VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

EVOLUTIONARY DESIGN OF CONVOLUTIONAL NEURAL NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MICHAL PIŇOS**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Piňos Michal, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Evoluční návrh konvolučních neuronových sítí**
Evolutionary Design of Convolutional Neural Networks
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s problematikou neuronových sítí, konvolučních neuronových sítí a s využitím evolučních algoritmů v návrhu neuronových sítí.
2. Seznamte se s knihovnami pro práci s konvolučními neuronovými sítěmi, zaměřte se na TensorFlow.
3. Navrhněte způsob využití evolučního algoritmu při návrhu konvoluční neuronové sítě.
4. Navržený způsob implementujte a ověřte jeho přínos ve zvolené případové studii.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 25. října 2019

Abstrakt

Cílem této práce je návrh a implementace programu pro automatizovaný návrh konvolučních neuronových sítí (CNN) s využitím evolučních výpočetních technik. Z praktického hlediska tento přístup redukuje potřebu lidského faktoru při tvorbě CNN, a tak eliminuje zdlouhavý a namáhavý proces ručního návrhu. Tato práce využívá speciální formu genetického programování nazývanou kartézské genetické programování, které pro zakódování řešeného problému využívá grafovou reprezentaci. Tato technika umožňuje uživateli parametrizovat proces hledání CNN, a tak se zaměřit na architektury zajímavé z pohledu použitých výpočetních jednotek, přesnosti či počtu parametrů. Navrhovaný přístup byl otestován na standardizované datové sadě CIFAR-10, která je často využívána výzkumníky pro srovnání výkonnosti jejich CNN. Provedené experimenty ukázaly, že tento přístup má jak výzkumný, tak praktický potenciál a implementovaný program otevírá možnosti vzniku zajímavých řešení.

Abstract

The aim of this work is to design and implement a program for automated design of convolutional neural networks (CNN) with the use of evolutionary computing techniques. From a practical point of view, this approach reduces the requirements for the human factor in the design of CNN architectures, and thus eliminates the tedious and laborious process of manual design. This work utilizes a special form of genetic programming, called Cartesian genetic programming, which uses a graph representation for candidate solution encoding. This technique enables the user to parameterize the CNN search process and focus on architectures, that are interesting from the view of used computational units, accuracy or number of parameters. The proposed approach was tested on the standardized CIFAR-10 dataset, which is often used by researchers to compare the performance of their CNNs. The performed experiments showed, that this approach has both research and practical potential and the implemented program opens up new possibilities in automated CNN design.

Klíčová slova

konvoluční neuronové sítě, evoluční algoritmy, kartézské genetické programování, neuroevoluce

Keywords

convolutional neural networks, evolutionary algorithms, cartesian genetic programming, neuroevolution

Citace

PIŇOS, Michal. *Evoluční návrh konvolučních neuronových sítí.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Evoluční návrh konvolučních neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Piňos
2. června 2020

Poděkování

Rád bych poděkoval svému vedoucímu práce panu profesoru Ing. Lukáši Sekaninovi, Ph.D. za odborné vedení, konzultace, čas a cenné rady při tvorbě této práce. Dále bych rád poděkoval doktoru Ing. Vojtěchu Mrázkovi a docentu Ing. Jiřímu Jarošovi, Ph.D. za zprostředkování přístupu na školní výpočetní server a pomoc při provádění experimentů.

Obsah

Kapitola 1

Úvod

Konvoluční neuronové sítě (Convolutional Neural Networks, zkráceně **CNN**) tvoří skupinu hlubokých neuronových sítí, které se specializují na zpracování obrazových dat. Jejich popularita vysoce vzrostla po úspěchu sítě AlexNet [17] na soutěži *ImageNet 2012 Challenge* [37], kde se tato síť, s poměrně velkým odstupem, umístila na první pozici. Jedním z hlavních důvodů úspěchu této sítě byla efektivní implementace vysoce výpočetně náročného procesu učení na grafických akcelerátorech (GPU). Druhým podstatným důvodem pak byla architektura této sítě, která musela být pečlivě vytvořena jejími autory.

Návrh architektury konvolučních neuronových sítí dnes hraje důležitou roli ve výzkumu umělé inteligence (konkrétně v oboru rozpoznávání obrázků). Návrh nových architektur CNN vyžaduje velké úsilí, mnoho zkušeností a pokusů. Tento proces je tak velmi namáhavý a ne vždy efektivní. Z tohoto důvodu je nutné přemýšlet o nových způsobech tvorby těchto sítí, které by nebyly tak složité a náročné, a přitom poskytovaly výsledky minimálně srovnatelné s ručně navrženými sítěmi. Technika automatizovaného hledání architektur umělých neuronových sítí, nazývaná NAS (Neural Architecture Search) [12], je v dnešní době velmi populární [50, 57] a již dala vzniknout hlubokým neuronovým sítím překonávajícím ty odborně navržené [73, 67, 72].

Tvorba nových architektur rovněž obnáší problém optimalizace navrhované sítě z hlediska počtu parametrů, hyperparametrů a potřebného výpočetního výkonu. Tento požadavek je dán tím, že velikost výsledné sítě (ať už co se týká počtu parametrů nebo hyperparametrů) hraje důležitou roli z pohledu požadavků na zdroje zařízení, na kterém poběží. V dnešní době mobilních a vestavěných zařízení s omezenými zdroji je tento požadavek zcela pochopitelný. Najít optimalizační metodu, která by byla schopna tento úkol zcela vyřešit, je ale prakticky nemožné, a tak je často nutné uchýlit se k různým heuristickým metodám. Ty jsou schopny na úkor přesnosti, úplnosti či efektivity nalézt takové řešení, které se co nejvíce blíží zadaným požadavkům. Jednu skupinu takovýchto metod tvoří evoluční výpočetní techniky [56], založené na poznatcích ze biologické evoluce.

Cílem této práce je využití evolučních výpočetních technik jako optimalizační metody při automatizovaném návrhu architektur konvolučních neuronových sítí. Spojení evolučních výpočetních technik a neuronových sítí se v literatuře označuje jako *neuroevoluce* [66].

Pro účely evolučního návrhu CNN byla v této práci zvolena speciální technika nazývaná kartézské genetické programování [43], která ve spojení s genetickým algoritmem *NSGA-II* [8] dovoluje zaměřit se na hledání zajímavých architektur z pohledu přesnosti, počtu parametrů či použitých výpočetních prostředků.

Výsledkem této práce je implementace programu pro automatizovaný návrh konvolučních neuronových sítí, který při procesu hledání vhodné architektury CNN pro konkrétní

úkol zohledňuje uživatelem zadaná kritéria, a tak je schopen nalézt taková řešení, která co nejlépe splňují zadané požadavky. Rozšířením základní funkcionality tohoto programu je pak možnost jeho využití pro návrh řešení pro konkrétní zařízení, které například z důvodu ušetření cenných zdrojů využívá netradiční výpočetní prostředky v podobě aproximačních násobiček.

Kapitola 2

Evoluční výpočetní techniky

Vědci a technici z mnoha oborů se v dnešní době setkávají s problémy, jejichž řešení je často velmi složité a výpočetně náročné. Jako příklad lze jmenovat optimalizační problémy z oboru teorie grafů, robotiky, finančnictví nebo při návrhu součástek či elektrických obvodů.

Optimalizační problém lze definovat jako problém nalezení optimálního řešení z množiny S všech možných kandidátních řešení daného problému. Cílem je minimalizovat (nebo maximalizovat) tzv. účelovou funkci $f : S \to \mathbb{R}^+$. Pro mnohé z těchto problémů existují algoritmy, které jsou schopny optimální řešení najít. Tyto algoritmy jsou ale většinou velmi časově náročné, svou složitostí spadají do exponenciální třídy složitosti, a řešení nemusí být v dohledné době k dispozici. Mnoho z těchto algoritmů je rovněž příliš specializovaných pro určitou skupinu problémů nebo naopak moc obecných a neefektivních. Z tohoto důvodu je někdy vhodné použít různé heuristické algoritmy, jako jsou například evoluční výpočetní techniky, které za cenu snížené optimality či úplnosti nalezeného řešení zmenšují potřebný čas. Tyto algoritmy jsou tak schopny poměrně rychle poskytnout alespoň nějaké řešení, v mnoha případech však v praxi postačující.

2.1 Historie

Evoluční výpočetní techniky (Evolutionary Computation) zahrnují skupinu biologií inspirovaných technik, metod a algoritmů pro řešení složitých optimalizačních úloh, pro které mnohdy neexistují žádné optimální algoritmy. Jedná se tedy o jakési obecné techniky, použitelné skoro ve všech případech. Podle potřeby lze pak simulovat různé přírodní jevy pro řešení zadaných problémů. Existují tak různé optimalizační metody založené na kolektivním chování různých živočichů, jako jsou například ptáci [10], mravenci [9], včely [69] nebo světlušky [49]. Dále se lze setkat s čím dál více populárními genetickými algoritmy inspirovanými procesem dědičnosti [20], evolučními algoritmy inspirovanými biologickou evolucí [5], neuronovými sítěmi, které jsou založeny na neuronech v mozku [36], a mnoho dalších [53]. Inspirací pro evoluční výpočetní techniky byla biologická evoluce [56].

> *V přírodě proces biologické evoluce dokázal vytvořit extrémně komplexní autonomní živé organismy, schopné řešit mimořádně složité problémy, jako je například nepřetržité přizpůsobování se složitému, nepředvídatelnému a neustále se měnícímu prostředí. Pro tyto účely byly vyšší formy života, jako jsou například savci, vybaveny schopnostmi pro rozpoznávání vzorů, učení a myšlení. Velká rozmanitost situací, kterým se byl schopen život přizpůsobit, ukazuje, že proces evoluce je velmi robustní a schopen řešit mnoho tříd problémů.*
>
> —Alain Petrowski, Sana Ben Hamida

Základním stavebním kamenem evolučních technik je mechanismus evoluce, jak je popsán Charlesem Darwinem v jeho práci *O původu druhů* [7]. Zde je popsán proces přirozeného výběru, který na základě různých kritérií potlačuje nebo naopak zvýhodňuje určité skupiny jedinců. Dochází zde tak k selekci nejlepších jedinců, kteří pak mají šanci předat své užitečné vlastnosti do další generace. Dalším pilířem je moderní neodarwinismus, který zahrnuje pokročilejší teorie než je původní Darwinův přístup.

Počátky výzkumu evolučních výpočetních technik se datují až do konce 50. let minulého století a to zejména pracemi R. M. Friedberga [15], H. J. Bremermanna [2] a dalších. Ovšem kvůli omezenému výpočetnímu výkonu tehdejších počítačů se tyto techniky neujaly a široké vědecké komunitě zůstaly neznámé. Do povědomí vědecké komunity se začaly pomalu dostávat až o několik let později, v 60. a 70. letech 20. století. Klíčovou roli zde sehrály tři fundamentální přístupy, které položily základy evolučním algoritmům používaných v dnešní době. Jedná se o *genetické algoritmy*, *evoluční programování* a *evoluční strategie*. Ačkoliv tyto přístupy vypadají velmi podobně, byly vyvinuty nezávisle na sobě a za různými účely. K popularizaci těchto metod a názvu evoluční výpočetní techniky došlo až začátkem 90. let, a to díky vzniku akademického žurnálu *Evolutionary computation*, nakladatelství MIT Press [23].

Autorem genetických algoritmů je J. H. Holland [25], který se snažil najít a popsat obecný model samostatně se přizpůsobujících systémů. O rozvoj genetických algoritmů se rovněž zasadili autoři D.E. Goldberg [19], J. D. Schaffer [54], J. R. Koza [33] a mnoho dalších.

Evoluční programování bylo z počátku navrženo jako alternativní přístup k tehdejší umělé inteligenci. Jednalo se o konečný automat, který byl vyvinut simulací evoluce s operátory mutace a selekce. Autorem této techniky je L. J. Fogel [14].

Evoluční strategie, představeny autory I. Rechenbergem [51] a H.-P. Schwefelem [55], byly původně navrženy pro řešení složitých diskrétních a spojitých optimalizačních problémů.

2.2 Evoluční algoritmy

Evoluční algoritmy (zkráceně *EA*) jsou stochastické, heuristicky založené algoritmy inspirované biologickou evolucí. Jejich primárním úkolem je optimalizace problémů, pro které neexistuje žádný efektivní algoritmus[1]. Základní premisou konvergence této metody je, že pouze jedinci, kteří splňují určité požadavky, budou mít potomky, a tak budou moci předat své užitečné vlastnosti do další generace.

2.2.1 Obecný EA

EA pracují nad množinou *jedinců*, kteří představují nějaká validní řešení daného problému. Evoluce probíhá postupně po iteracích, nazývaných *generace*, do té doby, dokud nejsou splněny *ukončující podmínky*. Mezi ty patří například nalezení řešení s dostatečnou kvalitou nebo vyčerpání maximálního počtu generací. Množina jedinců, se kterými algoritmus pracuje v rámci jedné generace, se nazývá *populace*. V každé generaci je s využitím operátoru *selekce* vybráno několik jedinců, kteří se budou podílet na tvorbě nové populace. Z vybraných jedinců jsou pomocí operátorů *křížení* a *mutace* vyrobeni *potomci*, kteří

[1] Efektivním algoritmem se z pohledu teorie složitosti rozumí algoritmus, který je schopný poskytnout řešení v polynomiálním čase.

tvoří populaci následující generace. Jedinci jsou uloženi v lineární struktuře nazývané *chromozom*. Jednotlivé části chromozomu se nazývají *geny*, přičemž konkrétní forma genu se nazývá *alela*. Chromozom konkrétního jedince se často nazývá *genotyp*. *Fenotypem* potom rozumíme souhrn všech pozorovatelných vlastností a znaků jedince. Obecná forma EA je uvedena v algoritmu 1.

Algorithm 1 Obecný evoluční algoritmus

```
1:  P ← initial_population()
2:  g ← 0
3:  repeat
4:      evaluate(P)
5:      P′ ← select_individuals(P)
6:      O ← crossover(P′)
7:      O′ ← mutate(O)
8:      P ← replacement(P ∪ O′)
9:      g ← g + 1
10: until stop_criteria_satisfied()
```

První krok algoritmu 1 sestává z vytvoření iniciální populace P obsahující různorodé jedince. Tato diverzita v úvodní populaci zajistí, že prohledávání prostoru řešení S daného problému začne na různých místech.

Algoritmus následně přechází do hlavní smyčky. Zde dochází k ohodnocení všech jedinců současné populace P. Těm je pomocí *fitness funkce* (účelové funkce) přiřazena hodnota, reprezentující kvalitu řešení, které představují. Tato hodnota bývá označována jako *fitness* a slouží jako indikátor toho, jak dobře daný jedinec řeší zadaný problém.

Následuje fáze selekce, ve které je ze současné populace P vybráno několik jedinců P' s ohledem na jejich hodnotu fitness. Jedinci s lepší fitness mají větší šanci, že budou vybrání k následné reprodukci a tvorbě další generace, zatímco jedinci s horší hodnotou fitness mají šanci menší. Pro zachování a zajištění diverzity jedinců v následující generaci je tak vhodné do procesu výběru uvést prvek náhody, aby měli i horší jedinci možnost podílet se na tvorbě potomků. Vybraným jedincům se někdy říká *rodiče* (*parents*).

Z vybraných jedinců jsou poté operátorem křížení vytvořeni potomci O (*offspring*). Tento krok je jeden z nejdůležitějších, jelikož umožňuje předání užitečných vlastností z rodičů na potomka. Jedním způsobem tvorby nových jedinců je náhodná kombinace dvou (někdy i více) rodičů, inspirovaná biologickým pohlavním rozmnožováním v přírodě. Druhou možností je nepohlavní rozmnožování, kdy k tvorbě potomka postačí pouze jeden rodič. V tomto případě je potomek klonem rodiče.

Nově vytvoření potomci jsou následně podrobeni mutaci, která spočívá v náhodné změně některých parametrů jedince. Hlavním důvodem potřeby mutace je zachování diverzity, jinak by se v průběhu evoluce mohlo stát, že by si jedinci začali být moc podobní. Mutace tak představuje možnost, jak se při optimalizaci vymanit z lokálních extrémů. Tato technika je opět inspirována biologií.

V posledním kroku je stará populace nahrazena nově vytvořenými jedinci. Zde existuje několik možností, jak náhradu provést. Jednou možností je nahradit celou populaci novými jedinci, druhou možností je nahradit jen její část. Dalším podstatným problémem je výběr jedinců, kteří budou nahrazeni. Výběr obětí je podobně jako selekce řízen pravděpodobností a hodnotou fitness. Zde se můžeme setkat s konceptem *elitismu*, při kterém se skupina nejlepších jedinců vždy a beze změny dostane do další generace.

2.2.2 Genetické programování

Genetické programování (*GP*) jakožto speciální varianta evolučního algoritmu vznikla v 80. letech a o její popularizaci se nejvíce zasloužil John Koza [34]. GP je založeno na evolučním vývoji funkčních programů s využitím znalostí a pojmů ze světa genetiky. Tato technika uvažuje program reprezentovaný ve formě výpočetního grafu. Fenotyp v tomto případě představuje výsledný program. Při evoluci jedinců v GP jsou využity stejné postupy jako u obecného EA (algoritmus 1).

Technika GP našla uplatnění v mnoha oblastech, ve kterých byla schopna poskytnou výsledky srovnatelné s dosud nejlepšími lidskými výtvory [35]. Díky své vhodné reprezentaci vyvíjených jedinců je tato technika nejvíce využívána v oblasti symbolické regrese.

Hlavní variantou GP je stromové GP, ve kterém jsou kandidátní programy reprezentovány pomocí syntaktických stromů. V lineárním genetickém programování pak programy mají tvar posloupnosti instrukcí, obvykle v tříadresovém kódu.

2.3 Kartézské genetické programování

Kartézské genetické programování (Cartesian Genetic Programming, zkráceně *CGP*) je speciální forma genetického programování, ve které jednotliví jedinci představují programy reprezentované acyklickými orientovanými grafy. Tuto techniku poprvé představil J. F. Miller v roce 1999 [42], přičemž se jednalo o zobecnění techniky, kterou J. F Miller a další použili již v roce 1997 pro evoluční návrh elektronických obvodů [46]. Od svého uvedení jako nové formy genetického programování v roce 2000, byla tato technika rozšířena, modifikována a zkoumána mnoha výzkumníky z různých sfér. CGP se tak může pochlubit působivými výsledky z velké škály oborů, od návrhu elektronických obvodů, přes symbolickou regresi [38], tvorbu neuronových sítí [31] až po hraní počítačových her [65].

2.3.1 Reprezentace jedinců

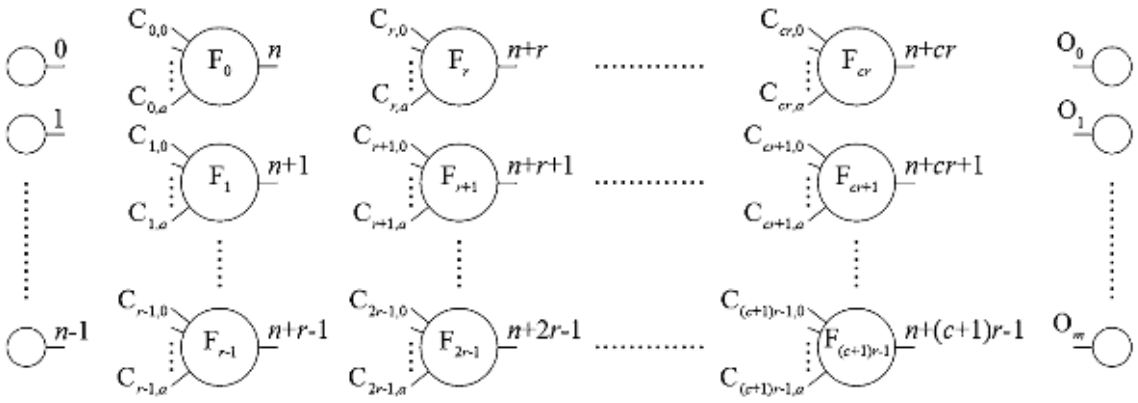
Jedinci v CGP představují funkční programy, reprezentované ve formě orientovaných acyklických grafů (Directed Acyclic Graph, zkráceně *DAG*). Uzly DAG jsou tvořeny výpočetními jednotkami, uspořádanými ve dvoudimenzionální mřížce (odtud název *"kartézské"*). Ze své podstaty je CGP přizpůsobeno pro práci se strukturami s více vstupy a výstupy.

