainstalován Linuxový operační systém CentOS 6. Při experimentech jsem použil Python ve verzi 3.6 a Tensorflow verze 1.1.

5.2.2 Trénovací množina

Griffin, Holub a Perona [7] vytvořili v roce 2007 trénovací množinu Caltech256. Tato trénovací množina byla zvolena pro použití v experimentech v této práci. Caltech256 je založen na předchozí trénovací množině Caltech101. Obě trénovací množiny obsahují různorodé obrázky (např.: zvířata, dopravní prostředky, nářadí atd.) rozdělené do tříd. Caltech256



Obrázek 5.2: Výpočetní graf znázorňující činnost jedné plně propojené vrstvy popsané rovnicí 2.2, kde W^T je transponovaná matice vah, X je vektor vstupů do vrstvy, B je bias a Ξ je výstup z vrstvy. Výpočet je rozdělen na dvě části, které se provádějí na dvou různých zařízeních.

obsahuje 29 780 obrázků rozdělených do 256 tříd, Caltech101 pouze 8677 obrázků rozdělených do 101 tříd. Dalším rozdílem je to, že obrázky v trénovací množině Caltech101 jsou uměle narotovány tak, aby objekt na všech obrázcích v dané třídě směřoval stejným směrem. V trénovací množině Caltech256 objekty nejsou narotovány. Obě trénovací množiny jsou zaměřeny na rozpoznávání jednotlivých objektů, které se nacházejí ve středu obrázku. Každá třída v Caltech256 obsahuje 80 nebo více obrázků, třída v Caltech101 obsahuje pouze 31 nebo více obrázků. Pro experimenty jsem vybral trénovací množinu Caltech256. Každý obrázek byl převeden na velikost 224×224 pixelů.

Kapitola 6

Experimenty

Důležitou částí této práce je ověření vzorců odvozených v kapitole 4 pomocí experimentů. Tato kapitola se zabývá těmito experimenty a jejich výsledky. V první sekci jsem popsal experimenty. Následující tři sekce této kapitoly se věnují třem způsobům trénování, které jsou popsané v sekci 6.1. V každé sekci je několik grafů. Každý graf zobrazuje závislost zrychlení trénování na počtu pracovních jednotek pro danou neuronovou síť a způsob distribuovaného trénování. Nové informace odvozené z každého grafu jsem popsal a zhodnotil.

6.1 Popis experimentů

Tato sekce sekce popisuje experimenty, které jsem provedl, abych ověřil jejich přesnost. Provedl jsem experimenty, které používají tři různé způsoby natrénování $D = 128$ dávek na N pracovních jednotkách.

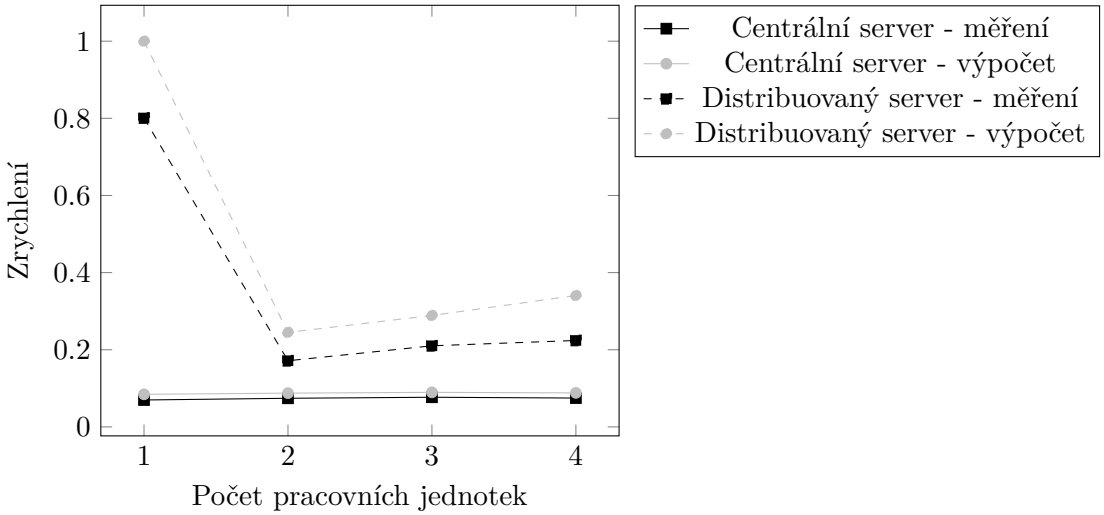
Synchronní trénování se spojením dávek. Při tomto způsobu trénování se na každé výpočetní jednotce provede výpočet gradientů jedné dávky z trénovací množiny. Tyto trénování proběhla synchronně a paralelně. Gradienty se pošlou na sever, který je aplikuje na váhy sítě. V jedné iteraci algoritmu se tak natrénuje N dávek. Po provedení $\frac{D}{N}$ iterací je natrénováno D dávek.

Synchronní trénování s rozdělením dávek. Tento způsob trénování na každé výpočetní jednotce provede výpočet gradientů jedné Ntiny dávky z trénovací množiny. Gradienty se následně odešlou na sever, který pomocí nich aktualizuje váhy sítě. V jedné iteraci algoritmu se tak natrénuje jedna dávka. Pro natrénování D dávek se tak musí provést D iterací algoritmu. Tento přístup dává stejné výsledky jako sekvenční trénování.

Asynchronní trénování. Poslední způsob trénování, se kterým jsem experimentoval, je asynchronní trénování, které na každé výpočetní jednotce vypočte gradienty jedné dávky z trénovací množiny. Gradienty se následně odešlou na sever, který pomocí nich provede výpočet nejnovějších vah sítě. Pro natrénování D dávek se tak musí provést $\frac{D}{N}$ trénování na každé pracovní jednotce.

Síť	Délka výpočtu (s)	Délka přenosu (s)
VGG16E	0,719	5,153
GoogLeNet	0,763	0,210
SqueezeNet	0,758	0,033
Resnet34	0,821	0,811

Tabulka 6.1: Naměřené střední délky výpočtu gradientů a přenosu pro neuronové sítě použité v této práci.



Obrázek 6.1: Zrychlení trénování sítě VGG16E při použití synchronního trénování se spojením dávek.

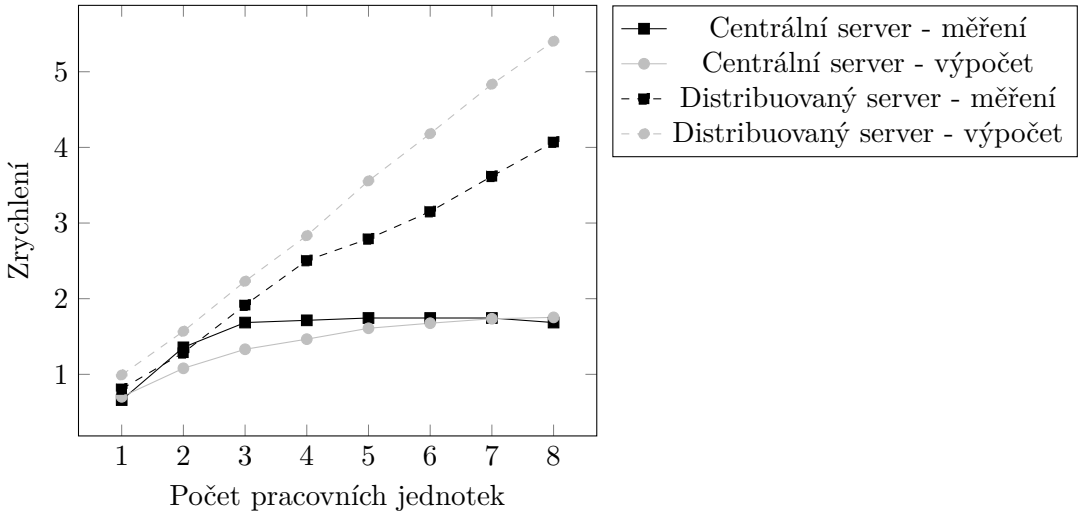
6.2 Sekvenční trénování

První částí experimentů je sekvenční trénování provedené na jedné pracovní jednotce. To slouží k určení hodnot t_{grad} a t_{comm}, které se použijí ve vzorcích z kapitoly 4. Výsledky těchto experimentů jsou zapsány v tabulce 6.1. Tabulka obsahuje naměřené střední hodnoty délky výpočtu a přenosu v sekundách.

6.3 Synchronní trénování se spojením dávek

Toto trénování jsem provedl na všech neuronových sítích popsaných v sekcích 2.4 až 2.7. Na grafu 6.1 je zrychlení trénování sítě VGG16E tímto způsobem na jedné až čtyřech pracovních jednotkách. Z tohoto grafu vyplývá, že tato síť není vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení nižšího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Pro distribuovaný server byl odhad méně přesný. Tento odhad ale správně říká, že nejlepším způsobem jak trénovat síť VGG16E je použití jedné pracovní jednotky.

Graf 6.2 ukazuje zrychlení trénování sítě GoogLeNet na jedné až osmi pracovních jednotkách. Z tohoto grafu vyplývá, že tato síť je vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení vyššího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Pro distribuovaný server byl odhad méně



Obrázek 6.2: Zrychlení trénování sítě GoogLeNet při použití synchronního trénování se spojením dávek.

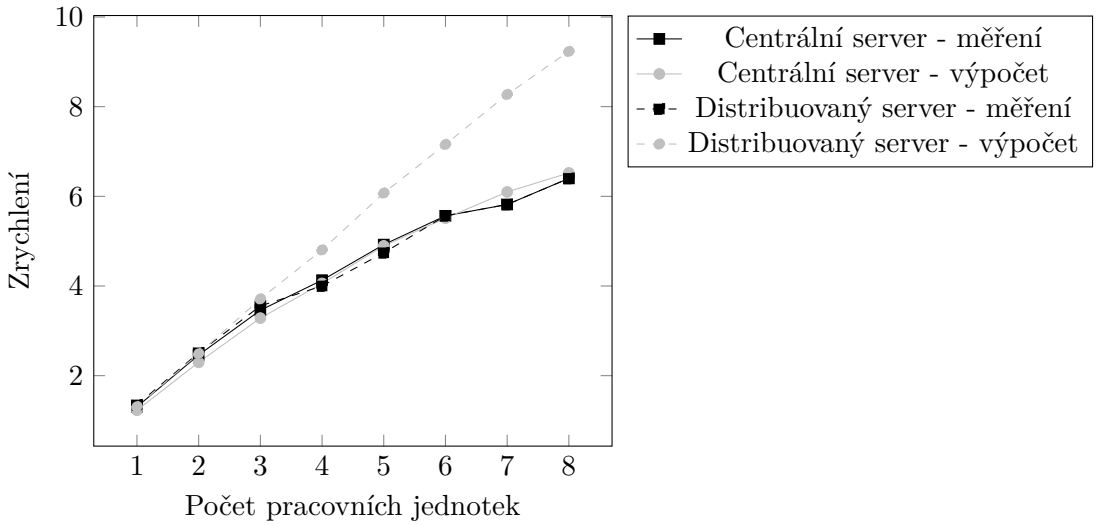
přesný, ale správně říkal, jak se zrychlení zachová se zvyšujícím se počtem pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak trénovat síť GoogLeNet je použití distribuovaného serveru a osmi pracovních jednotek.

Graf 6.3 zobrazuje zrychlení trénování sítě SqueezeNet na jedné až osmi pracovních jednotkách. Tento graf říká, že tato síť je velmi vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení vyššího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Pro distribuovaný server byl odhad méně přesný, ale správně říkal, jak se zrychlení zachová se zvyšujícím se počtem pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak trénovat síť SqueezeNet je použití osmi pracovních jednotek. Na typu použitého serveru nezáleží. To je způsobeno velmi malým počtem vah této sítě.

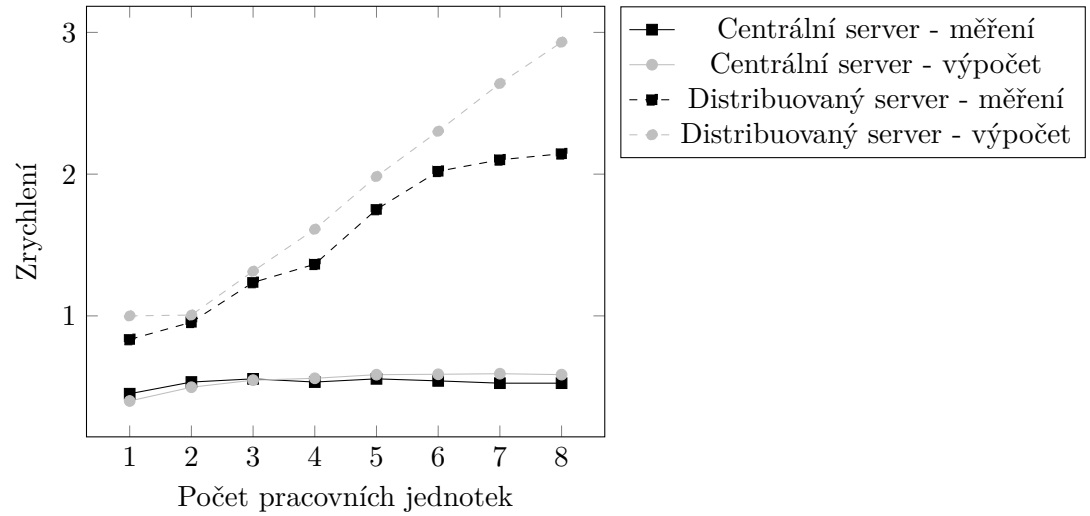
Zrychlení trénování sítě Resnet34 na jedné až osmi pracovních jednotkách zobrazuje graf 6.4. Z tohoto grafu vyplývá, že síť Resnet34 je vhodná pro tento způsob distribuovaného trénování, protože ve všech případech dosahuje zrychlení vyššího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Odhad zrychlení pro distribuovaný server byl méně přesný, ale správně předpověděl, jak se zachová při zvýšení počtu pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak trénovat síť Resnet34 je použití osmi pracovních jednotek a distribuovaného serveru.

6.4 Synchronní trénování s rozdělením dávek

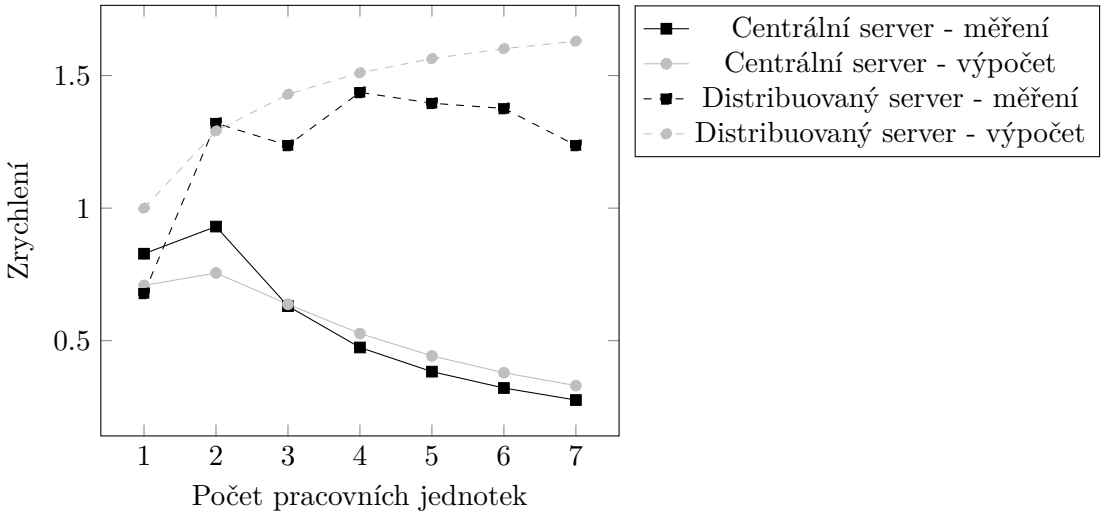
Tento způsob trénování jsem testoval na všech neuronových sítí kromě sítě VGG16E. Na grafu 6.5 je zakreslena závislost zrychlení trénování sítě GoogLeNet na počtu pracovních jednotek. Síť GoogLeNet je není vhodná pro trénování tímto způsobem, pokud je použit centrální server. Při použití distribuovaného serveru je výhodnost tohoto způsobu trénování vyšší. Přesnost odhadu pro centrální server je poměrně velká. Pro distribuovaný server je odhad velmi nepřesný a dokonce říká, že se trénování zrychlí, přestože měření tvrdí opak. Nejlepším způsobem jak trénovat



Obrázek 6.3: Zrychlení trénování sítě SqueezeNet při použití synchronního trénování se spojením dávek.