Kandidátní DAG je reprezentován pomocí konečného řetězce celých čísel. Jednotlivé geny tvořící daný genotyp jsou celá čísla, která symbolizují, odkud daný uzel bere svoje vstupy, jakou funkci nad daty uzel vykonává a kde se nachází výsledné výstupy. Každý uzel má tak svůj *funkční gen* (function gene) ve formě celého čísla, udávajícího adresu do uživatelem definované *LUT* (look-up tabulky), kde se nachází konkrétní výpočetní funkce. Následující část genů tvoří *propojovací geny* (connection genes), které říkají odkud daný uzel bude brát svoje vstupy. Uzel může využívat jako své vstupy primární vstupy programu nebo výstupy uzlů z předchozích sloupců. Opět se jedná o celá čísla symbolizující adresu uzlů v nějaké datové struktuře. Počet vstupů každého výpočetního uzlu je zvolen jako maximální *arita* (počet vstupů) u funkcí v LUT. Propojovací geny tak určují výsledné propojení uzlů v grafu. Poslední geny každého genotypu se nazývají *výstupní geny* (output genes) a obsahují adresy uzlů, odkud lze získat konečný výstup.

Obecná forma genotypu jednotlivce vypadá následovně:

$$F_0 C_{0,0} \dots C_{0,a} \cdots F_r C_{r,0} \dots C_{r,a} \cdots F_{(c+1)r-1} C_{(c+1)r-1,0} C_{(c+1)r-1,a} O_1 \dots O_m$$

Genotyp se skládá z $r \times c$ genů výpočetních uzlů a m výstupních genů označených O_1 až O_m, kde r značí počet řádků a c počet sloupců mřížky výpočetních uzlů. Každý gen výpočetního uzlu začíná funkčním genem F, následovaný $a+1$ propojovacími geny označenými $C_{F,0}$ až $C_{F,a}$, přičemž $a+1$ vyjadřuje počet vstupů funkce F. Pro program s n vstupy mají primární vstupy přiřazeny adresy od 0 po $n-1$ a adresa výstupu každého výpočetního uzlu je dána jeho pozicí v chromozomu (lokus) + n (číslováno od 0 po sloupcích), tedy n až $n+(c+1)r-1$. Poslední část genotypu je tvořena výstupními geny $O_1 \ldots O_m$, které určují, z výstupů kterých výpočetních uzlů nebo primárních vstupů lze získat konečný výstup. Délka genotypu je pevně dána a lze ji vypočítat pomocí vzorce $rc(a+2)+m$.

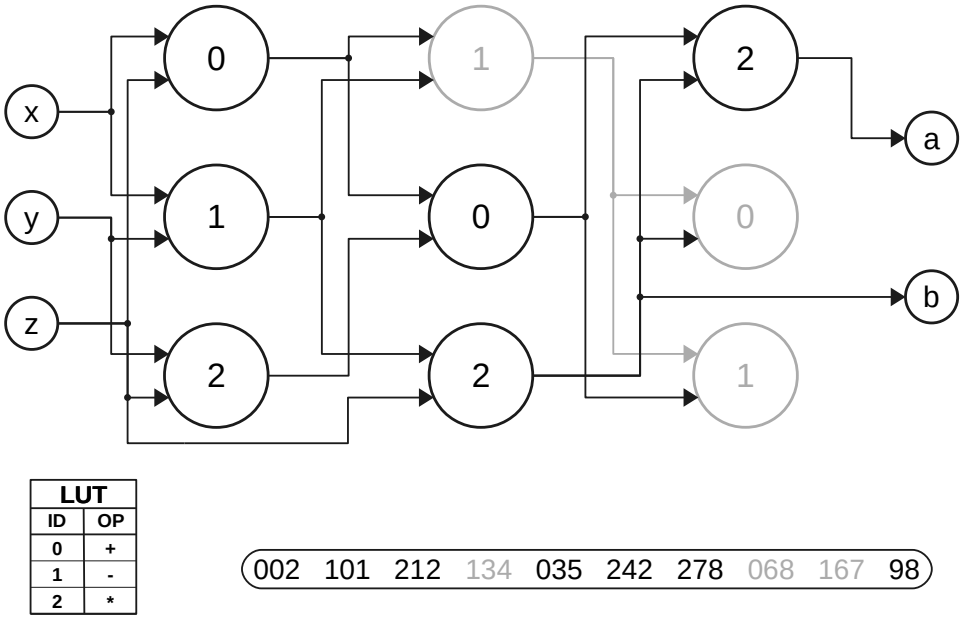


Obrázek 2.1: Obecný diagram CGP. Převzato z [43].

Obecná podoba CGP je uvedena na obrázku 2.1. CGP má tři uživatelem definované parametry, počet řádků r, počet sloupců c a levels-back l. První dva parametry udávají maximální počet výpočetních uzlů, který je $r \times c$. Nastavením parametru l uživatel udává, z kolika předchozích sloupců může daný uzel brát své vstupy. Tento parametr tak ukládá omezení při tvorbě propojení jednotlivých uzlů. Pokud $l = 1$, tak může každý uzel brát vstup z výstupů uzlů z předchozího sloupce nebo primárních vstupů. Při $l = 2$ může uzel brát svoje vstupy z výstupů uzlů ve dvou z levé strany sousedících sloupců nebo z primárních vstupů. Změnou těchto parametrů lze docílit různých topologií grafu. Speciálním případem je situace, kdy $r = 1$ a $l = c$. Při tomto nastavení může vzniknout libovolně propojený orientovaný graf, omezený pouze počtem uzlů, který je roven c.

Konkrétní příklad využití CGP pro návrh aritmetických výrazů je uveden na obrázku 2.2. Uvažujme situaci, kdy máme za úkol navrhnout program se třemi vstupy a dvěma výstupy, který řeší určitý matematický problém. Vstupy označme jako x, y, z a výstupy a a b, přičemž všechny tyto proměnné reprezentují reálná čísla. Výpočetní mřížka obsahuje 3 řádky a 3 sloupce výpočetních uzlů, představujících aritmetické operace sčítání, odečítání a násobení. Parametr l je nastaven na 1. Funkční geny byly zvoleny tak, že 0 značí sčítání, 1 odečítání a 2 násobení. Na obrázku 2.2 je ukázána funkční LUT, genotyp a fenotyp nalezeného řešení, kterému odpovídají následující rovnice:

$$a = ((x+z)+(y*z))*((x-y)*z)$$
$$b = (x-y)*z$$



Obrázek 2.2: Schéma k příkladu evolučního návrhu aritmetických výrazů. Šedé uzly symbolizují uzly, které se nepodílejí na výpočtu výsledku (*neaktivní uzly*). V genotypu jsou tyto části označeny šedě a nazývají se *nekódující geny*. Černá část schématu reprezentuje fenotyp.

2.3.2 Evoluce jedinců v CGP

Na rozdíl od GP používá CGP při evoluci pouze mutaci. Operátor křížení v této oblasti nezaznamenal velký úspěch [43], ačkoliv se dá s pokusy o jeho využití setkat [4]. Hlavní dva typy mutace používané v CGP jsou *bodová mutace* (point mutation) a *pravděpodobnostní mutace* (probabilistic mutation). Při bodové mutaci určuje uživatel parametrem μ_r (mutation rate) procento z celkového počtu genů, které se podrobí mutaci. U pravděpodobnostní mutace se uvažuje, že každý gen podstoupí mutaci s určitou pravděpodobností určenou uživatelem. Standardní CGP používá první způsob mutace, protože je efektivnější z pohledu rychlosti výpočtu. Při mutaci tak dochází ke změnám náhodně vybraných alel na nové, náhodné validní hodnoty. Validita nových hodnot je určena tím, o který gen se jedná. Pokud je k mutaci vybrán funkční gen, tak představuje validní hodnotu některá adresa z LUT. Při mutaci propojovacích genů jsou validní hodnoty adresy výstupů uzlů z předchozích sloupců respektujících parametr l nebo primárních vstupů. U výstupních genů patří mezi validní hodnoty adresy výstupu kteréhokoliv uzlu nebo primárního vstupu.

Neaktivní částí genotypu CGP se nazývají *redundantní*, jelikož jejich změna se ve fenotypu nijak neprojeví. Jak ukázali autoři Miller a Smith, tak CGP může obsahovat velké množství redundantních genů. V jejich experimentech se tento počet pohyboval až okolo 95 % [45]. Z toho tedy vyplývá, že v mnoha případech bude mutace probíhat právě v těchto redundantních regionech genotypu. Při evoluci má tento fakt jednu obrovskou výhodu, a to, že jelikož se fenotyp potomka nebude lišit od fenotypu rodiče, tak bude mít stejnou hodnotu fitness. Tím pádem odpadá náročná fáze vyhodnocení jedince.

Neutralita se ukázala být velmi podstatnou a užitečnou vlastností CGP. Výsledek mutace neaktivních částí genotypu se totiž může projevit až v pozdějších generacích, kdy se například po jedné aplikaci mutace výrazně změní topologie výsledného fenotypu. Dlouho-

době neaktivní segment uzlů se v průběhu evoluce mohl podstatně změnit a následně jednou mutací aktivovat. V tomto případě mluvíme o *neutrálním driftu*. Přínos tohoto jevu z pohledu efektivity procesu evoluce byl zkoumán mnoha autory a v řadě řešených problémů se ukázal být extrémně prospěšným [68, 63]. Avšak v nedávné publikaci článku, zabývajícím se optimalizací kombinačních obvodů pomocí CGP, bylo experimentálně ukázáno, že v tomto případě má neutrální mutace negativní dopad na výkonnost procesu evoluční optimalizace [71].

2.3.3 Prohledávací algoritmus

Proces evoluce je u CGP řízen algoritmem inspirovaným evoluční strategií $(\mu+\lambda)$, nejčastěji ve formě $(1+4)$ [43]. V tomto jednoduchém algoritmu značí μ počet nejlepších jedinců, kteří jsou v každé generaci vybráni jako rodiče. Z nich je následně pomocí mutace vytvořeno λ potomků. Jak si lze všimnout, tak se tato evoluční strategie, uvedená v algoritmu 2, moc neliší od obecného EA popsaném v algorimu 1. Hlavní změnou je, že CGP používá pouze mutaci a nejčastěji pracuje s jedním rodičem a λ potomky v každé generaci.

Další velmi podstatná vlastnost tohoto algoritmu se týká řádku 5. Při výběru nejlepšího jedince se může stát, že jeden či více potomků bude mít stejnou hodnotu fitness jako rodič a v populaci se nebude nacházet nikdo lepší. V tomto případě se za rodiče vybere náhodný potomek se stejnou fitness jako rodič. Toto rozhodnutí souvisí s jevem neutrality a má podstatný vliv na efektivitu procesu evoluce. Jak již bylo totiž zmíněno, tak i když má potomek stejnou fitness jako rodič, tak to neznamená že jsou identičtí. Jejich kódující i nekódující geny se totiž mohou lišit. Výběr nového jedince tak do procesu evoluce vnáší užitečnou diverzitu.

Algorithm 2 CGP algoritmus $(1 + \lambda)$

```
1:  P_0 ← generate λ random individuals
2:  g ← 0
3:  repeat
4:      evaluate P_g
5:      P ← select fittest individual from P_g
6:      O ← {}
7:      while |O| ≠ λ do
8:          O ← O ∪ {mutation of parent P}
9:      P_{g+1} ← O
10:     g ← g + 1
11: until stop criteria satisfied
```

2.3.4 Bloat

Jednou z největších výhod a záhad techniky CGP je, že na rozdíl od GP v něm nedochází ke vzniku *bloatu*. Ten je popsán jako negativní jev, kdy během evoluce velikost chromozomu roste, a to bez výraznějšího zlepšení fitness [44]. To se ukázalo být jednou z hlavních nevýhod GP, jelikož výsledné programy, či části výrazů v případě symbolické regrese, pak obsahují neefektivní nebo redundantní části. Vyhodnocení takovýchto jedinců je pak zbytečně náročné a nepřináší žádný užitek.

Absence bloatu v CGP byla dlouho zkoumána a za její příčinu byly nejčastěji považovány neutrální genetický drift nebo tzv. *length bias*. V prvním případě bylo argumentováno tím, že jelikož je v CGP výhodné pracovat s neaktivními uzly, tak během evoluce vzniká tlak, který limituje množství aktivních uzlů. Zdůvodnění absence bloatu v druhém případě vychází z propojovacích omezení při vytváření jedinců a jejich dopřednou výpočetní povahou. Jelikož každý uzel může být připojen pouze k uzlům z předchozích sloupců, tak uzly ve sloupcích blíže k primárním vstupům mají na výběr menší počet uzlů, se kterými se mohou spojit. To má za následek, že uzly ve sloupcích blíže k primárním vstupům mají větší pravděpodobnost být aktivní než ty vzdálenější. Pravděpodobnost, že bude některý uzel aktivní, tedy koresponduje s počtem uzlů, se kterými se může propojit.

Překvapivě ale bylo dokázáno, že ani jedna z těchto hypotéz není důvodem absence bloatu v CGP [62] a tak tato otázka zůstává otevřená.

Kapitola 3

Umělé neuronové sítě

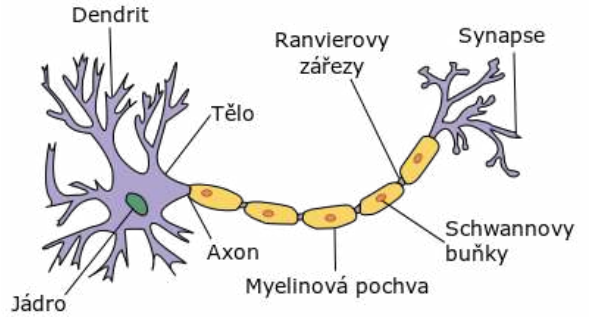
Umělé neuronové sítě (Artificial Neural Networks, zkráceně *ANN* nebo jen *NN*) představují jeden z mnoha výpočetních modelů používaných v oboru umělé inteligence. Dr. Robert Hecht-Nielsen, vynálezce prvních neuropočítačů, definoval v článku [3] neuronovou síť jako výpočetní systém složený z velkého počtu jednoduchých, hustě propojených výpočetních elementů, zpracovávajících informace podle jejich dynamické odezvy na externí vstupy. Ve své podstatě představují umělé neuronové sítě výpočetní model inspirovaný strukturou a funkcionalitou biologických neuronových sítí v lidském mozku. Jednotlivé výpočetní elementy jsou abstrakcí biologických neuronů, uspořádaných do vrstev, které jsou schopny reagovat na podněty a upravovat svoje chování s cílem vylepšit odezvu celé sítě.

Neuronové sítě byly poprvé představeny v roce 1944, kdy autoři Warren McCullough a Walter Pitts společně popsali, jak by mohl biologický neuron fungovat a s využitím elektrických obvodů sestrojili první model neuronové sítě. Hlavním smyslem jejich práce bylo ukázat, že lidský mozek může v principu fungovat jako výpočetní zařízení schopné realizovat stejné funkce jako počítač [22]. Dalším velmi podstatným pokrokem v této oblasti se stal model neuronu, vytvořený v roce 1957, nazvaný *perceptron*. Jeho autorem je Frank Rosenblatt, který svým modelem položil základy neuronovým sítím používaným v dnešní době.

3.1 Perceptron

Model perceptronu vychází z anatomie biologického neuronu, uvedeného na obrázku 3.1. Nervová buňka (nebo-li neuron) se skládá z těla, v němž se nachází buněčné *jádro*, obklopené velkým množstvím výběžků, nazvaných *dendrity*. Ty slouží jako vstupní část neuronu a jsou zodpovědné za přijímání informací (nervových vzruchů) od ostatních neuronů. Z neuronu vychází jedno dlouhé a tlusté vlákno, zvané *axon*, které zajišťuje rozšíření přijaté informace dále. Ten je u konce rozvětven, aby byl schopen předávat informace více neuronům. Na jednotlivých zakončeních axonu se nacházejí *synapse*, sloužící k přenosu informace mezi neurony. Synapse je elektrochemické spojení dvou neuronů, nejčastěji mezi axonem a dendritem.

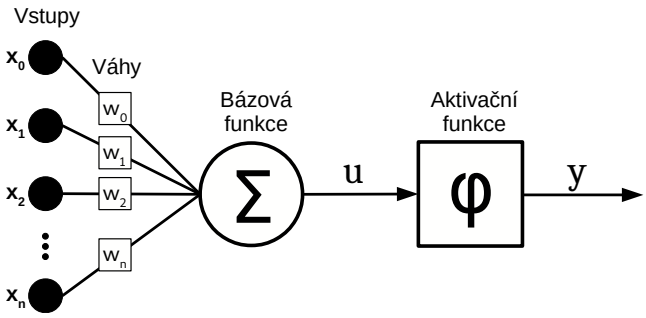
Neuron je po přijetí elektrického impulzu excitován (vybuzen), na základě čehož pak vzniká krátký elektrochemický pulz v podobě *akčního potenciálu*, který putuje dále po axonu až k synapsi. Podstatnou vlastností tohoto impulzu je to, že je generován jen v případě, že vstupní vybuzení přesáhne určitou hranici. Akční potenciál, který doputuje k synapsi zde způsobí chemickou reakci, která posílí nebo naopak utlumí vazbu mezi neurony.



Obrázek 3.1: Stavba biologického neuronu. Převzato z [13].

Tento koncept poprvé popsal Donald O. Hebb ve své knize *The Organization of Behavior* z roku 1949, ve které mimo jiné tvrdí, že tento proces je základní operací potřebnou pro učení. Tento princip vešel ve známost jako Hebbův zákon učení, který říká, že pokud je axon buňky A dostatečně blízko na to, aby excitoval buňku B a opakovaně nebo trvale se účastní na aktivaci této buňky, proběhne v jedné nebo obou těchto buňkách k růstovému procesu nebo metabolické změně, která má za následek, že je účinnost buňky A, jako buňky aktivující buňku B, zvýšena [24]. Z toho tedy vyplývá, že učení spočívá v nastavování synaptických vah.

S využitím Hebbova zákonu učení pak Frank Rosenblatt vylepšil původní model umělého neuronu. Hlavní změna spočívala v zavedení vah a algoritmu učení. Tento model byl pojmenován perceptron a je znázorněn na obrázku 3.2.



Obrázek 3.2: Model umělého neuronu – perceptronu.

Perceptron se skládá z $n+1$ rozměrného vstupního vektoru $\vec{x} = (x_0, x_1, \ldots x_n)$. S každým vstupem x_i je spojena jeho váha w_i pro $i = 0, 1, \ldots, n$. Váhy jsou rovněž uloženy ve váhovém vektoru $\vec{w} = (w_0, w_1, \ldots, w_n)$. Z jednotlivých vstupů a jejich vah je pomocí lineární bázové funkce (LBF) vypočítán *vnitřní potenciál* neuronu

$$u \ = \vec{x} \cdot \vec{w} = \sum_{i=0}^{n} x_i w_i. \tag{3.1}$$

Z vnitřního potenciálu u je následně prostřednictvím aktivační funkce φ spočítán výstup y, který závisí na tom, zda vnitřní potenciál přesáhl určitý práh θ. Výstup neuronu může být buď binární $y \in \{0, 1\}$ nebo bipolární $y \in \{-1, 1\}$.