Obrázek 6.4: Zrychlení trénování sítě Resnet34 při použití synchronního trénování se spojením dávek.



Obrázek 6.5: Zrychlení trénování sítě GoogLeNet při použití synchronního trénování s rozdělením dávek.

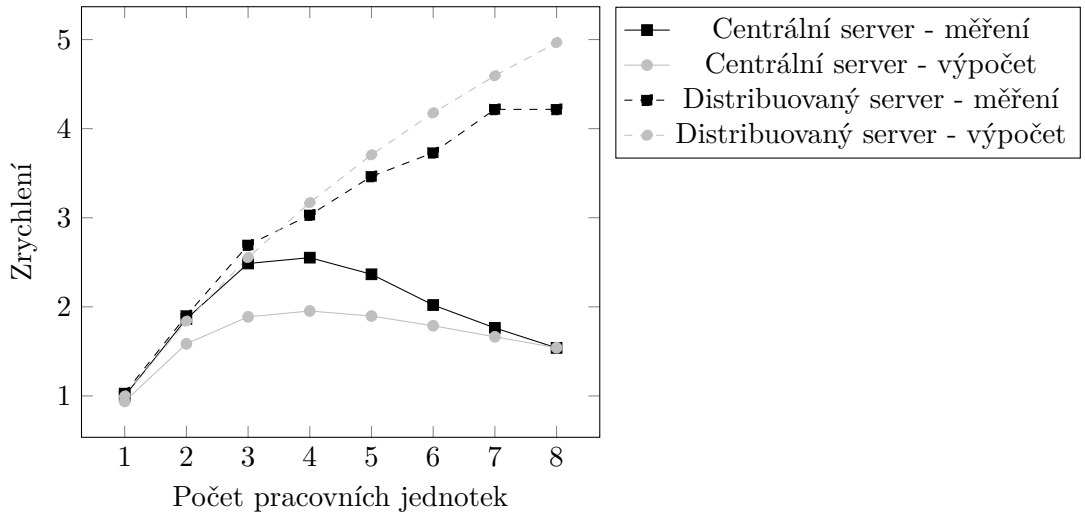
Graf 6.6 ilustruje závislost zrychlení trénování sítě SqueezeNet na počtu pracovních jednotek. Síť SqueezeNet je velmi vhodná pro trénování tímto způsobem pomocí obou typů serveru. Při použití distribuovaného serveru je výhodnost tohoto způsobu trénování vyšší. Pro oba typy serverů je odhad přesný a dokonce správně určuje, že při použití centrálního serveru je nejrychlejší výpočet na čtyřech pracovních jednotkách. Nejlepším způsobem jak trénovat je použití distribuovaného serveru a osmi pracovních jednotek.

Na grafu 6.7 je zanesena závislost zrychlení trénování sítě Resnet34 na počtu pracovních jednotek. Síť Resnet34 není vhodná pro trénování tímto způsobem pomocí obou typů serveru, protože zrychlení je vždy menší než 1. Při použití distribuovaného serveru je zrychlení tohoto způsobu trénování vyšší, ale je stále menší než 1. Pro centrální server je odhad velmi přesný. Pro distribuovaný server je odhad méně přesný, ale stále správně předpovídá, jak se velikost zrychlení zachová při zvyšujícím se počtu pracovních jednotek. Nejlepším způsobem jak trénovat je sekvenční trénování na jedné pracovní jednotce.

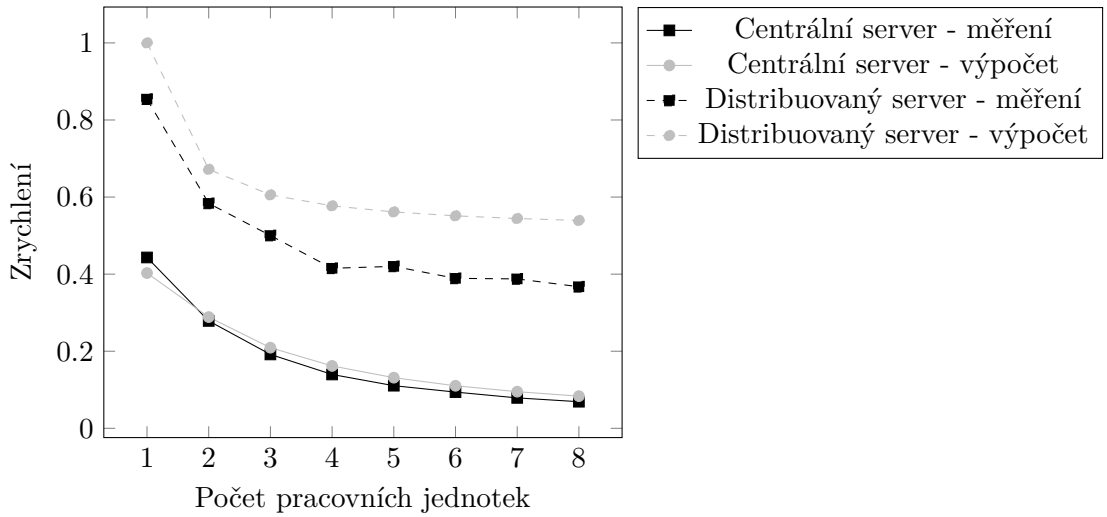
6.5 Asynchronní trénování

Asynchronní trénování jsem provedl na všech neuronových sítích popsaných v sekcích 2.4 až 2.7. Na grafu 6.8 je zrychlení trénování sítě VGG16E tímto způsobem na jedné až čtyřech pracovních jednotkách. Z tohoto grafu vyplývá, že tato síť není vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení nižšího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Pro distribuovaný server byl odhad méně přesný. Tento odhad ale správně říká, že nejlepším způsobem jak trénovat síť VGG16E je použití jedné pracovní jednotky a sekvenčního trénování.

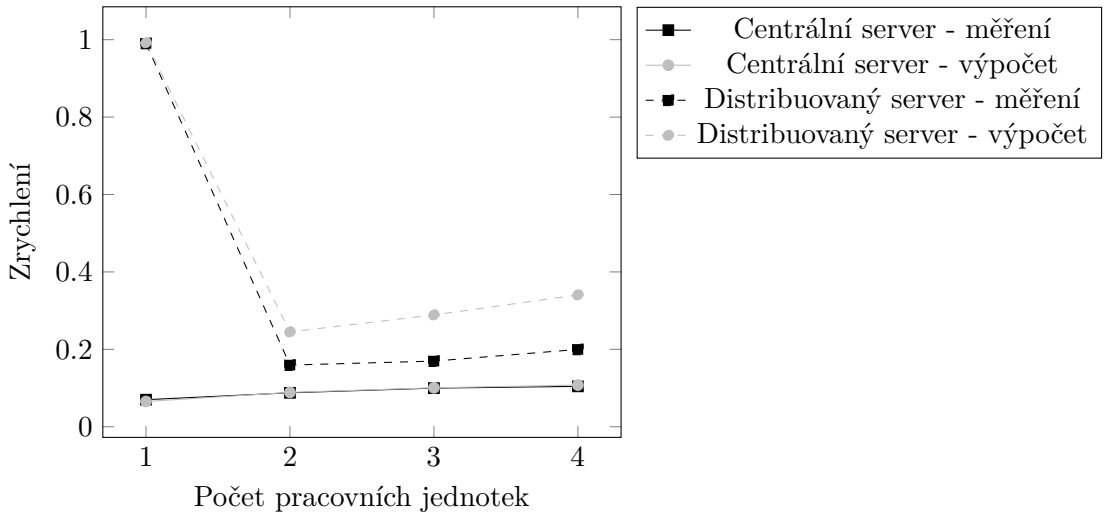
Graf 6.9 ukazuje zrychlení trénování sítě GoogLeNet na jedné až osmi pracovních jednotkách. Z tohoto grafu vyplývá, že tato síť je vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení vyššího než 1. Pro oba typy serverů byl odhad méně přesný, ale správně říkal, jak se zrychlení zachová se zvyšujícím se počtem pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak asynchronně trénovat síť GoogLeNet je použití distribuovaného serveru a osmi pracovních jednotek.