Pro jednoduchost je práh θ zahrnut ve vztahu pro výpočet vnitřního potenciálu, důsledkem čehož je dáno $x_0 = 1$ a $w_0 = -\theta$ (bias). Perceptron využívá bipolární skokovou

aktivační funkci

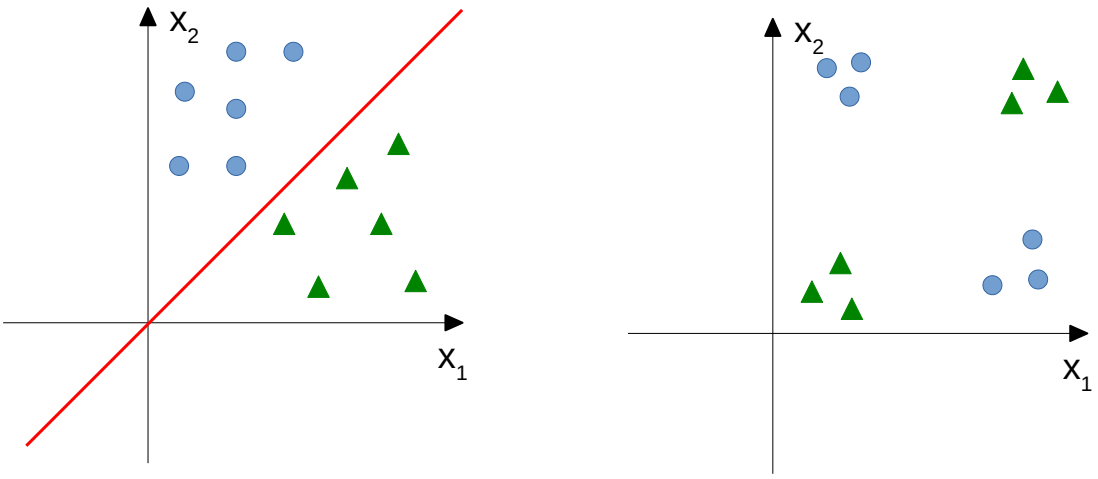
$$y = \begin{cases} 1 & \text{pro } u \geq 0 \\ -1 & \text{pro } u < 0 \end{cases} . \tag{3.2}$$

Proces učení perceptronu spočívá v nastavování váhového vektoru $\vec{w}$. Na vstup perceptronu je přiložen vstupní vektor $\vec{x}$ z trénovací množiny a následně je spočítán výstup y. Pokud se výstup perceptronu shoduje s očekávaným výstupem d, tak se váhy nemění. V opačném případě je váhový vektor $\vec{w}$ upraven tak, aby se chyba minimalizovala. Základní pravidlo pro učení perceptronu je uvedeno ve vzorci 3.3.

$$\vec{w}_{new} = \vec{w}_{old} + \mu(d - y)\vec{x} \tag{3.3}$$

Úvodní nastavení vah je libovolné a v každém dalším kroku učení jsou váhy upraveny podle uvedeného vzorce. Parametr μ se nazývá koeficient učení a většinou se pohybuje v intervalu $[0, 1]$.

Úkolem perceptronu je klasifikace vstupního vektoru do jedné ze dvou výstupních tříd. Grafická reprezentace klasifikace je uvedena na obrázku 3.3. Vstupní vektory zde představují body v prostoru a třída, do které patří, je reprezentována tvarem bodu. Proces učení je následně zodpovědný za hledání předpisu rovnice přímky, která dělí prostor na dvě části, čímž dochází ke klasifikaci podle toho, zda se body nacházejí pod nebo nad ní. V tomto případě vstupní vektory představují body v rovině, kterou dělí přímka. Obecně se u vícedimenzionálního prostoru jedná o hyperrovinu.



(a) Příklad lineárně separovatelného problému.

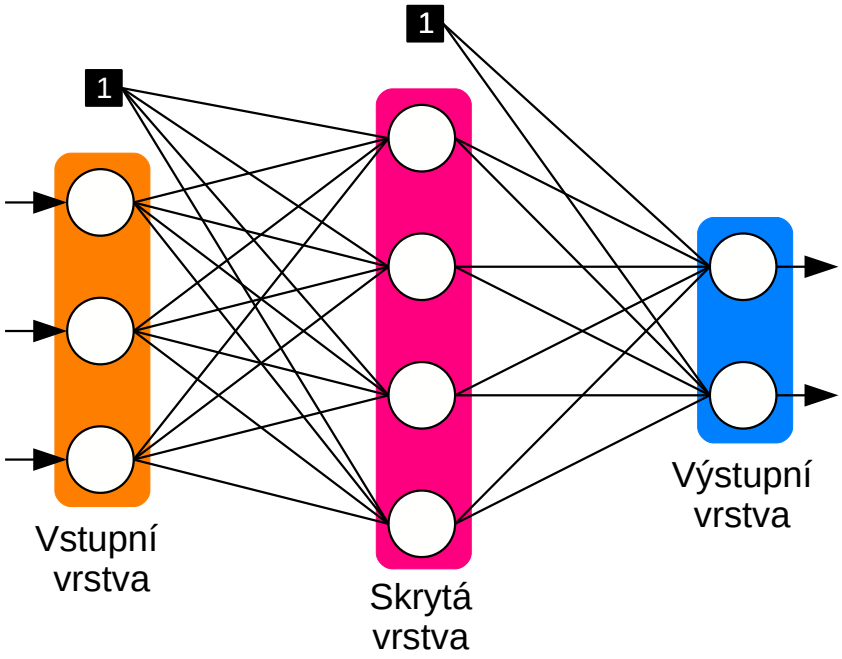
(b) Příklad lineárně neseparovatelného problému.

Obrázek 3.3: Grafická reprezentace klasifikace bodů pomocí perceptronu.

Perceptron představuje jednoduchou dopřednou umělou neuronovou síť zvanou *single layer perceptron* (SLP). Ta je schopna řešit pouze úlohy s lineárně separovatelnými vstupními vektory, jako je ta na obrázku 3.3a. To v roce 1969 dokázali Marvin Minsky a Seymour Papert ve své knize [41], čímž pozastavili výzkum v této oblasti až do 80. let. Odpovědí na tento problém bylo vytvoření vícevrstvového modelu vzájemně propojených perceptronů. Tento model byl pojmenován *vícevrstvý perceptron* (MLP) a společně s algoritmem zpětného šíření chyby tvoří základ moderních umělých neuronových sítí.

3.2 Dopředná umělá neuronová síť

Uspořádáním více neuronů do několika vrstev a jejich vzájemným propojením vznikne acyklická dopředná umělá neuronová síť, jako je ta na obrázku 3.4. Ta se skládá ze tří základních vrstev – vstupní, skryté a výstupní. Neurony jsou propojeny v dopředném směru (zleva doprava), což znamená, že vzájemné propojení neuronů v jedné vrstvě je zakázáno. Každý neuron je tak propojen se všemi neurony z předchozí vrstvy.



Obrázek 3.4: Model acyklické dopředné neuronové sítě se třemi vstupními neurony, čtyřmi neurony ve skryté vrstvě a dvěma výstupními neurony.

Acyklické dopředné neuronové sítě využívají pro výpočet vnitřního potenciálu u LBF (vzorec 3.1), z něhož je následně pomocí sigmoidální aktivační funkce (uvedené ve vzorci 3.4) spočítán výstup neuronu:

$$y = \frac{1}{1 + e^{-u}}. \tag{3.4}$$

Postup výpočtu odezvy dopředné neuronové sítě je jednoduchý. Na vstup sítě je přiložen vstupní vektor $\vec{x}$ a neurony první vrstvy si spočítají svoje výstupy (vzorce 3.1 a 3.4). Následuje postupný výpočet výstupů neuronů v dalších vrstvách, až se dojde k výstupní vrstvě, kde se nachází konečný výstup sítě. Pro správný výstup sítě je ovšem nutné tento model nejdříve naučit (správně nastavit váhy).

3.3 Algoritmus zpětného šíření chyby

Jedná se o nejrozšířenější a nejpoužívanější metodu učení acyklických dopředných neuronových sítí. Tento algoritmus je založen na minimalizaci chybové funkce E. Podmínkou pro použití této techniky je, aby byly aktivační funkce všech neuronů diferencovatelné. Následující popis algoritmu a vzorce byly převzaty z [70].

Uvažujme dopřednou neuronovou síť s n vstupními a m výstupními neurony. K výpočtu chybové funkce pro jeden prvek trénovací množiny p je použit vztah 3.5. Jedná se o jednu polovinu součtu kvadratických chyb, přes všechny neurony ve výstupní vrstvě. Symboly d_{pj} a o_{pj} značí po řadě očekávaný výstup j-tého neuronu a skutečný výstup j-tého neuronu výstupní vrstvy pro prvek p.

$$E_p = \frac{1}{2}\sum_{j=1}^{m}(d_{pj} - o_{pj})^2 \tag{3.5}$$

V první fázi algoritmu je na vstup sítě přiložen prvek p a v dopředném směru je spočítána chyba sítě E_p. Algoritmus se následně snaží tuto chybu minimalizovat. Toho je docíleno metodou gradientního sestupu, při které se váhový vektor $\vec{w}$ upraví tak, že se posune ve směru záporného gradientu chybové funkce E_p. Tato skutečnost je vyjádřena v následujícím vzorci

$$\Delta\vec{w} = -\mu\nabla E_p = -\mu\frac{\partial E_p}{\partial\vec{w}}. \tag{3.6}$$

S využitím řetízkového pravidla je změna i-té váhy j-tého neuronu v l-té vrstvě v případě použití LBF a sigmoidální aktivační funkce určena vzorcem

$$\Delta w_{ji}^l = -\mu\frac{\partial E}{\partial y_j^l}\cdot\frac{\partial y_j^l}{\partial u_j^l}\cdot\frac{\partial u_j^l}{\partial w_{ji}^l} = \mu\delta_j^l x_i^l, \tag{3.7}$$

kde μ je koeficient učení, $\frac{\partial u_j^l}{\partial w_{ji}^l} = x_i^l$ značí i-tý vstup v l-té vrstvě a δ_j^l je

$$\delta_j^L = -\frac{\partial E}{\partial y_j^L}\cdot\frac{\partial y_j^L}{\partial u_j^L} = (d_j - y_j^L)y_j^L(1 - y_j^L) \tag{3.8}$$

nebo

$$\delta_j^{l-1} = \sum_{k=1}^{n_l}(\delta_k^l w_{kj}^l)\cdot\frac{\partial y_j^{l-1}}{\partial u_j^{l-1}} = \sum_{k=1}^{n_l}(\delta_k^l w_{kj}^l)y_j^{l-1}(1 - y_j^{l-1}), \tag{3.9}$$

přičemž

$$\frac{\partial E}{\partial y_j^l} = (d_j - y_j^l) \tag{3.10}$$

a pro derivaci sigmoidy platí

$$\frac{\partial y_j^l}{\partial u_j^L} = y_j^l(1 - y_j^l). \tag{3.11}$$

Rovnice 3.8 platí pro neurony v poslední vrstvě L, zatímco rovnice 3.9 je určena pro výpočet δ_j^l pro $l = L-1, ..., 1$. Jak je vidět, tak algoritmus v této fázi pro výpočet δ_j^l postupuje ve zpětném směru (od výstupní vrstvy směrem ke vstupní).

V poslední fázi algoritmu si každý neuron upraví váhy podle vztahu

$$w_{ji}^l = w_{ji}^l + \Delta w_{ji}^l, \tag{3.12}$$

přičemž platí

$$\Delta w_{ji}^L = \mu(d_j - y_j^L)y_j^L(1 - y_j^L)x_i^L \tag{3.13}$$

nebo

$$\Delta w_{ji}^l = \mu\sum_{k=1}^{n_{l+1}}(\delta_k^{l+1}w_{kj}^{l+1})y_j^l(1 - y_j^l)x_i^l, \tag{3.14}$$

kde opět vzorec 3.13 je určen pro neurony výstupní vrstvy a 3.14 pro ostatní.

Pro účely klasifikace vstupních vektorů do K výstupních tříd je vhodnější použít ve výstupní vrstvě aktivační funkci *softmax*, uvedenou ve vzorci 3.15, kde u_k postupně značí vnitřní potenciál všech m neuronů v předchozí vrstvě. Tato funkce je schopna normalizovat výstup neuronové sítě tak, že konečný výstup sítě reprezentuje pravděpodobnost příslušnosti vstupního vektoru do dané třídy. Pro tyto účely je nutné, aby poslední vrstva neuronové sítě obsahovala neurony s aktivační funkcí softmax:

$$y_j = \frac{e^{u_j}}{\sum_{k=1}^{m} e^{u_k}}. \tag{3.15}$$

Dále je vhodné použít pravděpodobnostní chybovou funkci, která lépe vystihuje povahu klasifikačního problému. V ideálním případě totiž požadujeme, aby byl pro vstupní vektor aktivní pouze ten výstupní neuron, který reprezentuje třídu, do které vstupní vektor patří. Toho lze dosáhnou použitím chybové funkce 3.16, kde k značí neuron výstupní vrstvy, který reprezentuje správnou třídu:

$$E = -ln(y_k^L). \tag{3.16}$$

S využitím znalostí derivace chybové funkce 3.18 a derivace aktivační funkce softmax 3.17 lze upravit vztah pro výpočet δ_j^L v poslední vrstvě na vzorec 3.19. Výpočet δ_j^l v ostatních vrstvách zůstává při zachování sigmoidální aktivační funkce stejný jako ve vzorci 3.9.

$$\frac{\partial y_j}{\partial u_i} = \begin{cases} y_j(1-y_j) & \text{pro } j = i \\ -y_j y_i & \text{pro } j \neq i \end{cases} \tag{3.17}$$

$$\frac{\partial E}{\partial y_k^L} = \frac{\partial ln(y_k^L)}{\partial y_k^L} = \frac{1}{y_k^L} \tag{3.18}$$

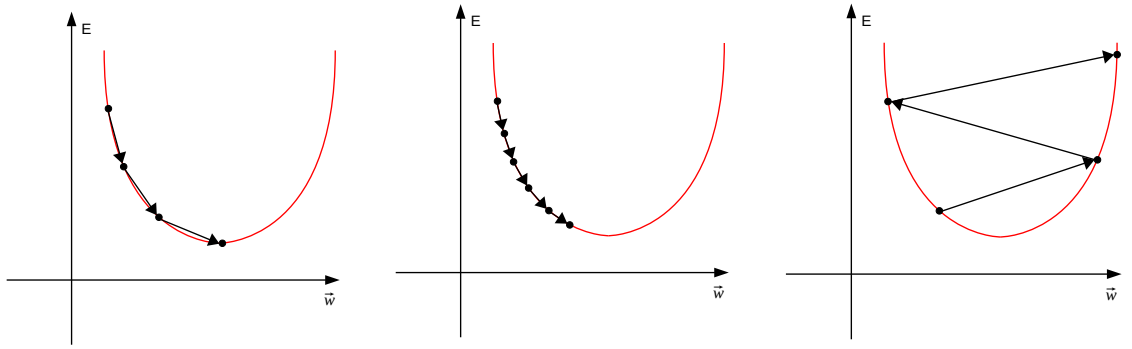
$$\delta_j^L = \frac{\partial \ln(y_k^L)}{\partial y_k^L} \cdot \frac{\partial y_j^L}{\partial u_j^L} = \frac{1}{y_k^L} \cdot \frac{\partial \frac{e^{u_k^L}}{\sum_{j=1}^{m} e^{u_j^L}}}{\partial u_j^L} = \begin{cases} 1 - y_j^L & \text{pro } j = k \\ -y_j^L & \text{pro } j \neq k \end{cases} \tag{3.19}$$

3.4 Metoda gradientního sestupu

Gradientní sestup představuje iterativní optimalizační metodu pro hledání lokálního minima nějaké funkce. V kontextu učení acyklických dopředných neuronových sítí je tato metoda použita pro hledání takových hodnot váhového vektoru $\vec{w}$, aby byla chybová funkce E co nejmenší. Toho je dosaženo inkrementálním postupem, kdy se při učení náhodně vybere prvek trénovací množiny p a je spočítána chyba sítě pro tento prvek. Pomocí vzorce 3.6 je pak upraven váhový vektor $\vec{w}$. Tento postup se opakuje pro všechny prvky trénovací množiny, dokud nejsou splněny ukončovací podmínky. Tato verze gradientního sestupu se nazývá stochastický gradientní sestup (SGD).

Parametr μ zde představuje velikost kroku ve směru záporného gradientu (proto znaménko $-$). Volba tohoto parametru má velký vliv na konvergenci sítě, jak je ukázáno na obrázku 3.5. Příliš malá hodnota způsobí, že se bude postupovat moc pomalu, zatímco příliš velká hodnota tohoto parametru může mít za následek divergenci.

Z tohoto důvodu vzniklo několik adaptivních metod, které mění velikost parametru μ v průběhu učení. Jednou z těchto metod je i metoda zvaná *Delta-Bar-Delta* [29], která uvažuje, že každá váha $\vec{w}_{ij}$ má vlastní koeficient učení μ_{ij}. Tyto koeficienty jsou následně



(a) Ukázka příkladu volby optimální hodnoty parametru μ.

(b) Ukázka příkladu volby příliš malé hodnoty parametru μ.

(c) Ukázka příkladu volby příliš velké hodnoty parametru μ.

Obrázek 3.5: Grafická reprezentace gradientního sestupu a volby parametru μ.

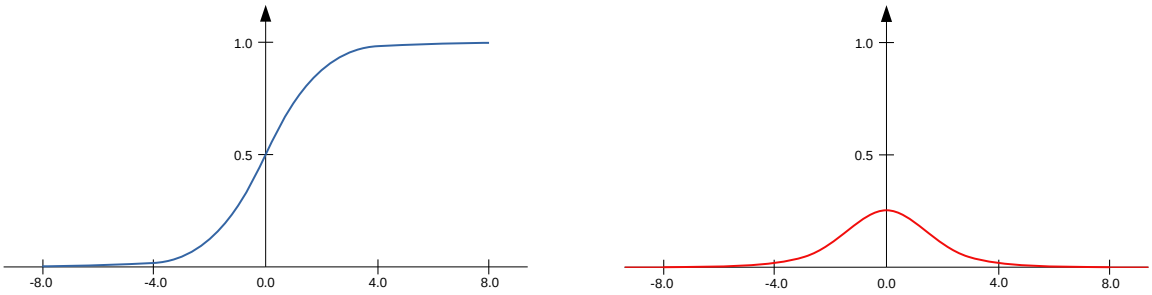
adaptivně upravovány tak, že pokud mezi předchozím a současným krokem nedošlo k překročení lokálního minima, tak je hodnota učícího parametru zvýšena. V opačném případě (tj. pokud došlo k překročení lokálního minima) je hodnota parametru μ_{ij} snížena. Tím je zajištěna rychlejší konvergence a vyhnutí se divergenci. Mezi další adaptivní metody patří například *SAB*, *SuperSAB* nebo *Rprop* [52].

3.5 Problém mizejícího gradientu

S použitím učících algoritmů založených na gradientním sestupu je spojen problém mizejícího gradientu, který má za následek to, že se proces učení sítě zpomalí, ne-li přímo zastaví. Tento problém pramení ze způsobu, jakým tyto metody aktualizují hodnoty vah. Změna hodnot jednotlivých vah každého neuronu je počítána při zpětném průchodu, kdy je chyba sítě propagována od koncových vrstev k počátečním. U sítí s mnoha vrstvami se tak stává, že se změna hodnot vah neuronů v počátečních vrstvách blíží nule. Jinými slovy je ve vztahu 3.12 změna i-té váhy j-tého neuronu v l-té vrstvě, tedy Δw_{ji}^{l}, velmi malá. Neurony v takovýchto vrstvách se tak v podstatě přestávají učit.

Jednou z hlavních příčin tohoto problému je špatná volba aktivační funkce. Na ilustraci 3.6 je zachycen průběh sigmoidální aktivační funkce (uvedené ve vzorci 3.4) a její derivace (uvedené ve vzorci 3.11). Z uvedených grafů je patrné, že obor hodnot této aktivační funkce je v intervalu $[0; 1]$, přičemž si lze všimnout, že pro dvě velmi velké (či velmi malé) hodnoty vstupu (osa x) bude rozdíl jejich výstupů velmi malý. Jinými slovy i velká změna na vstupu se na výstupu projeví velmi málo. To dokazuje i průběh derivace této funkce (uvedený v grafu 3.6b), kde se hodnota derivace pro velmi velké (či velmi malé) hodnoty blíží nule. Když si nyní uvědomíme, že v případě algoritmu zpětné propagace chyby se ve zpětném průchodu (při zpětném šíření chyby) využívá řetízkové pravidlo, při kterém dochází k postupnému násobení jednotlivých derivací v každé vrstvě, tak se s postupem do počátečních vrstev derivace exponenciálně zmenšuje a blíží nule.

Pro řešení tohoto problému existuje více možností. Nejjednodušší je zvolit jinou aktivační funkci, jako je třeba *ReLU* aktivační funkce, uvedená v sekci 4.9. Další možností je využití tzv. *skip* propojení, které přeskakují některé vrstvy a přivádějí nezměněný vstup do hlubších vrstev. Tento postup je uveden v sekci 6.3.4, kde je využit v implementaci *reziduální* vrstvy.



(a) Graf sigmoidální aktivační funkce.

(b) Graf derivace sigmoidální aktivační funkce.

Obrázek 3.6: Grafy průběhu sigmoidální aktivační funkce a její derivace.

3.6 Univerzální aproximační teorém

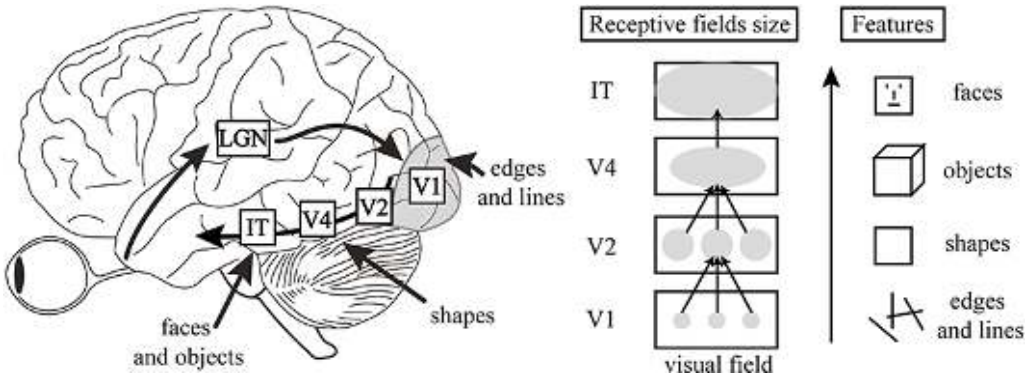
Nejpodstatnější část modelu vícevrstvé umělé neuronové sítě představuje skrytá vrstva, jejíž neurony jsou odpovědné za transformaci vstupního prostoru do vnitřní reprezentace neuronové sítě. Tím neuronová síť získává jiný pohled na vstupní data, která je pak výstupní vrstva lépe schopna zpracovat. Neuronovým sítím obsahujícím velké množství skrytých vrstev se říká *hluboké neuronové sítě* (anglicky Deep Neural Networks, zkráceně DNN).

Výpočetní sílu dopředných neuronových sítí objasňuje univerzální aproximační teorém, který poprvé popsal G. Cybenko v roce 1989. Ten říká, že dopředná acyklická neuronová síť s alespoň jednou skrytou vrstvou, obsahující dostatečný počet neuronů, je za určitých podmínek kladených na aktivační funkci, schopna aproximovat libovolnou spojitou funkci, definovanou v konečném n-rozměrném prostoru, v jiném konečném m-rozměrném prostoru s požadovanou nenulovou chybou [6]. V roce 1991 ukázal Kurt Hornik ve své práci [26], že schopnost neuronových sítí být univerzálními aproximátory vychází spíše z jejich vícevrstvové architektury než z volby aktivační funkce. Konečně v roce 2017 bylo na základě práce [40] ukázáno, že neuronová síť s ReLU aktivačními funkcemi a šířkou $n+1$ je schopna aproximovat libovolnou spojitou funkci n-rozměrných vstupních vektorů.

Kapitola 4

Konvoluční neuronové sítě

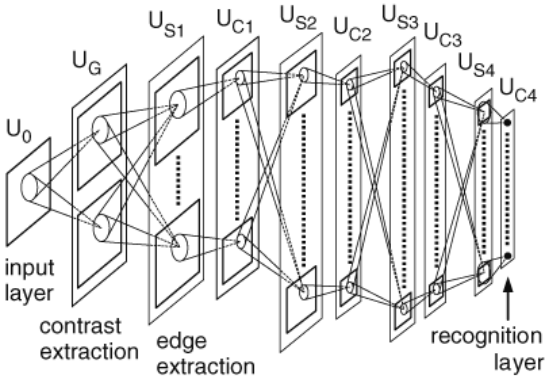
Tato skupina hlubokých neuronových sítí je zaměřena na rozpoznávání vzorů při analýze obrazových či zvukových dat. Jedná se o biologií inspirovanou variantu MLP, založenou na práci autorů Hubela a Wiesela [28], kteří v 60. letech minulého století ukázali, že určitá část mozkové kůry, umístěná v týlním laloku, nazývaná vizuální kortex, obsahuje neurony reagující na specifické oblasti zorného pole. Oblastem zorného pole, které se podílejí na excitaci určitého neuronu, se říká *receptivní pole* (receptive field). Ta se mohou lišit svojí velikostí či umístěním v zorném poli, přičemž receptivní pole sousedních neuronů se mohou překrývat.



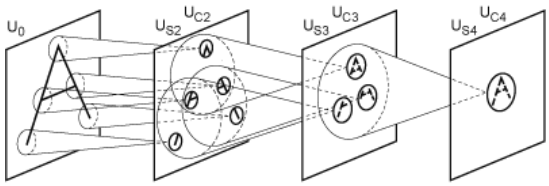
Obrázek 4.1: Model cesty vizuální informace mozkem. Informace putuje z očí do *LGM*, které rozvádí informaci dále do týlního laloku. Zde informace prochází několika vrstvami (*V1–V4*), odpovědnými za extrakci různých rysů ze vstupního signálu. Dále si zde lze všimnout variace velikosti receptivního pole, které se s každou následující vrstvou zvětšuje. Konečnou fázi tvoří *IT*, které je spojeno s pamětí a zodpovídá za rozpoznávání objektů. Obrázek byl převzat z [39].

Na základě tohoto popisu vytvořil v roce 1980 K. Fukushima [16] první neuronovou síť zaměřenou na rozpoznávání ručně psaných číslic, nazvanou *Neocognitron*. Tato síť byla tvořena dvěma základními typy neuronů, uspořádaných do vrstev, přičemž receptivní pole každého neuronu bylo tvořeno částí předchozí vrstvy. Jeden typ vrstvy se staral o extrakci rysů z předchozí vrstvy, zatímco druhý typ byl zodpovědný za toleranci odchylek v umístění těchto rysů. Z pohledu dnešních CNN se jednalo o konvoluční a seskupující vrstvy. Učení tohoto modelu spočívalo v nastavování vah neuronů, které byly v mnoha případech sdílené, což drasticky snižovalo požadavky na výpočetní výkon. Pro učení neocognitronu

bylo v průběhu let navrženo několik algoritmů, které byly mnohdy velmi složité. Při učení dnešních CNN ovšem převládá algoritmus zpětného šíření chyby, který ale nebyl v tehdejší době tolik rozšířený.



(a) Diagram znázorňující jednotlivé vrstvy modelu a jejich propojení.

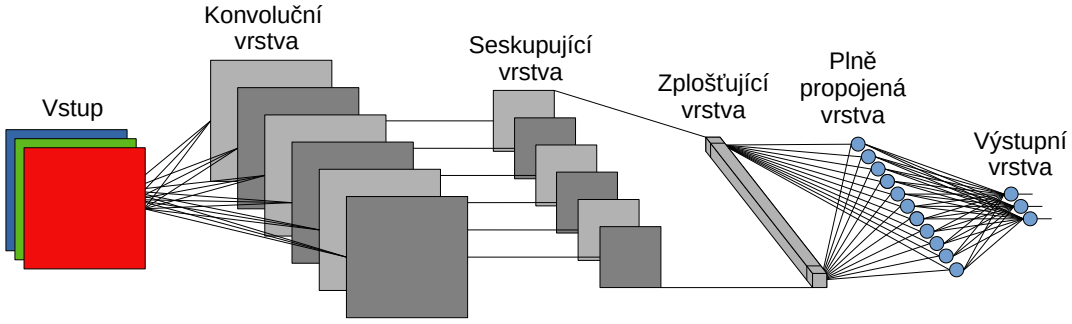


(b) Průřez architekturou modelu ukazující jak se liší velikost extrahovaných rysů s každou následující vrstvou.

Obrázek 4.2: Ukázka modelu neocognitronu převzatá z [16]. Z uvedených obrázků si lze všimnout inspirace tohoto modelu biologií mozku, zejména pak vrstvového modelu a postupné extrakce rysů.

4.1 Architektura CNN

Moderní konvoluční neuronové sítě se skládají ze čtyř základních typů vrstev – *konvoluční*, *aktivační*, *seskupující* a *plně propojené*. Jejich možné umístění v architektuře CNN je uvedeno na obrázku 4.3.

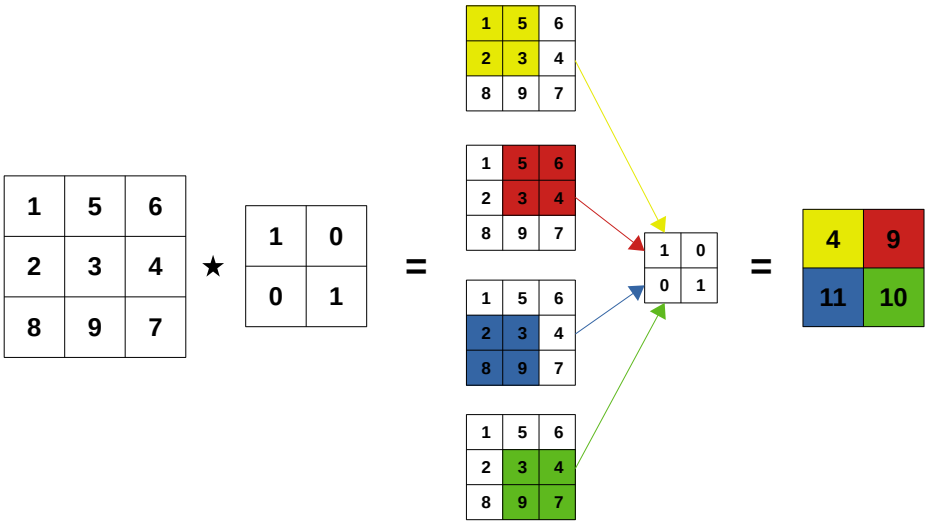


Obrázek 4.3: Klasická architektura CNN obsahující jednu konvoluční vrstvu, následovanou seskupující vrstvou, zplošťovací vrstvou, plně propojenou vrstvou a výstupní vrstvou.

4.1.1 Konvoluční vrstva

Tato vrstva je zodpovědná za extrakci užitečných rysů ze vstupního obrázku. K tomu využívá operaci konvoluce uvedenou ve vzorci

$$O[x,y] = (I \star K)[x,y] = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} I[x+i, y+j] K[-i,-j], \tag{4.1}$$



Obrázek 4.4: Grafická reprezentace operace konvoluce.

kde $O[x, y]$ představuje novou hodnotu pixelu na pozici $[x, y]$, I označuje vstupní obrázek a K značí filtr (kernel) o rozměrech $k_1 \times k_2$. Grafická reprezentace této operace je ukázána na obrázku 4.4. Jak je vidět, tak nová hodnota pixelu se získá přiložením filtru na patřičné místo ve vstupním obrázku a vynásobením pixelů s překrývajícími se koeficienty z filtru. Všechny tyto součiny jsou nakonec sečteny, čímž je získána výsledná hodnota. Pro získání hodnoty dalšího pixelu je nutné filtr posunout a výpočet opakovat.

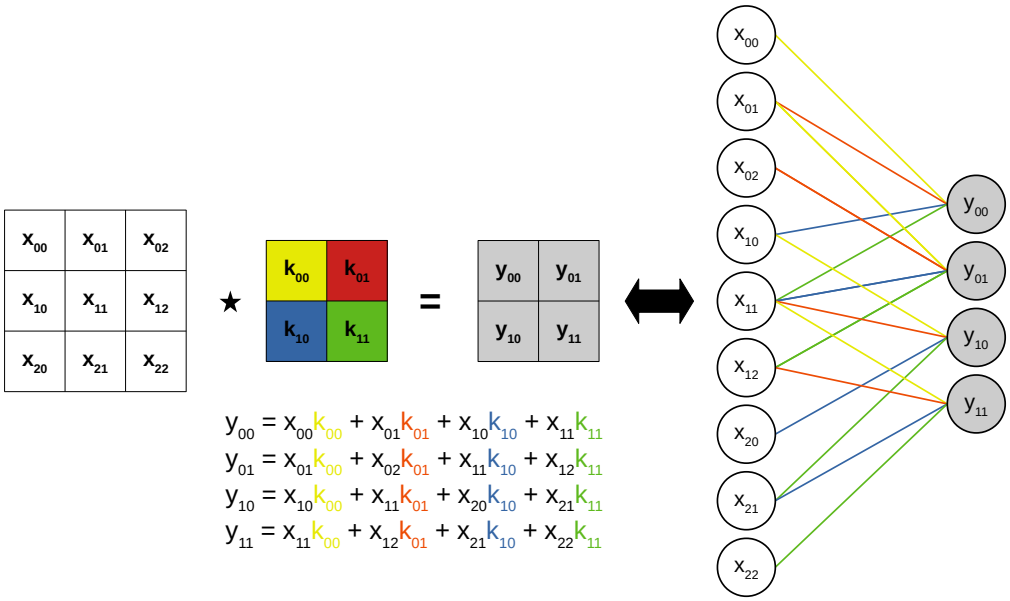
Jak již bylo zmíněno, tak se CNN podobá modelu MLP s tím rozdílem, že v této vrstvě nejsou neurony plně propojeny se všemi neurony z předchozí vrstvy. Tato skutečnost vyplývá z použití konvoluce namísto LBF, přičemž důležitým faktem je, že koeficienty filtru jsou uloženy ve váhovém vektoru. To je ukázáno na obrázku 4.5. Výpočet vnitřního potenciálu neuronu l-té vrstvy, představujícího pixel na pozici $[x, y]$, je tak dán vztahem

$$u^l_{x,y} = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} w^l_{i,j} o^{l-1}_{x+i,y+j} + b^l, \tag{4.2}$$

kde w^l značí filtr, $o^{l-1}_{x,y} = \varphi(u^{l-1}_{x,y})$ označuje výstup po aplikaci aktivační funkce na vnitřní potenciál neuronu předchozí vrstvy reprezentující pixel na pozici $[x, y]$ a b^l je bias.

Dalším velkým rozdílem oproti MLP je, že neurony v této vrstvě jsou uspořádány do 3D matice, jejíž dimenze jsou výška H, šířka W a počet kanálů C, který je pro vstupní obrázek nejčastěji 1 (černobílý) nebo 3 (barevný). Konvoluční vrstva pak obsahuje sadu filtrů pro každý kanál vstupního obrázku, přičemž počet sad filtrů udává výstupní počet kanálů této vrstvy.

Učení neuronů této vrstvy spočívá v hledání správných koeficientů filtrů, tedy váhového vektoru představující daný filtr. Při zpětném průchodu algoritmu zpětného šíření chyby to znamená úpravu vzorců pro výpočet gradientů. Zejména nás tedy zajímá výpočet vztahu uvedeného ve vzorci 4.3, tedy vliv váhy neuronu v l-té vrstvě, představující koeficient filtru na pozici $[i', j']$, na celkovou chybu sítě. Z definice operace konvoluce, použité při dopředném výpočtu vyplývá, že koeficient filtru označený $w^l_{i',j'}$ se podílí na výpočtu všech pixelů výstupního vektoru l-té vrstvy. Ten má pro filtr velikosti $k_1 \times k_2$ a vstupní obrázek o rozměrech $H \times W$ velikost $(H - k_1 + 1) \times (W - k_2 + 1)$. S využitím řetízkového pravidla tedy



Obrázek 4.5: Ilustrace podobnosti konvoluční vrstvy CNN a MLP. Důležitým detailem je sdílení vah, které je znázorněno barvami propojení mezi neurony.

můžeme psát

$$\frac{\partial E}{\partial w^l_{i',j'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \frac{\partial E}{\partial u^l_{i,j}} \frac{\partial u^l_{i,j}}{\partial w^l_{i',j'}}. \tag{4.3}$$

Parciální derivace vnitřního potenciálu $u^l_{i,j}$ vzhledem k proměnné $w_{i',j'}$ je podle vzorce 4.2 nenulová pouze v případě, kdy $i = i'$ a $j = j'$, důsledkem čehož platí

$$\frac{\partial E}{\partial w^l_{i',j'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+i',j+j'}, \tag{4.4}$$

přičemž $\delta^l_{i,j} = \frac{\partial E}{\partial u^l_{i,j}}$. Nyní je tedy potřeba zjistit, jaký vliv má vnitřní potenciál $u^l_{i',j'}$ na celkovou chybu sítě E. Z definice konvoluce uvedené ve vzorci 4.2 víme, že jeden pixel na pozici $[i', j']$ vstupního obrázku ovlivní $x_1 \times x_2$ pixelů další vrstvy. Opětným použitím řetízkového pravidla získáme

$$\frac{\partial E}{\partial u^l_{i',j'}} = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} \frac{\partial E}{\partial u^{l+1}_{i'-i,j'-j}} \frac{\partial u^{l+1}_{i'-i,j'-j}}{\partial u^l_{i',j'}}, \tag{4.5}$$

přičemž parciální derivace vnitřního potenciálu neuronu vrstvy $l+1$ je nenulová pouze pro člen $w^{l+1}_{i,j}\varphi(u^l_{i',j'})$, proto platí

$$\frac{\partial u^{l+1}_{i'-i,j'-j}}{\partial u^l_{i',j'}} = \frac{\partial (w^{l+1}_{i,j}\varphi(u^l_{i',j'}))}{\partial u^l_{i',j'}} = w^{l+1}_{i,j} \frac{\partial (\varphi(u^l_{i',j'}))}{\partial u^l_{i',j'}} = w^{l+1}_{i,j}\varphi'(u^l_{i',j'}), \tag{4.6}$$

kde φ' značí derivaci aktivační funkce.

Konečně po další substituci a dosazení vzorce 4.6 do 4.4 získáme vzorec pro výpočet $\delta_{i,j}$ v l-té vrstvě

$$\delta^l_{i,j} = \frac{\partial E}{\partial u^l_{i',j'}} = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} \delta^{l+1}_{i'-i,j'-j} w^{l+1}_{i,j} \varphi'(x^l_{i',j'}), \tag{4.7}$$

který lze vyjádřit jako součin úplné konvoluce δ^{l+1} s převráceným filtrem w^{l+1} a derivací aktivační funkce aplikované na vnitřní potenciál aktuálního neuronu l-té vrstvy

$$\delta^l_{i,j} = \delta^{l+1}_{i',j'} \star \mathrm{rot}_{180°}\{w^{l+1}_{i,j}\} \varphi'(x^l_{i',j'}). \tag{4.8}$$

Vzorce použité v této podsekci byly s určitými modifikacemi převzaty z [30], [18] a [21].

4.1.2 Aktivační vrstva

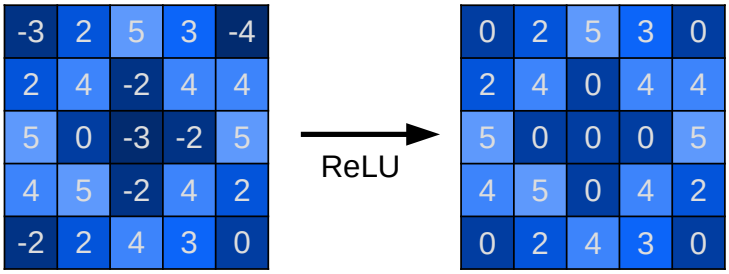
Účelem této vrstvy je aplikovat aktivační funkci φ na vstupní data, nejčastěji na výstup konvoluční vrstvy. Cílem použití aktivační funkce je vnést do modelu prvek nelinearity (konvoluce je lineární operace) a tím vylepšit vlastnosti sítě. Toho je dosaženo použitím nějaké nelineární funkce, nejčastěji ReLU definované jako

$$\varphi(x) = \begin{cases} 0 & \text{pro } x \leq 0 \\ x & \text{pro } x > 0 \end{cases}, \tag{4.9}$$

přičemž pro derivaci této funkce platí

$$\varphi'(x) = \begin{cases} 0 & \text{pro } x \leq 0 \\ 1 & \text{pro } x > 0 \end{cases}. \tag{4.10}$$

Příklad použití ReLU funkce je uveden na obrázku 4.6.