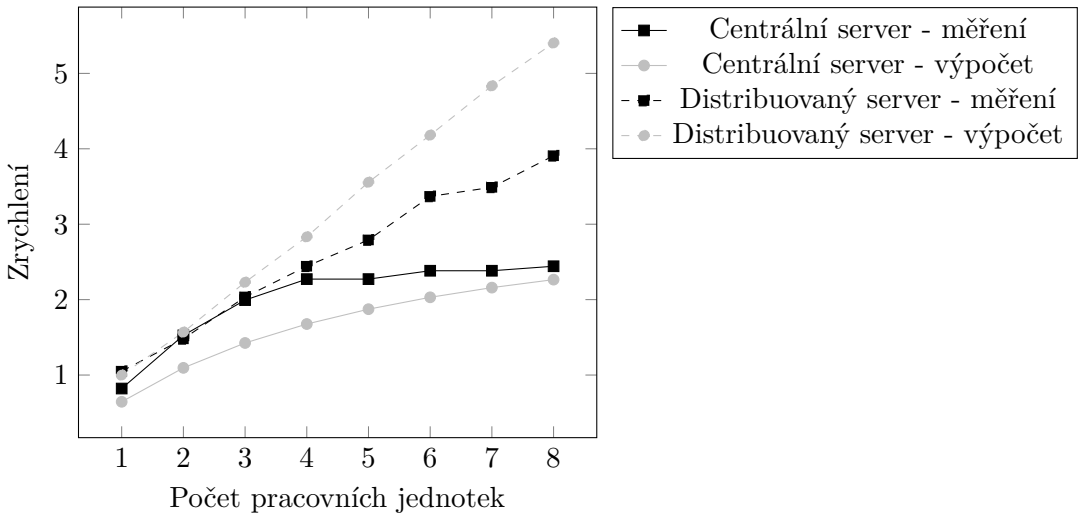
Obrázek 6.6: Zrychlení trénování sítě SqueezeNet při použití synchronního trénování s rozdělením dávek.



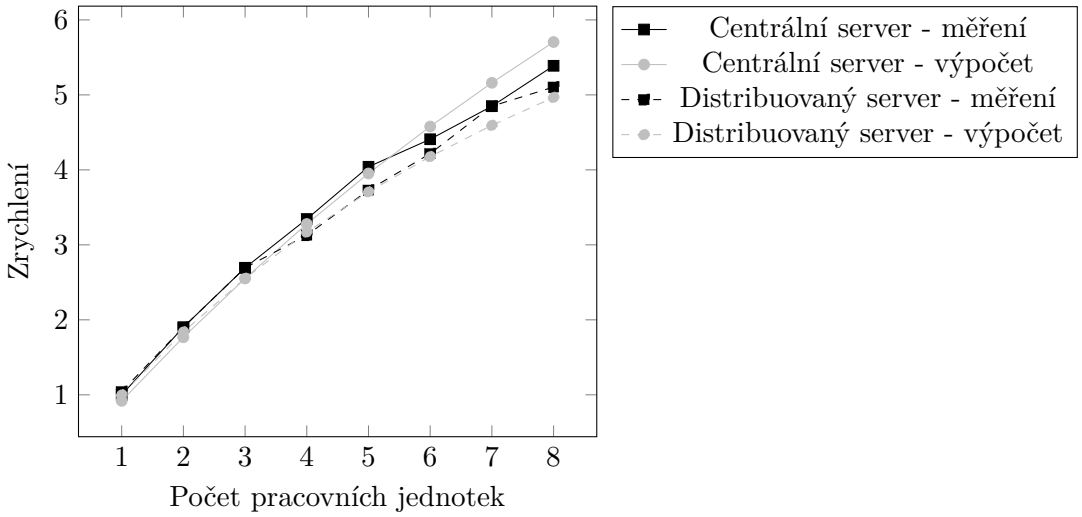
Obrázek 6.7: Zrychlení trénování sítě Resnet34 při použití synchronního trénování s rozdělením dávek.



Obrázek 6.8: Zrychlení trénování sítě VGG16E při použití asynchronního trénování.



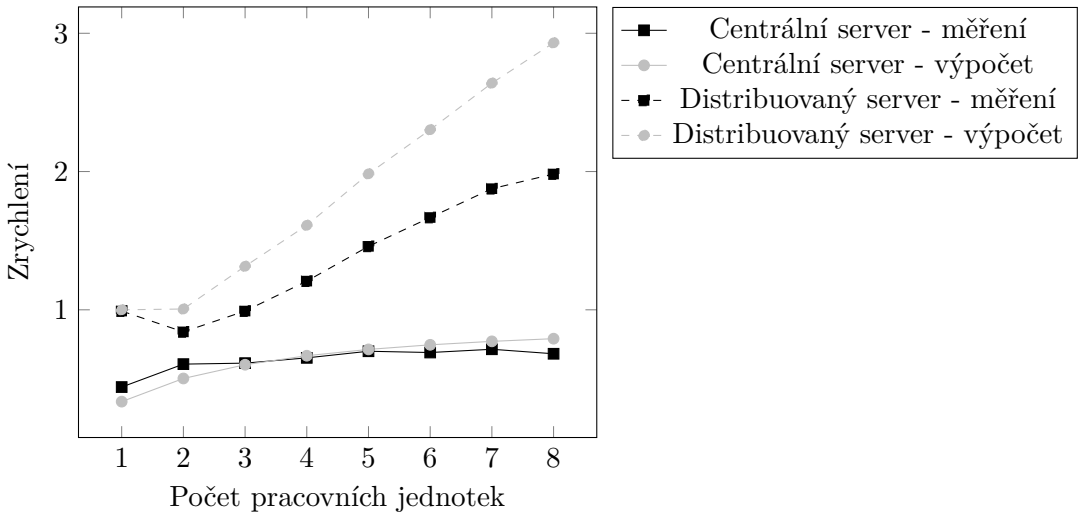
Obrázek 6.9: Zrychlení trénování sítě GoogLeNet při použití asynchronního trénování.



Obrázek 6.10: Zrychlení trénování sítě SqueezeNet při použití asynchronního trénování.

Graf 6.10 zobrazuje zrychlení trénování sítě SqueezeNet na jedné až osmi pracovních jednotkách. Tento graf říká, že tato síť je velmi vhodná pro distribuované trénování, protože ve všech případech dosahuje zrychlení vyššího než 1. Pro centrální server bylo toto přesně předpovězeno pomocí výpočtu odhadu zrychlení. Pro distribuovaný server byl odhad méně přesný, ale správně říkal, jak se zrychlení zachová se zvyšujícím se počtem pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak asynchronně trénovat síť SqueezeNet je použití osmi pracovních jednotek. Na typu použitého serveru nezáleží.

Zrychlení trénování sítě Resnet34 na jedné až osmi pracovních jednotkách zobrazuje graf 6.11. Z tohoto grafu vyplývá, že síť Resnet34 je vhodná pro tento způsob distribuovaného trénování, pouze pokud se použije distribuovaný server. Pro trénování pomocí centrálního serveru tato síť není vhodná, protože zrychlení trénování je nižší než 1. To bylo přesně předpovězeno pomocí výpočtu. Odhad zrychlení pro distribuovaný server je méně přesný, ale správně předpověděl, jak se zachová při zvýšení počtu pracovních jednotek. Tyto odhady správně určily, že nejlepším způsobem jak asynchronně trénovat síť Resnet34 je použití osmi pracovních jednotek a distribuovaného serveru.



Obrázek 6.11: Zrychlení trénování sítě Resnet34 při použití asynchronního trénování.

Kapitola 7

Diskuze

V této kapitole shrnu výsledky, které jsem prezentoval v předchozí kapitole. První polovinu této kapitoly zabírá zhodnocení výsledků experimentů. Druhá polovina je věnována návrhu mechanizmu, který by měl zrychlit paralelní trénování pomocí optimalizace komunikace se serverem.

7.1 Zhodnocení výsledků

Při sekvenčním trénování jsem naměřil délky výpočtu jedné dávky a přenosu vah od serveru k pracovní jednotce. Délky výpočtu gradientů všech čtyř sítí jsou si podobné, i když se sítě liší ve struktuře a počtu vah. To považuji za velmi výhodné, protože délky přenosu tak budou mít hlavní vliv na celkovou délku trénování. Délky přenosu vah sítí se výrazně liší v závislosti na jejich počtu. SqueezeNet je síť s nejmenším počtem vah a délka přenosu byla nejkratší. Délka přenosu vah sítě GoogLeNet je přibližně sedmkrát delší. Váhy sítě Resnet34 se přenesou za 27krát delší dobu. Pro síť VGG16E přenos vah trval 170krát déle než pro síť SqueezeNet. Délky přenosu vah odpovídají počtu parametrů sítě.

Síť VGG16E je velmi nevhodná pro paralelní trénování, protože používá velké množství vah. Jejich přenosy tak velmi výrazně zpomalovaly trénování sítě. Výsledkem je to, že nejlepším způsobem jak trénovat tuto síť je sekvenční trénování na jedné pracovní jednotce.

Resnet34 také není vhodná pro paralelní trénování, protože používá velké množství vah. Nejlepšího výsledku jsem dosáhl, když jsem Resnet34 synchronně trénoval na osmi pracovních jednotkách pomocí distribuovaného serveru. V tom případě bylo zrychlení 2,1. To je velmi malé zrychlení na to, že jsem použil osm pracovních jednotek.

Síť GoogLeNet je vhodná pouze v některých případech. Při asynchronním trénování je dosaženo zrychlení 3,9 a při použití synchronního trénování s rozdělením dávek zrychlení 4. V obou případech musí být použit distribuovaný server, jinak je zrychlení výrazně horší.

SqueezeNet je síť, která byla navržena, aby měla velmi malý počet vah. To je velmi výhodné pro její paralelní trénování a projevuje se to velmi výrazným zrychlením při trénování na více výpočetních jednotkách. Nejlepší výsledek je zrychlení 6,5, kterého bylo dosaženo při synchronním trénování se spojením dávek na osmi pracovních jednotkách pomocí distribuovaného serveru.

Ve většině případů bylo nejlepším způsobem trénování použití osmi výpočetních jednotek a distribuovaného serveru nebo sekvenční trénování na jedné jednotce. Zajímavým případem je trénování sítě SqueezeNet při použití synchronního trénování s rozdělením dávek pomocí centrálního serveru. Toto trénování je popsané grafem 6.6. V tomto případě je

trénování nejrychlejší při použití čtyř výpočetních jednotek. Výpočet odhadů tento nezvyklý případ předpověděl.

Zrychlení při použití distribuovaného serveru bylo většinou větší než při použití centrálního serveru. Jediným případem, kdy bylo přibližně stejně velké, je při asynchronním a synchronním trénování se spojením dávek sítě SqueezeNet. To bylo způsobeno velmi malým počtem vah této sítě. Kvůli tomu je vliv zrychlené komunikace na celkovou délku výpočtu velmi malý. Ve všech ostatních případech použití distribuovaného serveru zvýšilo zrychlení trénování. To je způsobeno velmi malým počtem vah této sítě.

Krajním případem bylo distribuované trénování na jedné pracovní jednotce. Při použití centrálního serveru bylo trénování vždy pomalejší než sekvenční. To je tím, že se vlastně jedná o sekvenční trénování, které před a po každém výpočtu gradientů provádí přenos dat z a na server. Toto trénování bude vždy pomalejší než sekvenční trénování na jedné jednotce. To se potvrdilo při výpočtu i při experimentování. Při použití distribuovaného serveru bylo trénování mírně pomalejší než sekvenční. To je způsobeno dodatečnými přenosy vah a gradientů v rámci jedné jednotky, mezi serverovou a výpočetní částí jednotky.

7.2 Návrh na další práci

V kapitole 4 jsem při trénování s centrálním serverem popsal případ nejlepšího rozložení komunikace a výpočtů. Tento případ využíval toho, že k serveru přistupovala pouze jedna pracovní jednotka. To způsobilo, že komunikace probíhala na nejvyšší možné rychlosti. To je ideální případ, který málokdy nastane. Existuje ale způsob jak zaručit, že tento případ nastane pokaždé, když se bude k serveru přistupovat. Je totiž možné zaručit exkluzivní přístup k serveru pomocí semaforu.

Navrhuji tedy vytvoření systému, který bude řídit přístup k serveru. Tento systém bude založen na semaforu definovaném Edsgerem W. Dijkstrou v práci [4]. Na serveru bude umístěn semafor s frontou, který omezuje přenos dat se serverem pouze na jednu pracovní jednotku. Ostatní jednotky musí čekat ve frontě, pokud chtějí přístup k serveru. Velmi krátké přenosy dat (ve srovnání s přenosem vah a gradientů) tomuto semaforu nepodléhají pro snížení režie. Pro server s duplexním spojením mezi ním a pracovními jednotkami je potřeba vytvořit dva semafory: jeden pro přenos dat směrem na server a druhý pro přenos dat ze serveru.

Pokud se serverem komunikuje všech N pracovních jednotek zároveň, pak je délka této komunikace $t_{par} = Nt_{comm}$. Pokud pracovní jednotky přistupují k serveru jedna po druhé, pak je délka přenosu dat vždy t_{comm}. V nejlepším případě je fronta semaforu volná, potom je délka komunikace

$$t_{sem}^{BEST} = 0t_{comm} + t_{comm} = t_{comm}. \tag{7.1}$$

V tomto případě nemusí jednotka čekat ve frontě a může okamžitě začít komunikovat. V nejhorším případě je ve frontě $N - 1$ pracovních jednotek, potom je délka komunikace

$$t_{sem}^{WORST} = (N - 1)t_{comm} + t_{comm} = Nt_{comm} = t_{par}. \tag{7.2}$$

To znamená, že jednotka musí čekat, než předchozích $N - 1$ procesů dokončí své přenosy dat. Pro reálnou délku komunikace omezenou semaforem musí platit

$$t_{sem} \in \langle t_{sem}^{BEST}, t_{sem}^{WORST} \rangle. \tag{7.3}$$

Z toho vyplývá, že $t_{sem} \leq t_{par}$.

Tento mechanizmus je možné zkombinovat s distribuovaným serverem.

Kapitola 8

Závěr

Má práce se zabývá paralelním trénováním neuronových sítí. Její hlavní částí je návrh vzorců, které je možné použít na odhad délky trénování sítě na základě délky trénování jedné dávky a doby přenosu vah ze serveru na pracovní jednotku. Z délky trénování je možné vypočítat zrychlení trénování na daném počtu pracovních jednotek. Pro ověření přesnosti těchto vzorců jsem provedl experimenty, které testovali přesnost odhadu zrychlení trénování čtyř neuronových sítí třemi různými způsoby pomocí dvou typů serveru s parametry. Tyto experimenty ukázali, že výpočty navržené v této práci je možné použít k odhadu doby trénování a zrychlení. V každém případě trénování výpočty určily ideální počet výpočetních jednotek a typ serveru. Pro implementaci trénování jsem použil knihovnu Tensorflow.

Pro sítě s velmi velkým počtem parametrů jako je VGG16E je nejvhodnější sekvenční trénování na jedné pracovní jednotce, protože doba strávená přenosem vah a gradientů je velmi velmi vysoká a zabraňuje zrychlení výpočtu. Ve všech případech paralelního trénování této sítě bylo zrychlení menší jak 1, to bylo předpovězeno pomocí výpočtu. Přestože síť Resnet34 má menší počet vah ve srovnání s dříve uvedenou sítí, paralelní trénování této sítě nebylo výhodné. Použití distribuovaného serveru bylo nutné pro dosažení zrychlení většího než 1. Síť GoogLeNet má malý počet vah ve srovnání s dříve uvedenými sítěmi, a proto paralelní trénování této sítě bylo výhodnější. Přestože má tato síť méně vah než předchozí sítě, je stále použití distribuovaného serveru vhodné pro dosažení vyššího zrychlení. Sítě s velmi malým počtem vah jako je SqueezeNet se trénují velmi dobře na více paralelních jednotkách. Doba strávené přenosem vah nebo gradientů je velmi malá ve srovnání s dobou výpočtu gradientů jedné dávky. Při trénování sítě SqueezeNet na osmi výpočetních jednotkách jsem dosáhl zrychlení 6,5. Ve všech případech dokázaly moje odhady určit, jak se bude měnit velikost zrychlení trénování sítě při trénování na více výpočetních jednotkách.