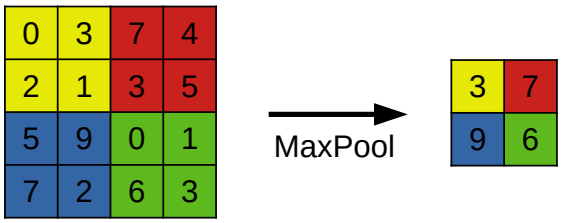
Obrázek 4.6: Příklad použití aktivační funkce ReLU. Z uvedeného příkladu je patrné, že funkce ReLU pouze nahrazuje všechny záporné hodnoty za 0.

4.1.3 Seskupující vrstva

Smyslem seskupujících vrstev je redukce prostorových rozměrů vnitřní reprezentace vstupního obrázku (*downsampling*). Díky tomu dochází ke snížení počtu vah a parametrů sítě, což má značný vliv na výkon. Další výhodou seskupující vrstvy je, že zachovává pouze důležité části obrázku a zbavuje se nepodstatných detailů, jako je například umístění či natočení extrahovaného rysu. Díky tomu je pak výsledný model invariantní vůči lokální translaci (posunu) ve vstupním obrázku, a tak robustnější. Tato vrstva se nejčastěji umisťuje za konvoluční a aktivační vrstvu, přičemž pracuje zvlášť pro každý vstupní kanál. Intuicí použití

této vrstvy je, že během konvoluce dojde k extrakci rysu, po aplikaci ReLU se zachovají jen dominantní části nalezeného rysu a při seskupení dochází k toleranci posunu či otočení daného rysu.

Nejpoužívanější metodou seskupení je vyhledávání maxim (*MAX Pooling*). Postup této techniky je uveden na obrázku 4.7.

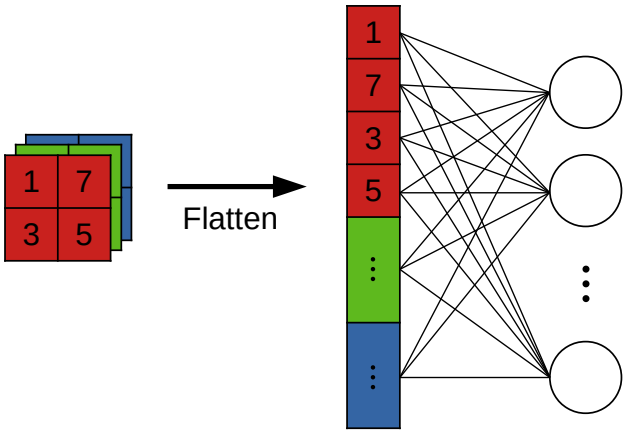


Obrázek 4.7: Příklad použití hledání maxim. Vstupní obrázek je rozdělen do dlaždic, přičemž výsledek je tvořen maximálními hodnotami jednotlivých dlaždic.

Jelikož tato vrstva neobsahuje žádné váhy, které by bylo potřeba se naučit, tak při zpětném průchodu dochází pouze k předání chyby neuronům předchozí vrstvy. Na výsledku sítě se podílely jen maximální prvky, a tak je chyba propagována jen k nim.

4.1.4 Zplošťovací vrstva

Tato vrstva je zodpovědná za transformaci vnitřní reprezentace obrázku do podoby vhodné pro plně propojenou vrstvu, jak je uvedeno na obrázku 4.8. Jedná se tedy o operaci zploštění vícedimenzionálního vstupu do jednodimenzionálního výstupu. Toho je dosaženo rozbalením jednotlivých kanálů vstupního vektoru za sebe.



Obrázek 4.8: Ukázka zplošťovací vrstvy, která po jednotlivých kanálech rozbaluje vícedimenzionální vstup na reprezentaci vhodnou pro plně propojenou neuronovou síť.

4.1.5 Plně propojená vrstva

Jedná se o poslední vrstvu CNN modelu, zodpovědnou za konečnou klasifikaci vstupního obrázku do jedné z výstupních tříd. K tomu je využita klasická architektura MLP s několika skrytými vrstvami a softmax aktivační funkcí ve výstupní vrstvě.

Kapitola 5

Neuroevoluce

Využití evolučních výpočetních technik při návrhu umělých neuronových sítí se zabývá obor zvaný *neuroevoluce*. Inspirací tohoto oboru byla opět příroda, ve které je za vznik inteligence odpovědný proces evoluce. Ve skutečnosti je evoluce jediným[1] známým procesem, který byl schopen vytvořit inteligentní bytosti (a inteligenci obecně). Evolučně navržené neuronové sítě se označují jako EANN (Evolutionary Artificial Neural Networks)[66] a jejich popis v této kapitole byl převzat z [66].

Jak již bylo zmíněno, tak v dnešní době je za vznikem umělé inteligence většinou člověk, který je odpovědný za návrh a konstrukci umělých neuronových sítí. Jako příklad lze uvést vznik modelu perceptronu, uvedený v kapitole 3. Za jeho výtvorem stál člověk, který jej vytvořil na základě znalostí poskytnutých lidmi zkoumající lidský mozek. Kdo ale stál za vznikem neuronových sítí v lidském mozku? Odpověď je snadná, byla to evoluce.

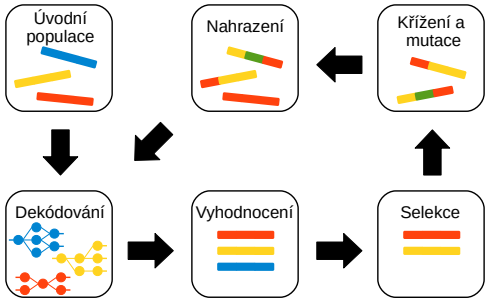
Cílem neuroevoluce je tak simulovat biologickou evoluci na výpočetních zařízeních za účelem vytvořit umělou inteligenci. Proces evoluce je v kontextu tvorby umělých neuronových sítí využit v několika instancích. První a asi nejčastější je využití evoluce pro návrh architektury neuronové sítě. Dalším použitím evoluce při návrhu neuronových sítí je evoluce vah. Jak bylo uvedeno, tak v dnešní době převládá pro nastavování vah algoritmus zpětného šíření chyby. Ten, ačkoli byl inspirován přírodou, opět představuje lidmi navrženou a formálně popsanou metodu, která nemá s biologickým učením moc společného. S tím souvisí i poslední využití evoluce, čímž je evoluční návrh učícího algoritmu. Grafická ukázka evolučního návrhu různých částí neuronových sítí je uvedena na obrázku 5.1.

Evoluční algoritmy určené k evoluci architektur a vah neuronových sítí se označují jako TWEANN (Topology and Weight Evolving Artificial Neural Network). Nejznámějším takovýmto algoritmem je genetický algoritmus NEAT (NeuroEvolution of Augmenting Topologies), představený autorem Kennethem O. Stanleym v roce 2002 [58].

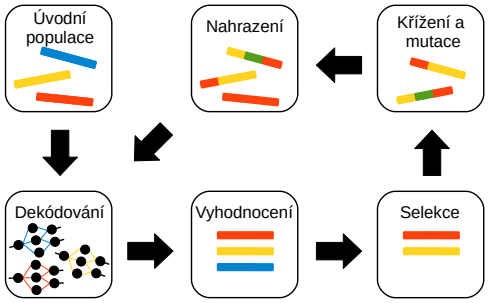
5.1 Evoluce architektury

Architektura je z pohledu výpočetní síly neuronových sítí velmi podstatná, jelikož udává způsob, jakým je vstupní informace zpracována a tím definuje i celkové schopnosti dané sítě. Z praktického pohledu přináší evoluční návrh architektury neuronové sítě možnost adaptovat se na dynamicky se měnící prostředí a specializovat se na různé úkoly, a to vše bez nutnosti zásahu člověka. Evolucí architektury neuronových sítí se rozumí evoluční návrh topologie, propojení a přenosové funkce jednotlivých uzlů umělé neuronové sítě.

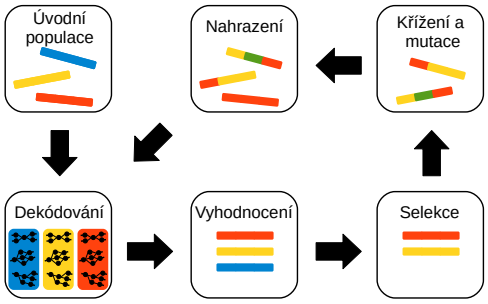
[1] Pomineme-li kreacionismus, který ovšem nemá žádné vědecké základy.



(a) Evoluční návrh architektur neuronových sítí.



(b) Evoluční návrh váhových vektorů neuronových sítí.



(c) Evoluční návrh učících algoritmů neuronových sítí.

Obrázek 5.1: Grafická reprezentace evolučního návrhu neuronových sítí.

Pro automatický návrh neuronových sítí byly navrženy metody, které se dají rozdělit na konstruktivní a destruktivní [66]. Konstruktivní metody začínají s minimální neuronovou sítí, obsahující minimální počet neuronů, vrstev a propojení. Postupem času pak podle potřeby přidávají další neurony, vrstvy a propojení s cílem vylepšit vlastnosti sítě. Destruktivní metody naopak začínají s maximální neuronovou sítí (maximální počet neuronů, vrstev a propojení) a v průběhu času postupně odebírají jednotlivé části modelu s cílem udržet vlastnosti sítě na určité úrovni.

Tyto metody ale mají tendenci zdržovat se v lokálních minimech a tak je jejich použití dost limitované. Z tohoto důvodu je evoluční návrh architektury neuronových sítí mnohem vhodnější, protože je schopen se vymanit z lokálního minima. Typický evoluční algoritmus určený pro návrh architektury neuronové sítě se skládá z následující sekvence kroků:

1. Podle použitého kódování je každý jedinec současné populace dekódován na odpovídající architekturu neuronové sítě.
2. S použitím učícího algoritmu je každá vytvořená neuronová síť podrobena učení.
3. Na základě přesnosti a složitosti jednotlivých jedinců je spočítána jejich fitness.
4. Podle fitness hodnoty jsou ze současné populace vybráni rodiče.
5. S využitím operátorů křížení a mutace je vytvořena nová generace
6. Pokud nebyla nalezena architektura splňující požadované kritéria nebo nebyl dosažen maximální počet opakování, vracíme se na krok 1.

5.2 Evoluce vah

V dnešní době, jak již bylo mnohokrát zmíněno, vévodí učení umělých neuronových sítí algoritmus zpětného šíření chyby. Ale i ten má své omezení, mezi které patří neschopnost vymanit se z lokálního minima, nemožnost použití tohoto algoritmu v modelech využívajících nediferencovatelné aktivační funkce, nebo problémy při učení rekurentních neuronových sítí. Učení, neboli nastavování vah, lze definovat jako optimalizační problém, jehož cílem je minimalizovat účelovou funkci. Pro tyto účely je tak vhodné využít evoluční výpočetní techniky.

Původní přístup k evolučnímu učení uvažoval fixní architekturu neuronové sítě, pro kterou byl následně s využitím evolučních technik hledán takový vektor váhových koeficientů, který pro zadaný úkol dosahuje co nejlepších výsledků. Obecný algoritmus evolučního učení sestává z těchto kroků:

1. Podle použitého kódování je každý jedinec současné populace dekódován na váhový vektor a následně je sestrojena neuronová síť s odpovídajícími váhami.
2. Pro každou neuronovou síť s danými váhami je spočítána fitness hodnota, která je založena na použité chybové funkci. Čím menší je chyba sítě pro daného jedince (váhový vektor), tím větší fitness hodnota je mu přiřazena a naopak.
3. Podle fitness hodnoty jsou ze současné populace vybráni rodiče.
4. S využitím operátorů křížení a mutace je vytvořena nová generace
5. Pokud nebyl nalezen váhový vektor splňující požadované kritéria nebo nebyl dosažen maximální počet opakování, vracíme se na krok 1.

5.3 Evoluce učícího algoritmu

Evoluční výpočetní techniky lze pozvednout až na úroveň návrhu samotného učícího algoritmu. Motivací tohoto kroku je skutečnost, že většina učících algoritmů se liší výkonem pro různé architektury neuronových sítí. S novými architekturami je tak potřeba přijít i s novými způsoby učení. Ty jsou ale většinou výsledkem matematické analýzy zkoumaného problému a jejich použití bývá často velmi specifické a limitované.

Cílem evoluce učícího algoritmu je tudíž poskytnout schopnost adaptace procesu učení pro nově vytvořené architektury neuronových sítí, které například vzešly z evolučního návrhu architektur neuronových sítí. Typický postup evoluce učícího algoritmu je shrnut v následující krocích:

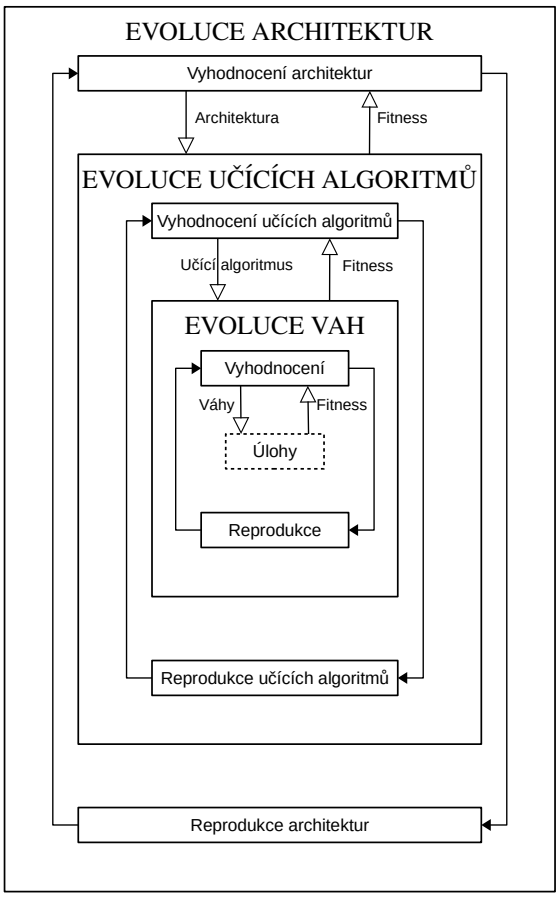
1. Podle použitého kódování je každý jedinec současné populace dekódován na učící algoritmus.
2. Následně je sestrojena množina náhodně vygenerovaných neuronových sítí s náhodně přiřazenými váhami.
3. Jednotlivé neuronové sítě jsou podrobeny učení jednotlivými učícími algoritmy.
4. Každému jedinci je přiřazena fitness hodnota podle toho, jak dobře byl schopen naučit individuální náhodně vygenerované neuronové sítě.
5. Podle fitness hodnoty jsou ze současné populace vybráni rodiče.

6. S využitím operátorů křížení a mutace je vytvořena nová generace jedinců.

7. Pokud nebyl nalezen učící algoritmus splňující požadované kritéria nebo nebyl dosažen maximální počet opakování, vracíme se na krok 1.

5.4 Univerzální schéma evolučních neuronových sítí

Jak už bylo řečeno, tak ultimátním cílem neuroevoluce je, po vzoru přírody, vytvoření plně autonomní, univerzální a hlavně adaptivní umělé inteligence, schopné poradit si se všemi úkoly, na které narazí. Schéma 5.2 zachycuje obecnou formu neuroevoluce, která spojuje výše popsané evoluční návrhy jednotlivých částí neuronových sítí.



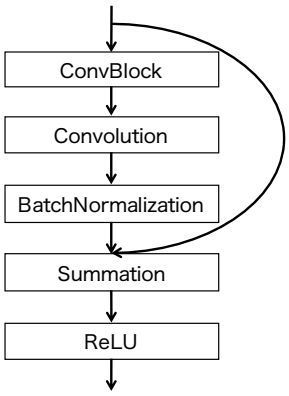
Obrázek 5.2: Univerzální schéma EANN. Převzato z [66].

5.5 Využití CGP v kontextu neuroevoluce

Využití CGP při návrhu umělých neuronových sítí se označuje jako *CGPANN* (Cartesian Genetic Programming of Artificial Neural Networks) [32]. Jedná se o neuroevoluční algoritmus, který využívá přímého zakódování architektury, vah a funkcí do genotypu. CGPANN tak představuje TWEANN algoritmus, který při procesu evoluce využívá výhradně operátoru mutace. Na rozdíl od genetických TWEANN algoritmů je CGPANN konstruktivní i destruktivní algoritmus.

Příklad úspěšného využití CGP při návrhu CNN je uveden v práci [59]. Její autoři vytvořili metodu nazvanou CGP-CNN, která používá CGP kódování jedinců, schopné reprezentovat architekturu a propojení výsledné CNN. Výhodou tohoto kódování je jeho flexibilita, jelikož umožňuje reprezentaci různě hlubokých sítí i využití *skip* propojení. Mimo jednoduchých výpočetních uzlů CGP mřížky, jako jsou konvoluční a seskupující vrstvy, autoři uvedli i složitější modul, nazvaný *ResBlock*. Tento modul představuje komplexnější výpočetní uzel vytvořený z jednodušších operací.

ResBlock je složen z konvoluce, konkatenace a ReLU aktivační funkce. *Skip* propojení přeskakuje konvoluční část tohoto modulu a přivádí nezměněný vstup modulu do konkatenačního uzlu, kde dochází ke konkatenaci s výsledkem konvoluce. Na výsledek konkatenace je následně použita aktivační funkce ReLU. Architektura modulu ResBlock je uvedena na obrázku 5.3.



Obrázek 5.3: Architektura modulu ResBlock. BatchNormalization provádí normalizaci hodnot po procesu konvoluce. Převzato z [59].

Kapitola 6

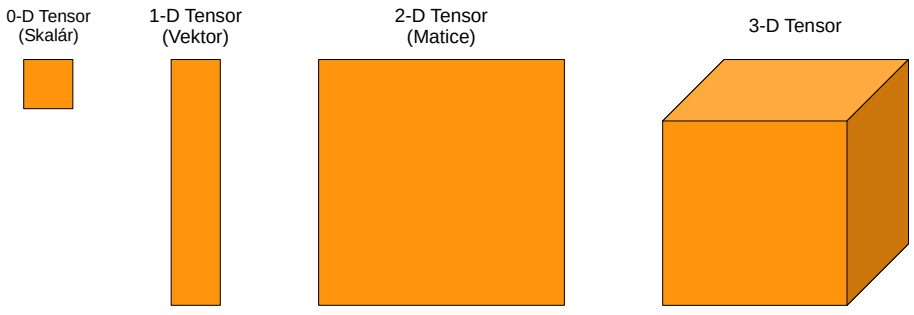
Návrh řešení

Jak již bylo řečeno, tak návrh architektur konvolučních neuronových sítí je časově náročný proces, nejčastěji založen na metodě pokus-omyl. K návrhu efektivní architektury konvolučních neuronových sítí je rovněž potřeba odborných znalostí. Proces návrhu neuronových sítí je tak velmi namáhavý, zdlouhavý a výsledné neuronové sítě jsou často specializované a ne vždy optimální (co se týče počtu parametrů sítě).

Cílem této kapitoly je popis programu pro automatický návrh konvolučních neuronových sítí. Pro tyto účely byla zvolena technika kartézského genetického programování popsaného v sekci 2.3. Metoda CGP je totiž ze své podstaty vhodná pro práci s acyklickými dopřednými sítěmi a dále dovoluje vlastní definici výpočetních uzlů výpočetní mřížky, což je z praktického pohledu velmi výhodné, protože uživatel si může zvolit, z jakých výpočetních uzlů bude výsledná síť složena. Pro práci s konvolučními neuronovými sítěmi byla zvolena knihovna TensorFlow [1], která je v dnešní době velmi populární a poskytuje možnost akcelerace výpočtů na GPU.