Druhou nejdůležitější částí práce je návrh distribuovaného serveru. Cílem tohoto serveru zkrácení doby komunikace a tudíž i trénování. Toho je dosaženo zvýšením využití duplexního spojení mezi jednotkami, které provádějí trénování. Distribuovaný server zvýšil zrychlení nebo dosáhl stejného zrychlení jako centrální server ve všech případech. Použití distribuovaného serveru nikdy nezpůsobilo snížení zrychlení.

Dále jsem navrhl mechanizmus pro zefektivnění předávání vah a gradientů při trénování. Cílem tohoto mechanizmu je zajistit co nejrychlejší přenos dat mezi jednotkami. Toho by se mělo dosáhnout pomocí omezení počtu jednotek, které komunikují s jednou jednotkou v každém směru, pomocí semaforu. Tento mechanizmus je možné zkombinovat s distribuovaným serverem.

Literatura

[1] Abadi, M., Agarwal, A., Barham, P. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*. 2015, roč. 1, č. 212. S. 19.

[2] Abadi, M., Barham, P., Chen, J. et al. TensorFlow: A system for large-scale machine learning. *Google Brain*. 2016.

[3] Dean, J., Corrado, G., Monga, R. et al. Large scale distributed deep networks. *Advances in Neural Information Processing Systems*. 2012.

[4] Dijkstra, E. W. *The Origin of Concurrent Programming*. 2002. Cooperating Sequential Processes, s. 65–138.

[5] Feldman, J. a Rojas, R. *Neural Networks: A Systematic Introduction*. 1996.

[6] Goodfellow, I., Bengio, Y. a Courville, A. *Deep Learning*. 2016. <http://www.deeplearningbook.org>.

[7] Griffin, G., Holub, a. a Perona, P. Caltech-256 object category dataset. *Caltech mimeo*. 2007, roč. 11, č. 1.

[8] Gupta, S., Zhang, W. a Milthorpe, J. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning. In *The IEEE International Conference on Data Mining*. 2016.

[9] He, K., Zhang, X., Ren, S. et al. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*. 2016.

[10] Hochreiter, S. *Untersuchungen zu dynamischen neuronalen Netzen*. Institut für Informatik, Technische Universität München, 1991. Disertační práce.

[11] Hochreiter, S. Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies. *A Field Guide to Dynamical Recurrent Networks*. 2001.

[12] Iandola, F. N., Moskewicz, M. W., Ashraf, K. et al. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <1MB Model Size. *CoRR*. 2015, roč. 9349.

[13] Ioffe, S. a Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*. 2015.

[14] Kingma, D. P. a Ba, J. L. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*. 2015.

[15] Krizhevsky, A., Sutskever, I. a Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*. 2012.

[16] LeCun, Y. a Others. Generalization and network design strategies. *Connectionism in perspective*. 1989.

[17] Lin, M., Chen, Q. a Yan, S. Network In Network. In *International Conference on Learning Representations*. 2013.

[18] Ooi, B. C., Wang, Y., Xie, Z. et al. Singa: A Distributed Deep Learning Platform. In. 2015.

[19] Pacheco, P. *An Introduction to Parallel Programming*. 2011.

[20] Rosenblatt, F. The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain. *Psychological Review*. 1958, roč. 65, č. 6.

[21] Rumelhart, D. E., Hinton, G. E. a Williams, R. J. Learning Internal Representations by Error Propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. 1986. S. 318–362.

[22] Sermanet, P., Eigen, D., Zhang, X. et al. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. In. 2013.

[23] Simonyan, K. a Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations*. 2015.

[24] Strom, N. Scalable distributed DNN training using commodity GPU cloud computing. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*. 2015. S. 1488–1492.

[25] Szegedy, C., Liu, W., Jia, Y. et al. Going deeper with convolutions. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2015.

[26] Zinkevich, M., Alex, S., Weimer, M. et al. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*. 2010. S. 2595–2603.

Přílohy

Příloha A

Obsah DVD

Tato příloha slouží k popsání struktury paměťového média přiloženého k této práci. Následující seznam je obsah kořenového adresáře média.

`/report` Zdrojové soubory této práce.

`/src` Zdrojové soubory praktické části této práce.

`/local` Logovací výpisy vytvořené skriptem pro lokální trénování.

`/transport` Logovací výpisy vytvořené skriptem pro přenos vah.

`/distributed` Logovací výpisy vytvořené skriptem pro distribuované trénování.

`/results.ods` Výsledky trénování a přenosů vah ve formátu OpenDocument Spreadsheet.

`/report.pdf` Elektronická verze tohoto dokumentu.

`/poster.pdf` Elektronická verze plakátu.

Složka `/src` obsahuje mnoho skriptů a jiných pomocných souborů. Význam a obsah každého souboru je popsán v následujícím seznamu.

`abstract_network.py` Funkčnost sdílená všemi neuronovými sítěmi.

`barrier.py` Implementace synchronizačního mechanizmu bariéra v Tensorflow.

`caltech256_network.py` Funkčnost sdílená všemi neuronovými sítěmi, které je možné trénovat pomocí trénovací množiny Caltech256.

`dist.py` Distribuované trénování.

`distributed.sh` Rozdělení distribuovaného trénování na více jednotek.

`estimate.py` Výpočet odhadu zrychlení.

`googlenet.py` Implementace neuronové sítě GoogLeNet pro skripty `dist.py`, `local.py` a `transport.py`.

`killnodes.sh` Ukončení distribuovaného trénování na více jednotkách.

`local.py` Lokální trénování.

nodefile.txt Ukázka souboru s adresami pro distribuovaný výpočet.

resnet34.py Implementace neuronové sítě Resnet34 pro skripty `dist.py`, `local.py` a `transport.py`.

squeezenet.py Implementace neuronové sítě SqueezeNet pro skripty `dist.py`, `local.py` a `transport.py`.

transport.py Měření délky přenosu.

vgg16e.py Implementace neuronové sítě VGG16E pro skripty `dist.py`, `local.py` a `transport.py`.

Příloha B

Požadavky

- Python 3.5.2 nebo kompatibilní
- matplotlib 1.5.1 nebo kompatibilní
- Tensorflow 1.1 nebo kompatibilní

Příloha C

Popis rozhraní

C.1 Lokální trénování

Skript `local.py` slouží k lokálnímu trénování neuronových sítí. Skript používá rozhraní příkazové řádky. Podrobný popis tohoto rozhraní je na následujícím seznamu.

`-h` Vypiš nápovědu.

`--datadir DIR` Trénovací množina je uložená v adresáři `DIR`.

`--logdir DIR` Logovací výpisy se uloží do adresáře `DIR`.

`--weights FILE` První hodnoty vah jsou uloženy v souboru `FILE`.

`--net FILE` Neuronová síť pro trénování je uložena v souboru `FILE`.

`--accuracy` Spočítej přesnost sítě.

`--profile` Proveď profilování výpočtu. Výsledky jsou uloženy do adresáře pro logovací výstupy.

Pokud je zadán přepínač `-h`, všechny ostatní se ignorují. Pokud není zadán přepínač `--net`, program se ukončí kvůli chybě na příkazovém řádku.

C.2 Přenos vah

Skript `transport.py` slouží ke změření délky přenosu vah neuronových sítí ze serveru na pracovní jednotku. Musí se spustit na dvou jednotkách (pracovní jednotka a server), mezi kterými změří délku přenosu. Skript používá rozhraní příkazové řádky. Podrobný popis tohoto rozhraní je na následujícím seznamu.

`-h` Vypiš nápovědu.

`--logdir DIR` Logovací výpisy se uloží do adresáře `DIR`.

`--net FILE` Neuronová síť pro trénování je uložena v souboru `FILE`.

`--my-address ADDRESS` Adresa této jednotky.

`--other-address ADDRESS` Adresa jiné jednotky.

`--my-role ROLE` Role této jednotky. Parametr `ROLE` může nabývat hodnot `ps` pro server s parametry a `worker` pro pracovní jednotku.

Pokud je zadán přepínač `-h`, všechny ostatní se ignorují. Pokud není zadán přepínač `--net`, program se ukončí kvůli chybě na příkazovém řádku.

C.3 Distribuované trénování

Skript `dist.py` slouží k distribuovanému trénování neuronových sítí. Musí se spustit na každé jednotce, která bude použita pro trénování. Skript používá rozhraní příkazové řádky. Podrobný popis tohoto rozhraní je na následujícím seznamu.

`-h` Vypiš nápovědu.

`--datadir DIR` Trénovací množina je uložená v adresáři `DIR`.