6.1 TensorFlow

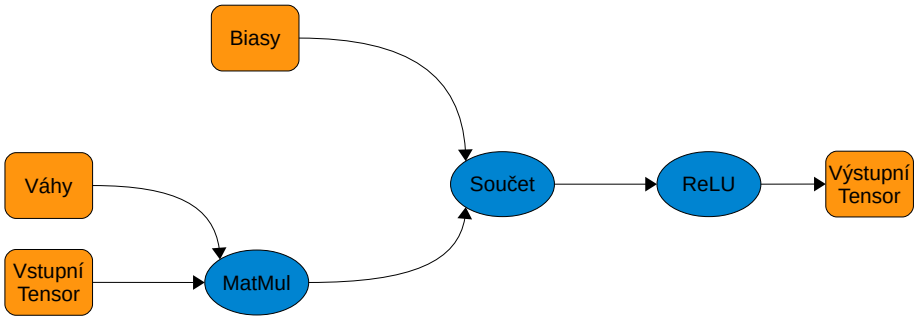
Knihovna TensorFlow byla vyvinuta týmem Google Brain. Tato knihovna poskytuje rozhraní pro manipulaci s výpočetním grafem, nejčastěji pro práci s neuronovými sítěmi. TensorFlow poskytuje dvě základní abstrakce pro práci s výpočetním grafem. První jsou tenzory, které představují n-rozměrná data. Příklad tenzorů je uveden na obrázku 6.1.



Obrázek 6.1: Ukázka různých tenzorů.

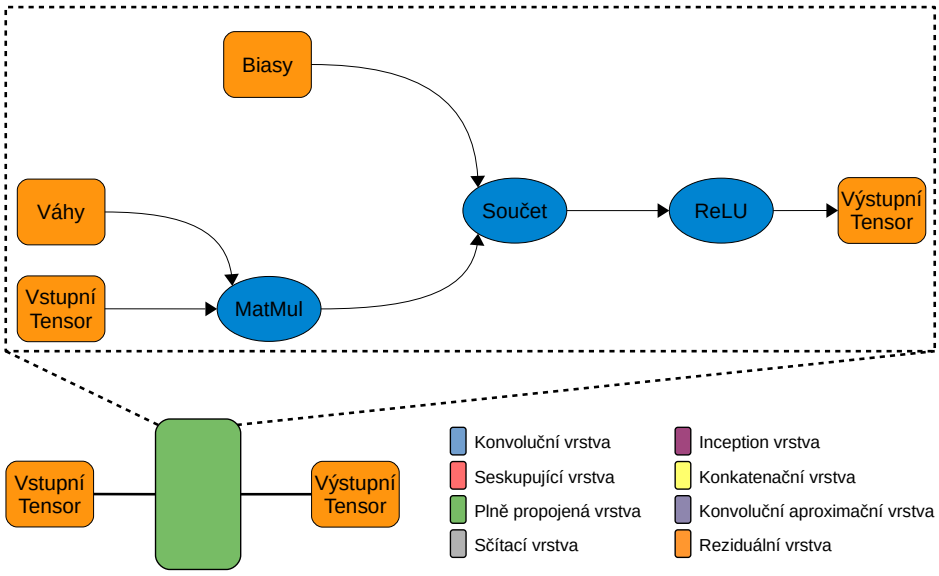
Druhou abstrakcí jsou výpočetní uzly realizující určité operace. Vstupem a výstupem výpočetních uzlů jsou tenzory. Jednoduchý příklad výpočetního grafu je uveden na obrázku 6.2, kde oranžové uzly reprezentují tenzory a modré uzly představují operace. Uvedený pří-

klad realizuje dopředný výpočet jedné vrstvy umělých neuronů. Vstupní tenzor je vynásoben s váhovým tenzorem pomocí operace násobení matic (MatMul). K výsledku násobení matic je připočítán tenzor s předpětími jednotlivých neuronů (biasy) a na výsledek je aplikována aktivační funkce ReLU. Výstupní tenzor pak obsahuje výstupy jednotlivých neuronů dané vrstvy, který slouží jako vstup další vrstvě.



Obrázek 6.2: Příklad jednoduchého výpočetního grafu. Výstupní tenzory výpočetních uzlů *MatMul* a *Součet* zde nejsou naznačeny.

Pomocí výpočetních grafů lze implementovat různé vrstvy konvolučních neuronových sítí, jak je ukázáno na obrázku 6.3.

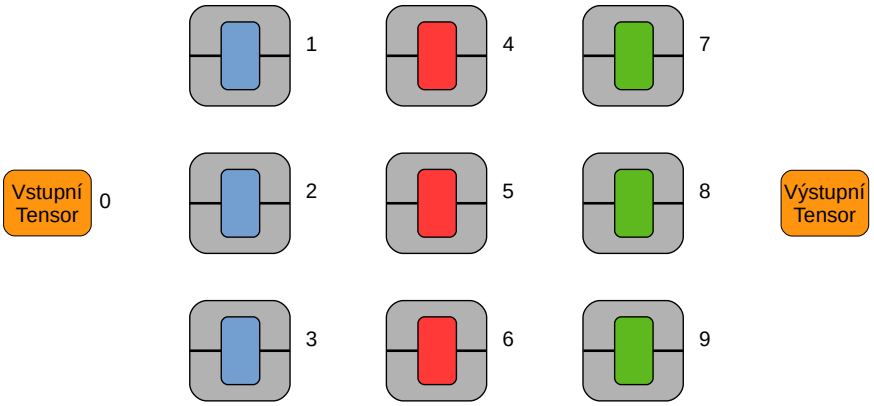


Obrázek 6.3: Příklad implementace plně propojené vrstvy pomocí TensorFlow výpočetního grafu.

6.2 Využití CGP pro návrh CNN

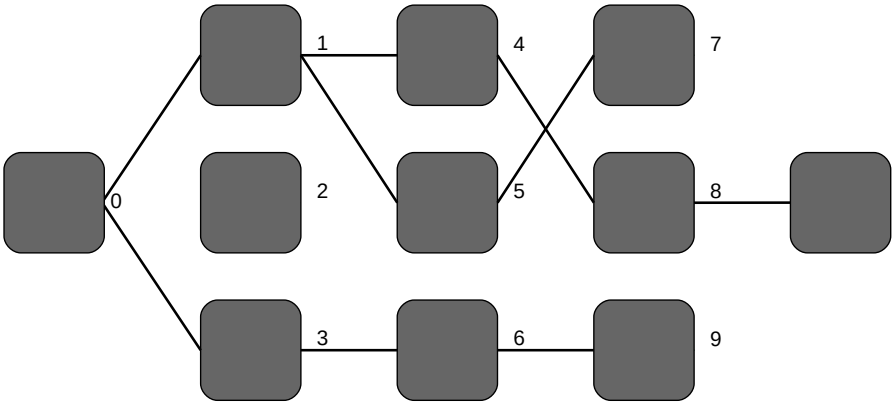
Jelikož CGP pracuje s orientovanými acyklickými grafy, tak je nutné abstrahovat práci s jednotlivými vrstvami CNN jako práci s DAG. Toho je docíleno vytvořením abstraktní vrstvy pro práci s grafem, jehož uzly uvnitř implementují jednotlivé vrstvy CNN. Tato skutečnost je znázorněna na obrázku 6.4, kde lze vidět klasickou výpočetní mřížku CGP

s očíslovanými výstupy jednotlivých uzlů. Každý výpočetní uzel uvnitř pomocí knihovny TensorFlow implementuje určitou vrstvu neuronové sítě. Vstup a výstup CGP zde zastupují vstupní a výstupní tenzory.



Obrázek 6.4: Abstrakce vrstev konvoluční neuronové sítě jako uzlů výpočetní mřížky CGP. Různé barvy zde označují různé vrstvy – konvoluční (modrá), seskupující (červená) či plně propojená (zelená).

CGP pracuje nad obecným grafem, přičemž se vůbec nezajímá o to, co jednotlivé uzly grafu představují. Tato skutečnost je naznačena na obrázku 6.5, kde si lze všimnout, že CGP pracuje s uzly grafu jako s tzv. *blackboxy* (nezajímá se o jejich vnitřní implementaci).

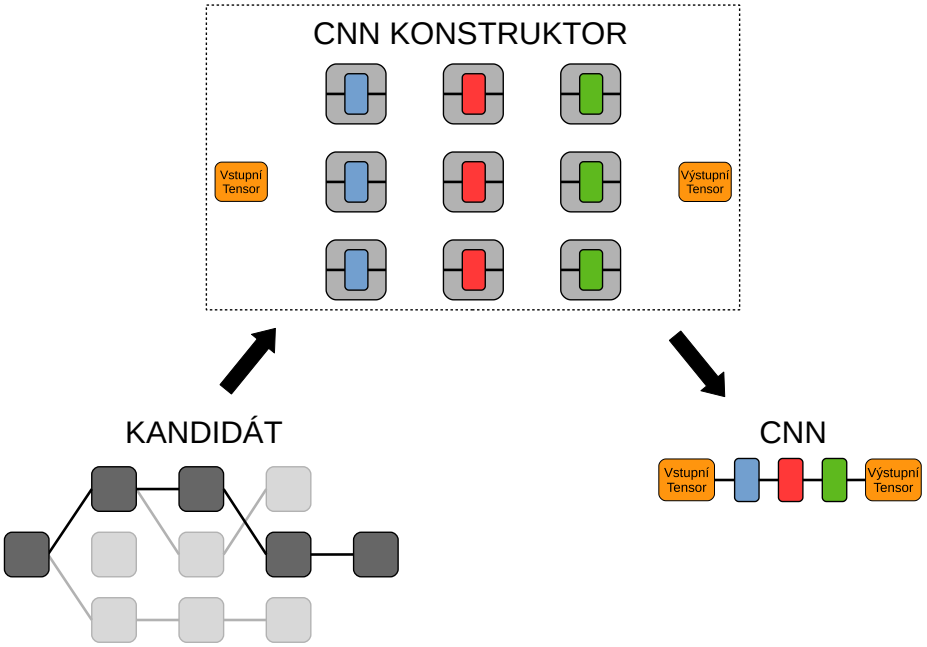


Obrázek 6.5: Ukázka grafu z pohledu CGP, který jej vidí jako obyčejný graf.

Poté, co CGP nalezne nějaké kandidátní řešení, v podobě grafu, spojujícího vstupní uzel s výstupním, je nutné převést tuto grafovou reprezentaci kandidátního řešení na CNN. Za to je zodpovědný *CNN konstruktor*, který jako vstup bere graf, popisující nějaké kandidátní řešení, ze kterého následně vytvoří odpovídající CNN. To je ukázáno na obrázku 6.6.

6.3 Speciální vrstvy a moduly

Kromě základních vrstev modelu CNN, popsaných v sekci 4.1, program implementuje i složitější výpočetní celky, které se již osvědčily a mohou tak výrazně vylepšit či urychlit automatizované hledání architektur CNN.



Obrázek 6.6: CNN konstruktor, který z grafového popisu kandidátního řešení vytváří odpovídající CNN.

První dvě popisované vrstvy představují moduly, realizující binární operace sčítání a konkatenace. Tyto moduly, ačkoliv jsou relativně jednoduché, dovolují vznik tzv. *skip* propojení. To jsou taková propojení vrstev, která přeskakují některé ostatní vrstvy modelu. Přínos takovýchto propojení spočívá v tom, že při zpětném šíření chyby (popsaném v sekci 3.3) umožňují lepší propagaci chyby do předchozích vrstev a tím řeší problém spojený s mizejícím gradientem (popsaném v podsekci 3.5).

Další dva implementované moduly, *Reziduální* a *Inception*, jsou založeny na již existujících a v praxi osvědčených přístupech použitých v modelech DenseNet [27] a GoogLeNet [61].

6.3.1 Sčítací vrstva

Vstupem sčítacího modulu jsou dva tenzory t_1 a t_2 velikosti $shape(t_1) = (h_1, w_1, c_1)$ a $shape(t_2) = (h_2, w_2, c_2)$, kde h_x značí výšku, w_x šířku a c_x počet kanálů daného tenzoru $x \in \{1, 2\}$. Výstupem je tenzor t_o, představující součet tenzorů t_1 a t_2 po jednotlivých složkách, tedy

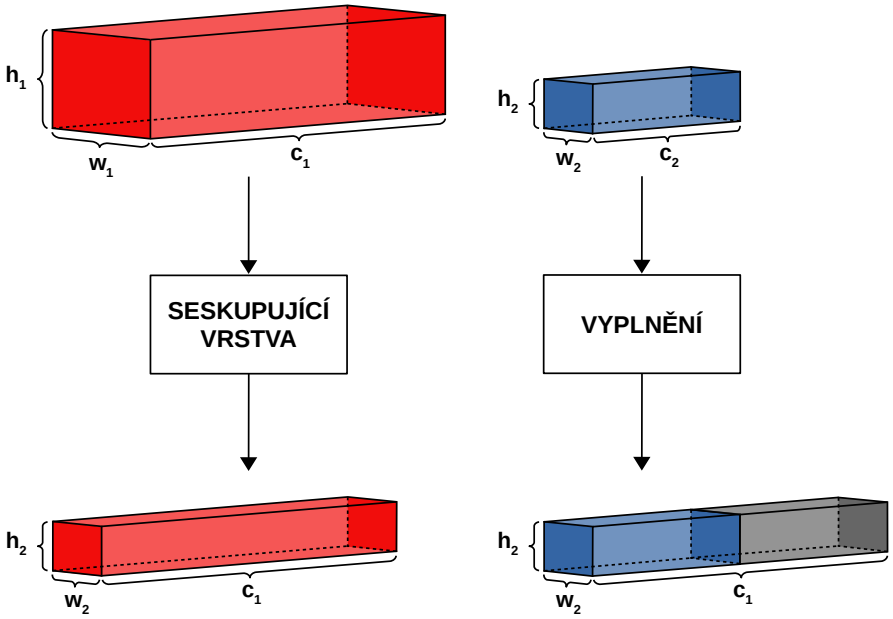
$$t_o = t_1 + t_2 \iff t_o^{ijk} = t_1^{ijk} + t_2^{ijk} \text{ pro } \begin{aligned} i &= 0, ..., c-1 \\ j &= 0, ..., h-1 \\ k &= 0, ..., w-1. \end{aligned} \tag{6.1}$$

Pro správné fungování je nutné zajistit, aby byly rozměry obou vstupních tenzorů shodné. Je tedy nutné upravit výšku, šířku a počet kanálů vstupních tenzorů tak, aby se jejich rozměry shodovaly.

Pokud je výška a šířka vstupních tenzorů rozdílná, tak jsou rozměry většího tenzoru redukovány na rozměry menšího tenzoru pomocí seskupující vrstvy s vhodnými parametry. Pro jednoduchost je uvažováno, že se výška a šířka všech tenzorů v celém modelu shoduje.

Toho je docíleno tím, že všechny tenzory, ať už se jedná o vstupní obrázek nebo filtry, jsou definovány jako čtvercové. V případě, že se neshoduje počet kanálů vstupních tenzorů, tak je tenzor s menším počtem kanálů vyplněn nulovými hodnotami tak, aby se shodoval s počtem kanálů většího tenzoru. Grafická reprezentace transformace vstupních tenzorů je uvedena na obrázku 6.7.

Pro výstupní tenzor t_o tedy platí $shape(t_o) = (h_o, w_o, c_o)$, kde $h_o = min(h_1, h_2)$, $w_o = min(w_1, w_2)$ a $c_o = max(c_1, c_2)$. Grafická ilustrace součtu dvou tenzorů je uvedena na obrázku 6.8a.



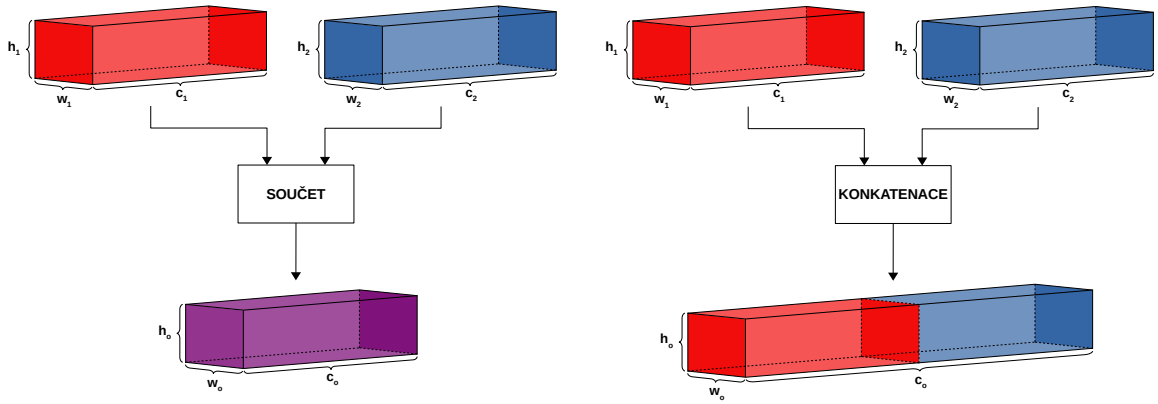
Obrázek 6.7: Ukázka transformace dvou vstupních tenzorů různé velikosti na stejnou velikost. Výška a šířka většího (červeného) tenzoru je vždy redukována, pomocí seskupující vrstvy, na velikost menšího (modrého) tenzoru. Počet kanálů menšího tenzoru je vždy doplněn nulami tak, aby se počet kanálů menšího tenzoru shodoval s počtem kanálů většího tenzoru.

6.3.2 Konkatenační vrstva

Konkatenační modul provádí konkatenaci vstupních tenzorů t_1 a t_2 velikosti $shape(t_1) = (h_1, w_1, c_1)$ a $shape(t_2) = (h_2, w_2, c_2)$, kde h_x značí výšku, w_x šířku a c_x počet kanálů daného tenzoru $x \in \{1, 2\}$. Výstupem je tenzor t_o, představující konkatenaci tenzorů t_1 a t_2 na úrovni kanálů, tedy podle 3. dimenze. Grafická ilustrace konkatenace dvou tenzorů je uvedena na obrázku 6.8b.

Pro správné fungování v tomto případě stačí zajistit, aby se šířka a výška obou vstupních tenzorů shodovala. Toho je dosaženo, stejně jako v případě sčítací vrstvy, použitím seskupující vrstvy s vhodnými parametry, díky čemuž je provedena redukce rozměrů většího tenzoru na rozměry menšího tenzoru.

Pro výstupní tenzor t_o pak platí $shape(t_o) = (h_o, w_o, c_o)$, kde $h_o = min(h_1, h_2)$, $w_o = min(w_1, w_2)$ a $c_o = c_1 + c_2$.



(a) Diagram součtu dvou vstupních tenzorů. 6.3.1.

(b) Diagram konkatenace dvou vstupních tenzorů.

Obrázek 6.8: Ilustrace součtu a konkatenace dvou tenzorů, které byly převedeny na stejnou velikost.

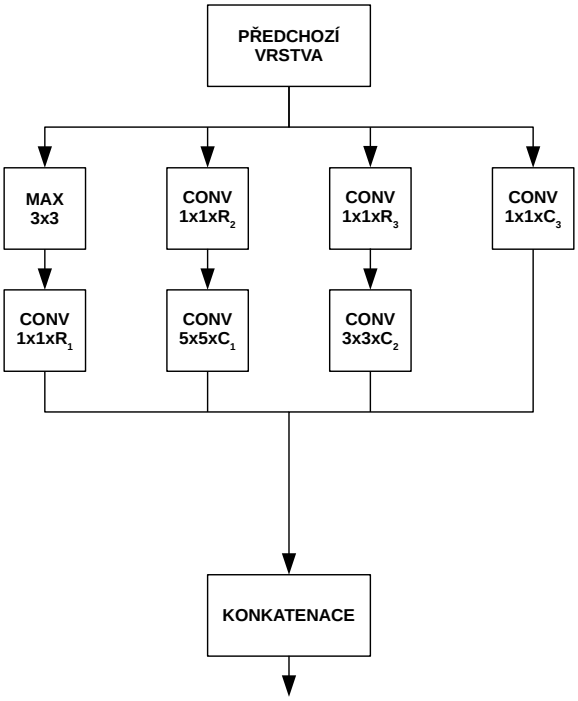
6.3.3 Inception modul

Tento modul je založen na využití více konvolučních vrstev na stejné úrovni. Velikost sítě v tomto modulu tak roste více do šířky než do hloubky. Paralelně jsou zde tak počítány konvoluce s velikostmi filtrů 5×5, 3×3 a 1×1, přičemž je do výpočtu přidána i 3×3 seskupující vrstva. Výstupy těchto vrstev jsou na konec konkatenovány podél 3. dimenze, čímž je získán výstup. Architektura tohoto modulu je uvedena na obrázku 6.9a. Z optimalizačních důvodů jsou před konvoluční vrstvy s velikostmi filtrů 3×3 a 5×5 (a za 3×3 seskupující vrstvu) umístěny konvoluční vrstvy s velikostí filtru 1×1, které provádějí redukci počtu vstupních kanálů.

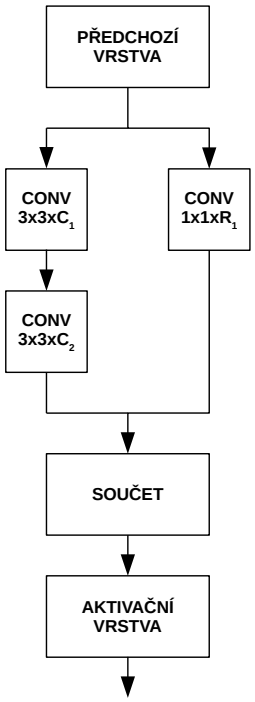
Parametry tohoto modulu jsou rozděleny do dvou skupin, kde první skupinu tvoří parametry C_1, C_2 a C_3, které udávají počet výstupních kanálů konvolučních vrstev s filtry velikosti 5×5, 3×3 a 1×1. Druhou skupinu parametrů tohoto modulu představují parametry R_1, R_2 a R_3, jejichž nastavením lze určit míru redukce počtu vstupních kanálů. Konvoluční vrstvy mají nastavenu velikost kroku na 1 a používají ReLU aktivační funkci.

6.3.4 Reziduální modul

Hlavní roli v tomto modulu hraje využití skip propojení. Tento modul v sobě ukrývá dvě konvoluční vrstvy, s velikostí filtru 3×3, v řadě za sebou. První konvoluční vrstva využívá ReLU aktivační funkci, přičemž druhá vrstva používá aktivační funkci v podobě identity. Výstup druhé konvoluční vrstvy je přiveden na vstup sčítací vrstvy, jejíž druhý vstup je tvořen výstupem redukční konvoluční vrstvy. Ta, podobně jako u Inception modulu, slouží pro redukci počtu vstupních kanálů. Redukční konvoluční vrstva používá velikost filtru 1×1 a aktivační funkci v podobě identity. Výstup tohoto modulu je získán aplikací ReLU aktivační funkce na výstup sčítací vrstvy. Architektura tohoto modulu je uvedena na obrázku 6.9b.



(a) Architektura Inception modulu. 6.3.1.



(b) Architektura reziduálního modulu. 6.3.1.

Obrázek 6.9: Diagramy architektur parametrizovatelných modulů.

6.4 Multikriteriální optimalizace

Úloha nalezení optimální architektury CNN z pohledu přesnosti a počtu parametrů představuje *multikriteriální optimalizační úlohu*, tedy úlohu s cílem současně optimalizovat více než jednu účelovou funkci (v našem případě přesnost klasifikace a počet parametrů).

Uvažujme účelovou funkci pro výpočet přesnosti

$$f_1(x) = acc(x) \tag{6.2}$$

a účelovou funkci pro výpočet počtu parametrů

$$f_2(x) = \frac{1}{\log_2(params(x) + 2)}, \tag{6.3}$$

kde $x \in \Omega$ označuje nějaké řešení v podobě CNN z prostoru řešení Ω, přičemž $acc(x)$ značí přesnost a $params(x)$ počet parametrů nějakého řešení x. Vztah 6.3 byl zvolen tak, aby se obor hodnot obou účelových funkcí pohyboval v intervalu $[0, 1]$.

Cílem multikriteriálních optimalizačních metod je nyní maximalizovat (nebo minimalizovat) obě účelové funkce, neboli nalézt takové řešení x, pro které jsou hodnoty obou účelových funkcí maximální (nebo minimální). Pro nejlepší řešení x' v našem případě očividně platí $acc(x') = 1$ a $params(x') = 0$, z čehož plyne, že nejlepším řešením je CNN se 100% přesností a 0 parametry. V tomto případě se tedy hodnoty obou účelových funkcí snažíme maximalizovat. Najít takovéto řešení je ale pro multikriteriální optimalizační metody nemožné, uvědomíme-li si, že účelové funkce jsou *konfliktní*. S rostoucí účelovou funkcí f_1 (přesností) totiž roste i počet parametrů a účelová funkce f_2 tak klesá. Naopak s rostoucí

hodnotou účelové funkce f_2 (a klesajícím počtem parametrů) klesá hodnota účelové funkce f_1 (přesnost).

V praxi existuje více optimálních řešení, která jsou navzájem objektivně neporovnatelná, a tak výstupem multikriteriálních optimalizačních metod bývá množina řešení, nazývaná *Pareto fronta* P, pro kterou platí

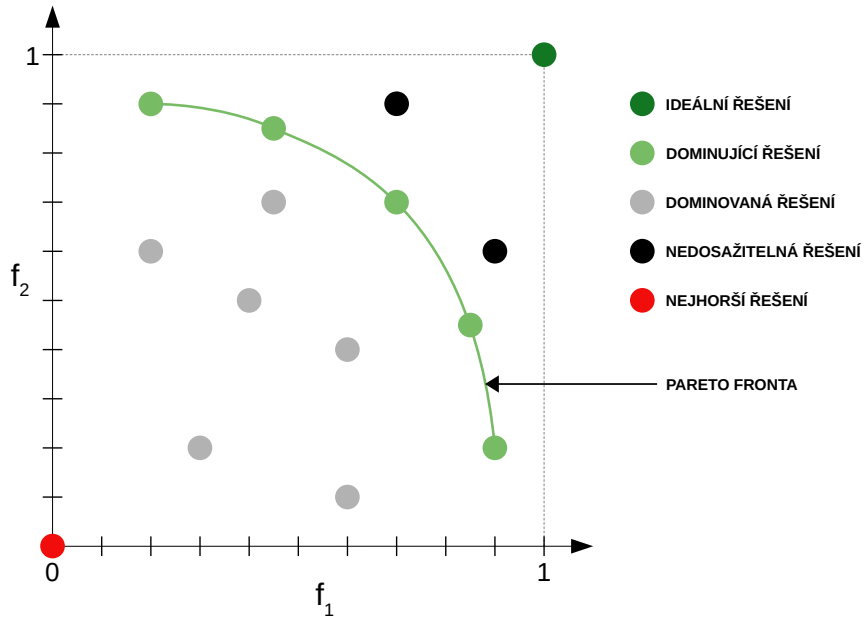
$$P = \{x \in \Omega \mid \nexists x^* \in \Omega : x^* \succ x\}, \tag{6.4}$$

kde $x^* \succ x$ značí relaci *Pareto dominance*, která je definována jako

$$\begin{aligned} x^* \succ x \iff & (f_1(x^*) \geq f_1(x) \wedge f_2(x^*) \geq f_2(x)) \wedge \\ & (f_1(x^*) > f_1(x) \vee f_2(x^*) > f_2(x)). \end{aligned} \tag{6.5}$$

Pokud $x^* \succ x$, pak říkáme, že x^* dominuje x nebo x je dominováno x^*.

Pareto fronta tedy podle vzorce 6.4 obsahuje pouze taková řešení, která nejsou dominována, a proto se často označují jako *Pareto optimální* nebo *Pareto dominující* řešení. Obrázek 6.10 graficky znázorňuje Pareto frontu.



Obrázek 6.10: Grafická ilustrace Pareto fronty, pokud je cílem maximalizovat účelové funkce f_1 a f_2.

6.5 Evoluční algoritmus

V této práci byl jako evoluční algoritmus zvolen vícekriteriální optimalizační genetický algoritmus *NSGA-II* (Non-Dominated Sorting Genetic Algorithm 2) [8], který je založen na hledání optimálních jedinců s využitím uspořádání podle relace Pareto dominance. Algoritmus NSGA-II, použitý v této práci, je uveden v algoritmu 3.

První dva kroky algoritmu se skládají z úvodní inicializace populace a jejího ohodnocení. Algoritmus poté vstupuje do cyklu, který obsahuje jednotlivé kroky prováděné v každé generaci. Na počátku je ze staré populace vytvořena množina potomků O. Pro tyto účely

byl zvolen postup, který z každého jedince staré populace vytvoří jeho kopii, na kterou následně aplikuje operaci mutace, a proto platí $|O| = |P|$. Mutace potomka probíhá do té doby, dokud se od rodiče neliší alespoň v jednom aktivním uzlu. Množina potomků O je vzápětí ohodnocena a množiny jedinců staré populace P a jejich potomků O jsou sjednoceny, čímž vytvářejí smíšenou množinu jedinců R, pro kterou platí $|R| = |P| + |O|$.

Hlavní změnu oproti obecnému evolučnímu algoritmu, uvedeném v kapitole 2.2, představuje cyklus na řádku 10, který je zodpovědný za tvorbu nové populace. V každé iteraci tohoto cyklu je z množiny R extrahována Pareto fronta PF, tedy množina dominujících řešení. Pokud se jedinci z Pareto fronty vejdou do nové populace, tak jsou do ní rovnou přidání. V opačném případě je potřeba z množiny PF odebrat ty jedince, kteří se do nové populace již nevejdou. To je implementováno funkcí $crowding\_reduce(PF, n)$, která na základě tzv. *crowding distance*, popsané v podsekci 6.5.3, z množiny PF eliminuje n jedinců, kteří jsou v prostoru řešení moc blízko u sebe.

Algoritmus končí po splnění ukončujících podmínek, v tomto případě po dosažení maximálního počtu generací. Výstupem tohoto algoritmu je zpravidla množina jedinců, která je tvořena jedinci v Pareto frontě populace poslední generace. Grafická ukázka tohoto algoritmu je uvedena na obrázcích 6.13 a 6.14.

Algorithm 3 Evoluční algoritmus

```
1:  P ← initial_population()
2:  evaluate(P)
3:  g ← 0
4:  repeat
5:      P' ← replicate(P)
6:      O ← mutate(P')
7:      evaluate(O)
8:      R ← P ∪ O
9:      P ← ∅
10:     while |P| ≠ population_size do
11:         PF ← non_dominated(R)
12:         if |P ∪ PF| ≤ population_size then
13:             P ← P ∪ PF
14:         else
15:             n ← |PF ∪ P| − population_size
16:             P ← P ∪ crowding_reduce(PF, n)
17:         R ← R \ PF
18:     g ← g + 1
19: until stop_criteria_satisfied()
```

6.5.1 Kódování jedinců

Implementovaný program pro zakódování jedinců nevyužívá klasické kódování jako sekvenci celých čísel, jak je tomu u většiny genetických algoritmů. V tomto případě je jedinec reprezentován přímo jako graf $G = (N, E)$, kde N značí množinu uzlů (vrcholů) a $E = \{(u, v) \mid u \in N \wedge v \in N\}$ množinu hran. Toto rozhodnutí bylo motivováno tím, že reprezentace jedince ve formě grafu je vyhovující a konverze do jiné reprezentace by byla zbytečná.

6.5.2 Mutace

Mutace je prováděna přímo na grafu G, přičemž při mutaci dochází pouze ke změně propojení uzlů. V terminologii genetického programování to znamená, že mutace mění pouze propojovací a výstupní geny chromozomu. Funkční geny zůstávají nezměněny.

Mutace probíhá tak, že je náhodně vybrán uzel k mutaci $n \in N$. Pro uzel n jsou odstraněny všechny jeho vstupní hrany, tedy $E = E \setminus \{(x, n) \mid x \in N\}$. Proces přidání nové hrany následně obnáší výpočet množiny

$$L(n) = \{u \mid u \in N \text{a } u \text{ je uzel respektující parametr l-back pro vybraný uzel } n\},$$

ze které je náhodně vybrán uzel $m \in L(n)$. Do množiny hran E je poté přidána nová hrana (m, n), tedy $E = E \cup (m, n)$. Pokud uzel n reprezentuje funkci s vyšší aritou než je 1, tak je proces přidání nové hrany opakován, přičemž uzly, které jsou již připojeny k uzlu n, jsou z množiny $L(n)$ odstraněny.

6.5.3 Fitness funkce

Z algoritmu 3 plyne, že v případě, kdy se všichni jedinci Pareto fronty PF již nevejdou do nové populace, tak je nutné provést selekci nejvhodnějších jedinců z množiny PF. Pro výběr nejvhodnějších jedinců je použita fitness funkce, označována jako *crowding distance* (shluková vzdálenost), která jedincům přiřazuje hodnotu podle jejich relativní vzdálenosti od ostatních jedinců. Pro výpočet shlukové vzdálenosti D jedince x_j, pro $j \in 1, ..., |PF| - 2$, od sousedních jedinců x_{j-1} a x_{j+1} je použit vzorec

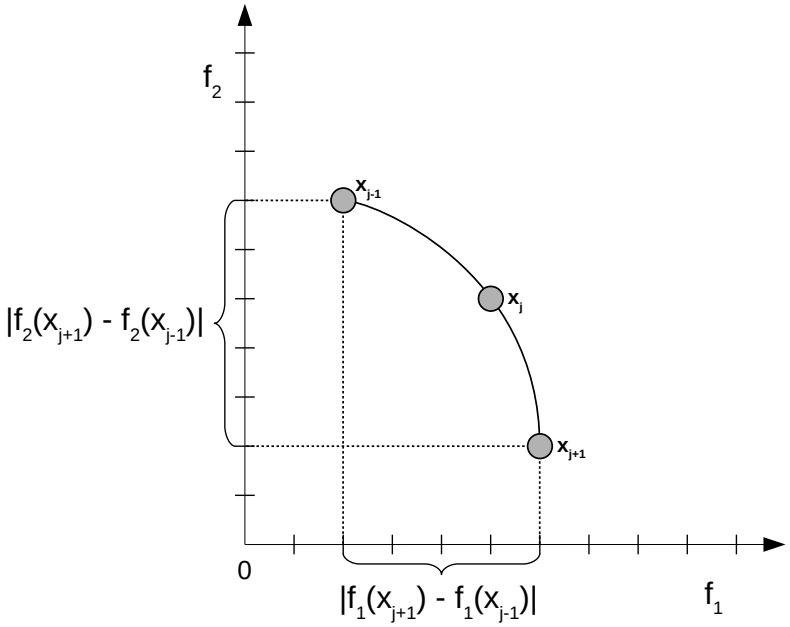
$$D(x_j) = \sum_{o=1}^{N} \frac{d_o^j}{max(f_o) - min(f_o)}, \tag{6.6}$$

kde N značí počet účelových funkcí, d_o^j označuje vzdálenost jedince x_j od sousedních jedinců při použití účelové funkce f_o a jmenovatel obsahuje výpočet rozdílu maximální a minimální hodnoty účelové funkce f_o. Výpočet vzdálenosti d_o^j je realizován vzorcem $|f_o(x_{j+1}) - f_o(x_{j-1})|$ a při použití účelových funkcí 6.2 a 6.3 získáme finální vzorec

$$D(x_j) = \frac{|f_1(x_{j+1}) - f_1(x_{j-1})|}{max(f_1) - min(f_1)} + \frac{|f_2(x_{j+1}) - f_2(x_{j-1})|}{max(f_2) - min(f_2)}. \tag{6.7}$$

Pro jedince na okrajích Pareto fronty PF platí $x_0 = x_{|PF|-1} = \infty$, z čehož plyne, že jedinci na okraji Pareto fronty budou vždy vybráni (pokud se samozřejmě oba do nové populace vejdou, pokud ne, tak je vybrán jeden z nich). Grafická reprezentace výpočtu shlukové vzdálenosti je uvedena na obrázku 6.11.

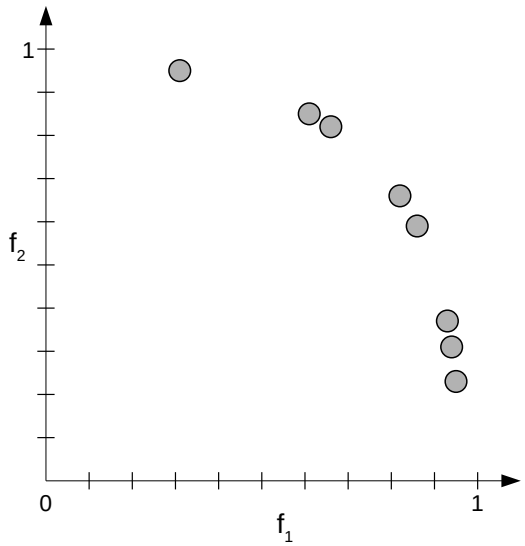
Ohodnocení jedinci jsou následně uspořádáni sestupně, podle jejich fitness hodnoty. Z takto uspořádané množiny jedinců je poté odstraněno posledních n jedinců, kteří se již do nové populace nevejdou. Tvorba nové populace je graficky znázorněna na obrázku 6.14b.



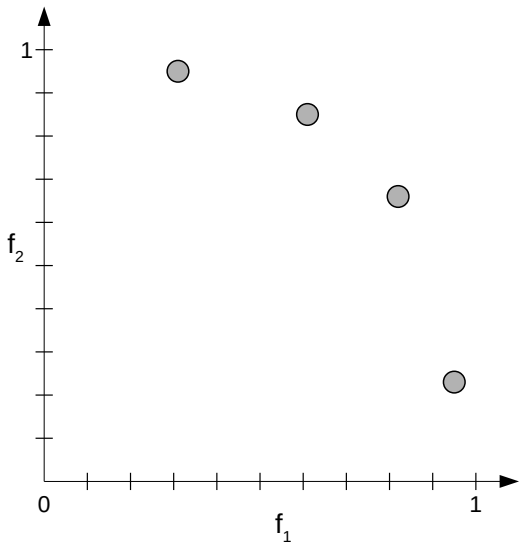
Obrázek 6.11: Grafická ilustrace výpočtu shlukové vzdálenosti pro jedince x_j

6.5.4 Zachování rozmanitosti

Výpočet shlukové vzdálenosti hraje v NSGA-II důležitou roli z pohledu zachování diverzity jedinců v populaci. To je ukázáno na obrázku 6.12, kde je znázorněna situace, kdy je z populace 8 jedinců v jedné Pareto frontě (obrázek 6.12a) nutné vybrat 4 nejlepší jedince. Díky použití shlukové vzdálenosti jako fitness funkce, je provedena selekce jedinců, kteří se nenacházejí příliš blízko u sebe (obrázek 6.12b).

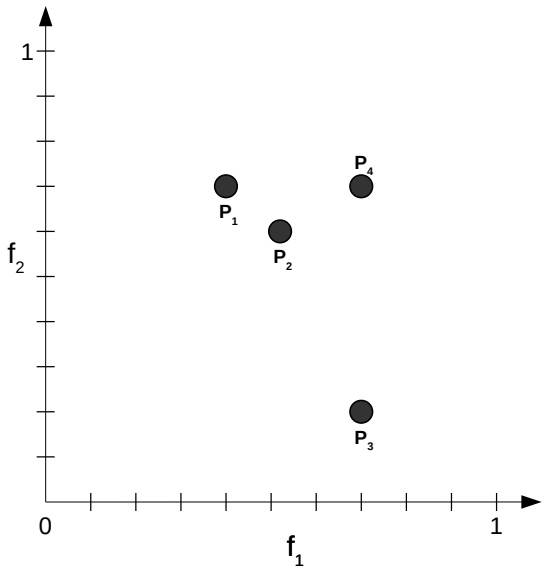


(a) Jedinci před selekcí pomocí shlukové vzdálenosti.