`--logdir DIR` Logovací výpisy se uloží do adresáře `DIR`.

`--weights FILE` První hodnoty vah jsou uloženy v souboru `FILE`.

`--net FILE` Neuronová síť pro trénování je uložena v souboru `FILE`.

`--nodefile FILE` Seznam adres pracovních jednotek oddělených novým řádkem je uložený v souboru `FILE`.

`--index INDEX` Index adresy této pracovní jednotky v seznamu adres pracovních jednotek je `INDEX`.

`--sync SYNC` Způsob synchronizace trénování je `SYNC`.

`--server TYPE` Typ serveru pro uložení parametrů je `TYPE`. Hodnota `central` značí centrální server, hodnota `distributed` značí distribuovaný server.

`--accuracy` Spočítej přesnost sítě.

`--profile` Proveď profilování výpočtu. Výsledky jsou uloženy do adresáře pro logovací výstupy.

Pokud je zadán přepínač `-h`, všechny ostatní se ignorují. Pokud není zadán přepínač `--net`, program se ukončí kvůli chybě na příkazovém řádku. Hodnoty, kterých může nabývat parametr `SYNC` přepínače `--sync`, jsou vypsány v následujícím seznamu.

`none` Asynchronní trénování.

`join` Synchronní trénování se spojením dávek.

`split` Synchronní trénování s rozdělením dávek.

C.4 Výpočet odhadů zrychlení

Skript `estimate.py` slouží k odhadu zrychlení při trénování na více výpočetních jednotkách. Skript používá rozhraní příkazové řádky. Podrobný popis tohoto rozhraní je na následujícím seznamu.

`-h` Vypiš nápovědu.

`--t-grad TIME` Délka výpočtu gradientů je `TIME`.

`--t-comm TIME` Délka přenosu vah sítě je `TIME`.

`--type TYPE` Způsob trénování sítě je `TYPE`.

`--server TYPE` Typ serveru pro uložení parametrů je `TYPE`. Hodnota `central` značí centrální server, hodnota `distributed` značí distribuovaný server.

`--workers N` Maximální počet pracovních jednotek je `N`.

`--batches D` Počet dávek při trénování je `D`.

`--output OUT` Výstup skriptu je tvaru `OUT`.

Pokud je zadán přepínač `-h`, všechny ostatní se ignorují. Pokud nejsou zadány přepínače `--t-grad` a `--t-comm`, program se ukončí kvůli chybě na příkazovém řádku. Hodnoty, kterých může nabývat parametr `SYNC` přepínače `--sync`, jsou vypsány v následujícím seznamu.

`async` Asynchronní trénování.

`sync-join` Synchronní trénování se spojením dávek.

`sync-split` Synchronní trénování s rozdělením dávek.

Hodnoty, kterých může nabývat parametr `OUT` přepínače `--output`, jsou vypsány v následujícím seznamu.

`single` Výstupem skriptu je pouze hodnota zrychlení na maximální počtu pracovních jednotek.

`csv` Výstupem skriptu jsou hodnoty zrychlení pro jednu až maximální počet pracovních jednotek oddělené středníky.

`plot` Skript zobrazí hodnoty zrychlení pro jednu až maximální počet pracovních jednotek pomocí grafu.

Příloha D

Diagramy použitých neuronových sítí

Tato příloha obsahuje diagramy sítí popsaných v sekcích 2.4 až 2.7. Každý diagram se skládá z bloků popisující vrstvy sítě a hran, které značí napojení vstupů a výstupů těchto vrstev. Vysvětlení činnosti vrstev je uvedeno níže.

Konvoluce $n\times m\times c/s(T)$ Konvoluční vrstva, která používá filtr o velikosti $n\times m$, výsledkem sítě je c map příznaků a konvoluce se provádí s krokem s. Symbol T, značí jakým způsobem se má konvoluce chovat při práci s okrajovými body. Hodnota S (same, stejné) říká, že se má vstup vycpat stejnými hodnotami jako jsou na okrajích. Hodnota V (valid, validní) říká, že se počítá pouze s body vstupu, v tomto případě bude velikost výstupu menší než velikost vstupu. Tomuto parametru se obvykle říká padding (vycpávka).

Plně propojená n Plně propojená vrstva, která obsahuje n neuronů.

Maxpool $n\times m/s$ Podvzorkovací vrstva typu maxpool, která používá filtr o velikosti $n\times m$. Podvzorkování se provádí s krokem s.

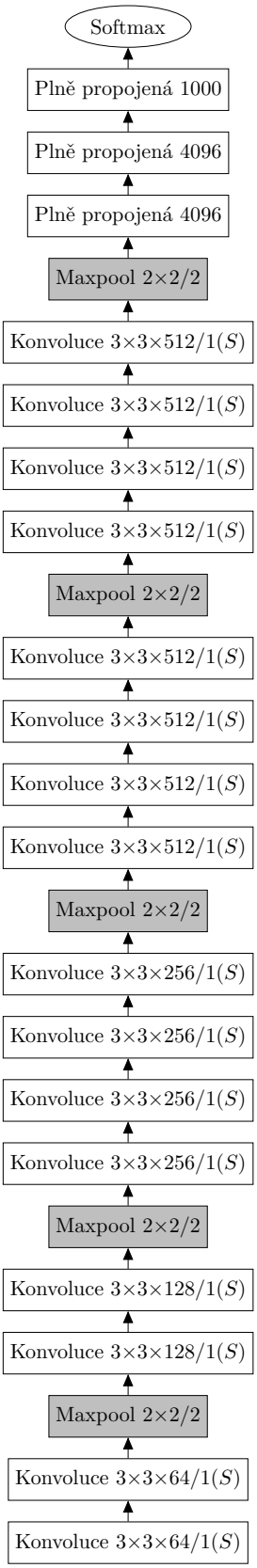
Avgpool $n\times m/s$ Podvzorkovací vrstva typu avgpool, která používá filtr o velikosti $n\times m$. Podvzorkování se provádí s krokem s.

Konkatenace n Vrstva, která provádí matematickou operaci konkatenace. Jejím výsledkem je n map příznaků.

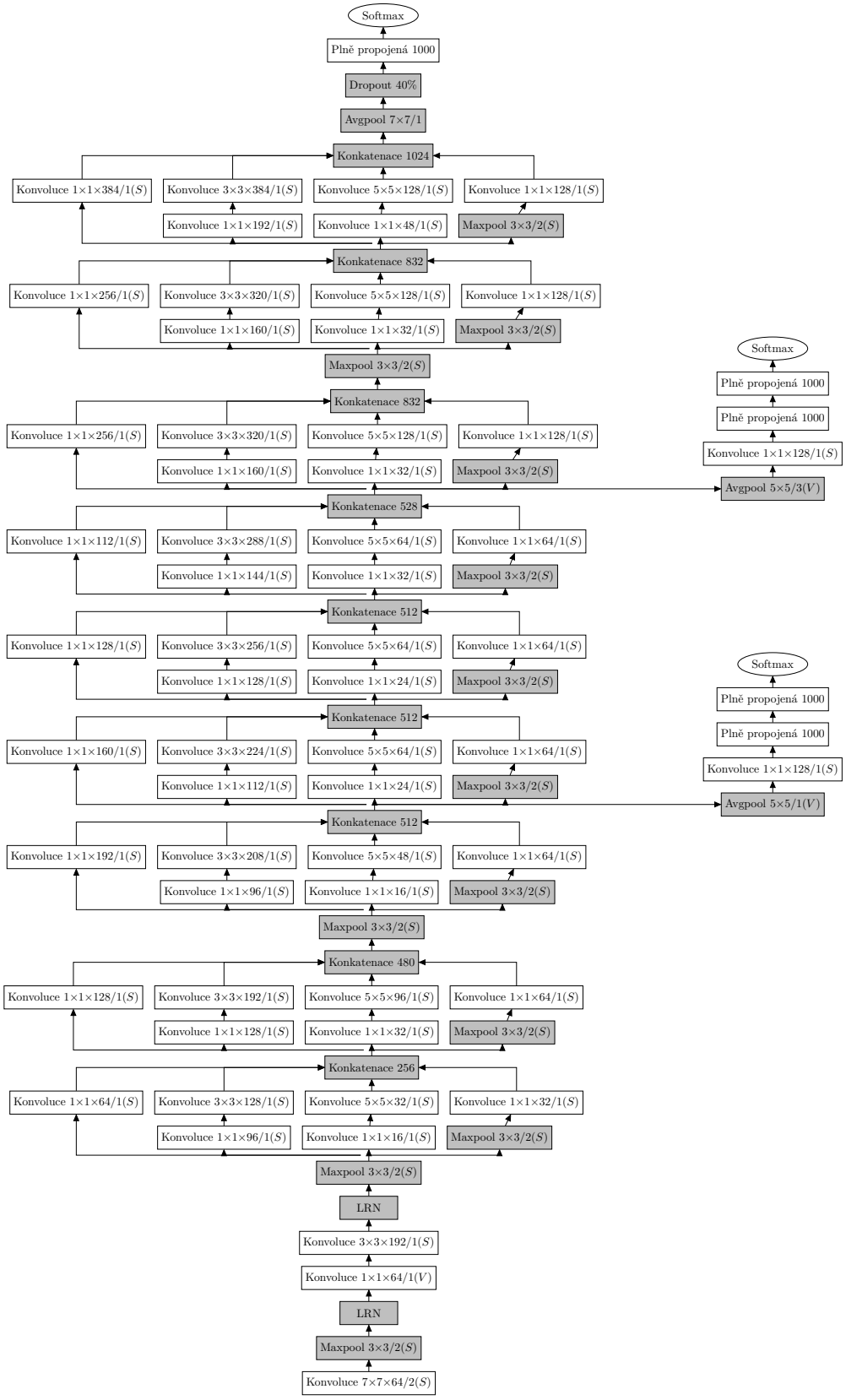
Sčítací vrstva Vrstva, která provádí matematickou operaci sčítání výstupů z různých vrstev. Všechny vstupy vrstvy musí mít stejnou velikost a výstup vrstvy má stejnou velikost jako vstupy. Tato vrstva je v diagramech označena jako kruh se symbolem + uprostřed.

Zdvojnásobení počtu map příznaků Tato vrstva provádí zdvojnásobení počtu map příznaků v reziduálním bloku sítě Resnet34. Detailní popis je v podsekci 2.7.

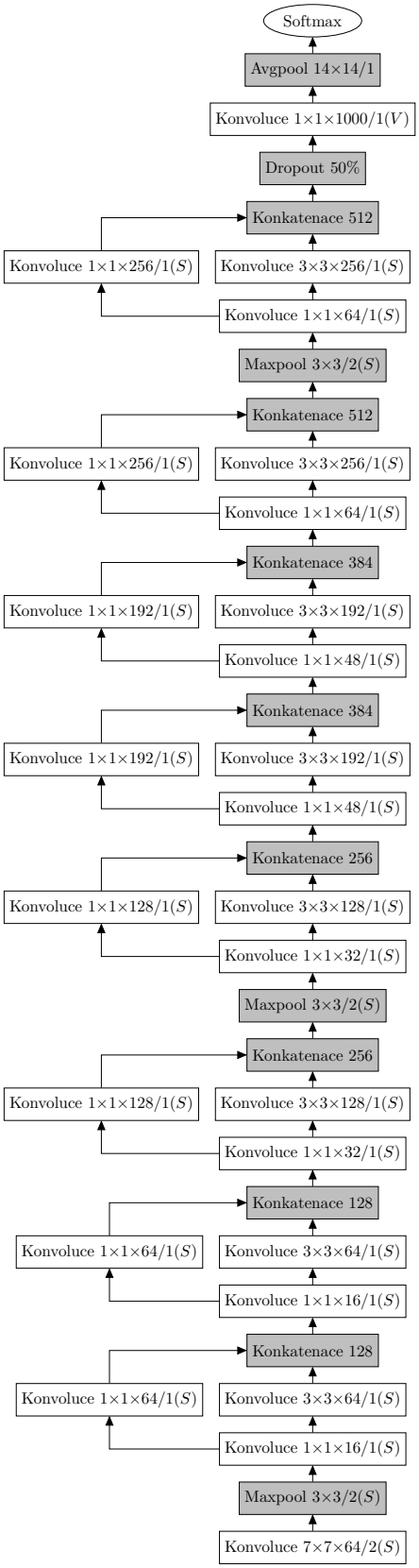
Softmax Vrstva implementující funkci softmax, která je popsaná v podsekci 2.3.5.



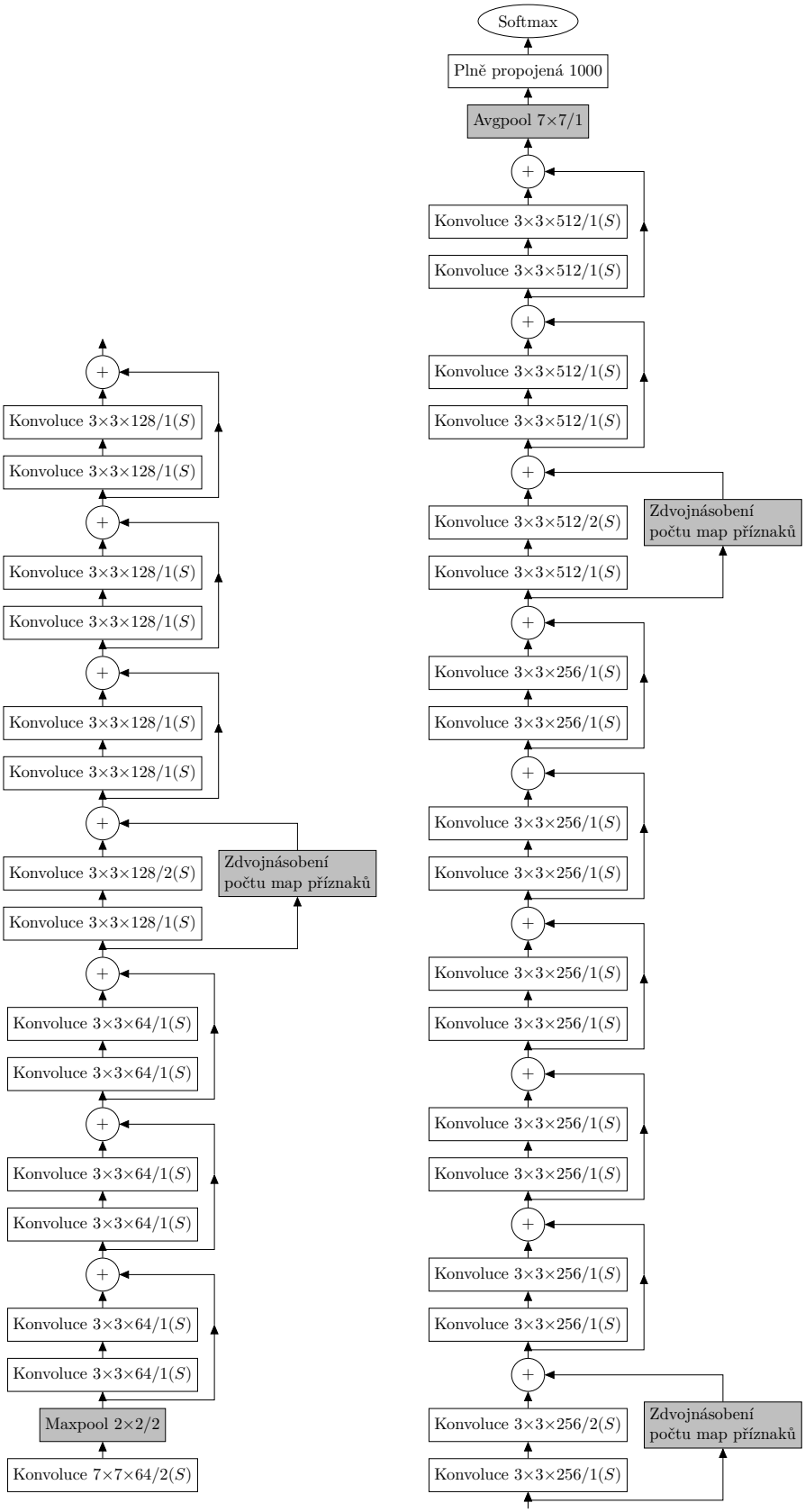
Obrázek D.1: Diagram sítě VGG16E.



Obrázek D.2: Diagram sítě GoogLeNet.



Obrázek D.3: Diagram sítě SqueezeNet.



Obrázek D.4: Diagram sítě Resnet34.