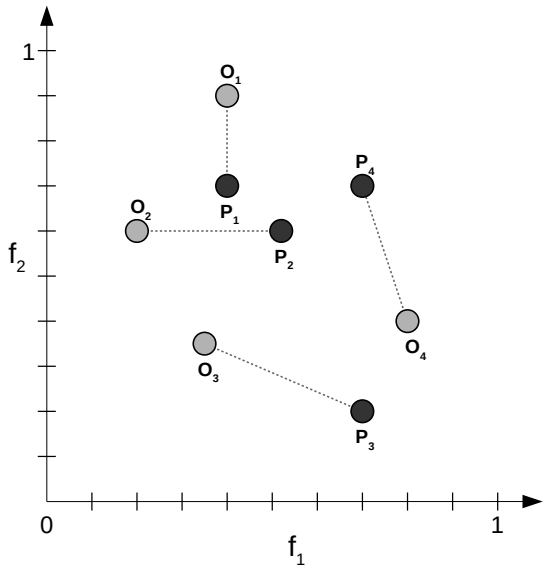


(b) Vybraní jedinci po aplikaci shlukové vzdálenosti.

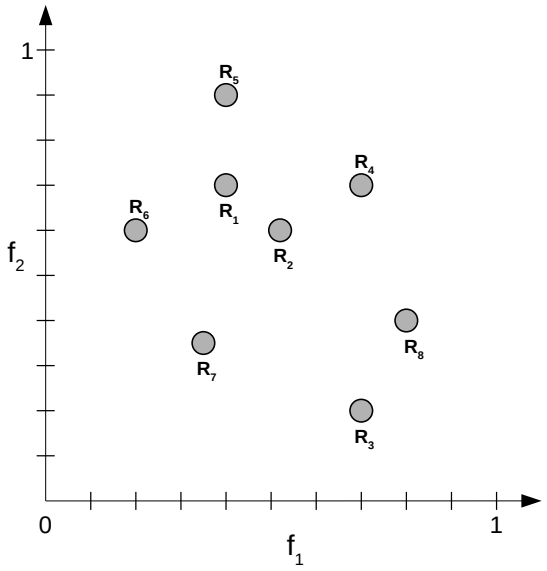
Obrázek 6.12: Ilustrace vlivu selekce jedinců pomocí shlukové vzdálenosti na diverzitu populace.



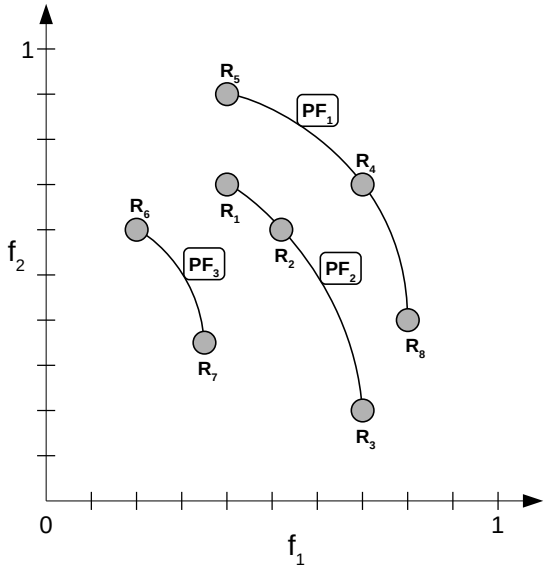
(a) Ilustrace staré populace P, kde umístění jednotlivých jedinců v prostoru odpovídá hodnotám jejich účelových funkcí f_1 a f_2.



(b) Vytvoření potomků z původní populace, kde O_n značí zmutovanou kopii P_n pro $n \in \{1, 2, 3, 4\}$.

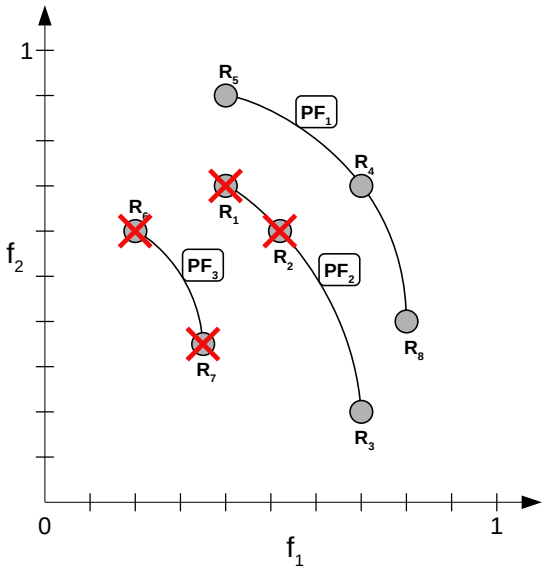


(c) Vytvoření smíšené populace, skládající se z jedinců staré populace a jejich potomků.

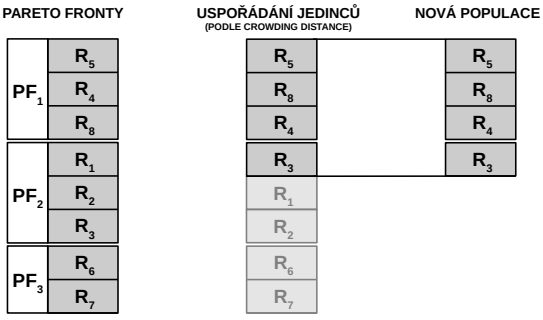


(d) Grafická ilustrace zobrazující vytvoření jednotlivých Pareto front.

Obrázek 6.13: Grafická reprezentace evolučního algoritmu 3. Uvedený příklad ilustruje postup algoritmu pro velikost populace 4.



(a) Grafická reprezentace jednotlivých Pareto front.



(b) Diagram znázorňující selekci jedinců, kteří budou tvořit novou populaci.

Obrázek 6.14: Ilustrace tvorby nové populace z postupných Pareto front PF_1, PF_2 a PF_3. Jedinci, kteří se již do nové populace nevejdou jsou odmítnuti, což je graficky naznačeno červenými křížky. V případě PF_2 byli odmítnuti jedinci R_1 a R_2, jelikož jejich shluková vzdálenost byla moc malá.

6.6 Dědičnost vah

Jak již bylo popsáno v genetickém evolučním algoritmu 3, tak tvorba nových potomků ve všech generacích je zajištěna pomocí replikace a mutace každého jedince v současné populaci, jak je zachyceno na řádcích $5 - 6$ zmíněného algoritmu. Potomci jsou z rodičů vytvořeni pomocí mutace, která zajistí, že potomek se bude od rodiče lišit alespoň v jednom aktivním uzlu, z čehož plyne, že potomek může (a s velkou pravděpodobností i bude) sdílet některé výpočetní uzly CGP mřížky s jeho rodičem. V tomto případě může být užitečné, aby potomek od rodiče zdědil naučené váhy pro tyto sdílené uzly a nemusel začínat od začátku s náhodnou inicializací vah.

Dědičnost vah má omezení v tom, že váhy jsou děděny pouze pro ty uzly CGP mřížky, které jsou využívány jak rodičem, tak jeho potomkem. Překážku dále tvoří fakt, že ačkoliv je uzel používán rodičem i potomkem, tak v obou případech může brát vstup od různých CGP uzlů, důsledkem čehož váhové tenzory nemusejí rozměrově souhlasit. V tomto případě jsou zděděné váhové tenzory ořezány nebo naopak vycpány náhodnými hodnotami tak, aby rozměrově souhlasily s rozměry váhových tenzorů potomka.

Kapitola 7

Implementace a použití

Tato kapitola obsahuje informace o implementaci programu, navrženého v kapitole 6, a jeho použití. Dále se zde nachází popis knihoven a prostředků použitých při tvorbě tohoto programu.

Program je implementován v jazyce Python, přičemž pro práci s neuronovými sítěmi byla zvolena knihovna TensorFlow. Ta patří mezi jednu z nejpopulárnějších a nejvýkonnějších knihoven pro práci s neuronovými sítěmi. Hlavní výhodou této knihovny je, že podporuje akceleraci procesu učení na grafických akcelerátorech (GPU), což byla jedna z hlavních motivací použití této knihovny v této práci.

7.1 Rozdělení programu

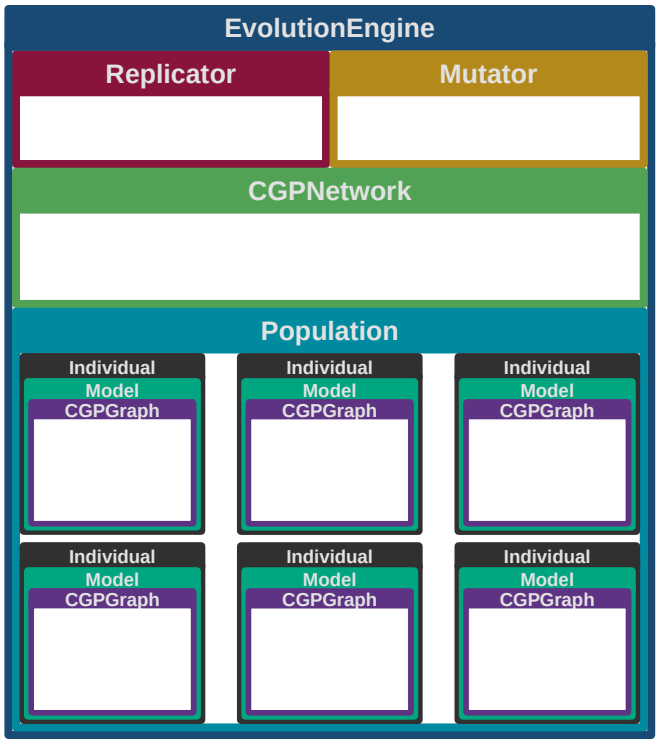
Hlavní část tohoto programu tvoří objekt `EvolutionEngine`, který obaluje a řídí celý proces evoluce. Uvnitř tohoto objektu jsou definované 4 základní pomocné objekty, které jsou použity v průběhu evoluce. Prvním z nich je objekt `Individual`, reprezentující jednoho jednotlivce v populaci. Druhým pomocných objektem je `Replicator`, který provádí replikaci jedinců, která je použita před operací mutace. Dalším důležitým pomocným objektem je `Mutator`, který je zodpovědný za mutaci jedinců. Posledním významným objektem je `CGPNetwork`, který zodpovídá za celou CGP část evolučního algoritmu, obsahuje konfiguraci CGP a spravuje výpočetní mřížku, obsahující definici jednotlivých výpočetních uzlů. Hierarchie těchto objektů je zachycena na diagramu 7.1

7.1.1 Reprezentace jedince

Jak bylo popsáno v podsekci 6.5.1, tak je pro kódování jedince zvolena reprezentace v podobě grafu. Z tohoto důvodu je součástí objektu `Individual` objekt `Model` (reprezentující model konvoluční neuronové sítě představovaný daným jedincem), který obsahuje popis architektury CNN v podobě grafu, popsaného objektem `CGPGraph`. V průběhu evoluce je s jedinci zacházeno na úrovni tohoto grafu, přičemž převod na samotnou CNN je proveden až ve chvíli, kdy je potřeba jedince vyhodnotit.

7.2 Definice výpočetní mřížky

Důležitým vstupem tohoto programu je definice výpočetní mřížky CGP. Zde si může uživatel určit, které výpočetní uzly budou použity. Vhodnou volbou a rozmístěním výpočetních uzlů v CGP mřížce může totiž uživatel evolučnímu algoritmu velmi pomoci.



Obrázek 7.1: Diagram hierarchie klíčových objektů.

Samotné konstrukci výpočetní mřížky předchází definice výpočetních uzlů, které budou ve výpočetní mřížce použity. Uživatel si tak může definovat různé výpočetní uzly s různými parametry. Ukázka definice některých výpočetních uzlů je ukázána v kódu 7.1.

```
# Definice konvolucni vrstvy s~velikosti kernelu 2x2 a 64 vystupnimi kanaly
conv_2x2_64 = {
                    "func": Conv2D,
                    "channels": 64,
                    "kernel_size": [2, 2],
                    "strides": [1, 1]
                }
# Definice 2x2 max sdruzujici vrstvy
max_2x2 = {
                "func": MaxPool,
                "kernel_size": [2, 2],
                "strides": [2, 2]
            }
# Definice plne propojene vrstvy se 128 neurony
fc_64 = {
            "func": FullyConnected,
            "neurons": 64
        }
# Definice konkatenacni a scitaci vrstvy
concatenation = {"func": ConCat}
```

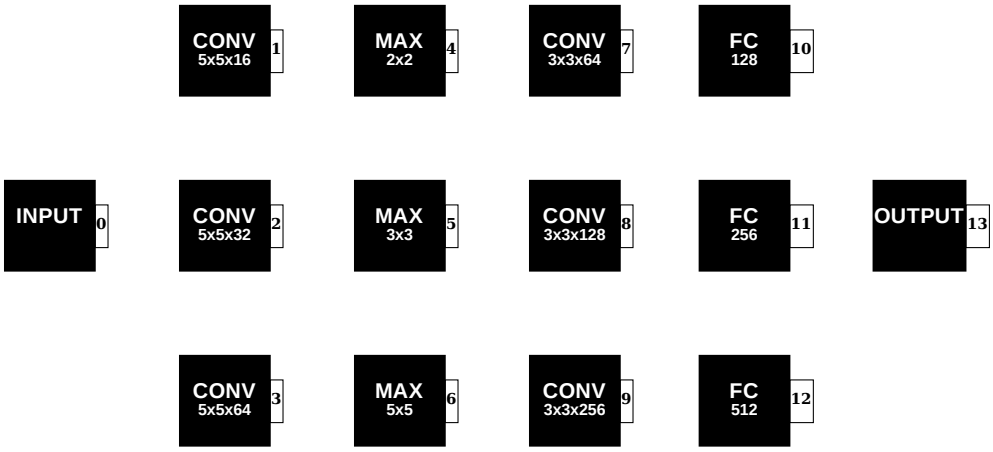
```
summation = {"func": Summation}
```

Výpis 7.1: Ukázka definice výpočetních uzlů.

Z definovaných výpočetních uzlů lze následně vytvořit výpočetní mřížku. Příklad definice jednoduché výpočetní mřížky, nazvané `BasicTemplate`, se třemi řádky a čtyřmi sloupci je uveden v kódu 7.2. Grafická ilustrace této mřížky je ukázána na obrázku 7.2.

```
BasicTemplate = []

# Prvni sloupec
BasicTemplate.append(conv_5x5_16)
BasicTemplate.append(conv_5x5_32)
BasicTemplate.append(conv_5x5_64)

#Druhy sloupec
BasicTemplate.append(max_2x2)
BasicTemplate.append(max_3x3)
BasicTemplate.append(max_5x5)

# Treti sloupec
BasicTemplate.append(conv_3x3_64)
BasicTemplate.append(conv_3x3_128)
BasicTemplate.append(conv_3x3_256)

# Ctvrty sloupec
BasicTemplate.append(fc_128)
BasicTemplate.append(fc_256)
BasicTemplate.append(fc_512)
```

Výpis 7.2: Příklad definice jednoduché výpočetní mřížky se třemi řádky a čtyřmi sloupci.



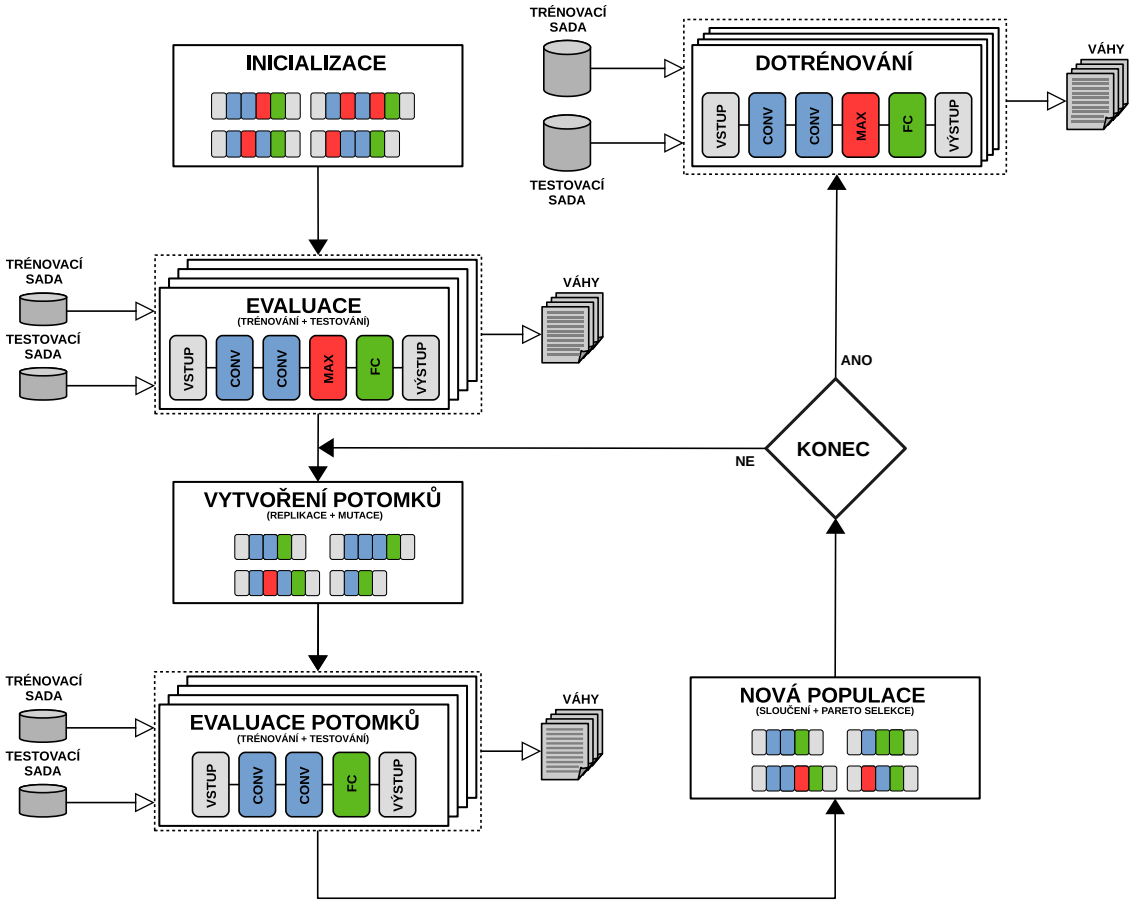
Obrázek 7.2: Ilustrace výpočetní mřížky, definované v kódu 7.2. Vstupní a výstupní uzly jsou přidány automaticky.

7.3 Průběh evoluce

Evoluční algoritmus je implementován podle algoritmu 3, uvedeném v kapitole 6. Grafická ilustrace průběhu evoluce je ukázána na obrázku 7.3.

Evaluaci každého jedince předchází sestrojení CNN modelu, jak je popsáno v sekci 6.2, konkrétně obrázkem 6.6. Samotná evaluace jedince se skládá z učení jedince na náhodně vybrané podmnožině originální trénovací sady `mini_train_dataset` po dobu `mini_train_epochs` epoch. Po skončení učení jedince je vyhodnocena jeho přesnost pomocí náhodně vybrané podmnožiny originální testovací sady `mini_test_dataset`. Evaluace jedince je završena výpočtem účelových funkcí 6.2 a 6.3, popsaných v sekci 6.4.

Z každého jedince ohodnocené populace je vytvořen jeho potomek, čehož je docíleno pomocí replikace jedince a následné mutace. Množina potomků je poté vyhodnocena. Nová populace je získána sloučením jedinců staré populace a množiny jejich potomků, ze které je s využitím Pareto selekce vytvořena nová populace, jak je znázorněno na ilustraci 6.14 v sekci 6.5 předchozí kapitoly. Výsledkem evoluce je množina nejlepších jedinců, která je určena Pareto frontou populace v poslední generaci. Nejlepší jedinci jsou po skončení evolučního algoritmu dotrénováni na originální trénovací datové sadě po dobu `final_train_epochs` epoch, přičemž finální přesnost je spočítána pro originální testovací datovou sadu.



Obrázek 7.3: Diagram průběhu evoluce.

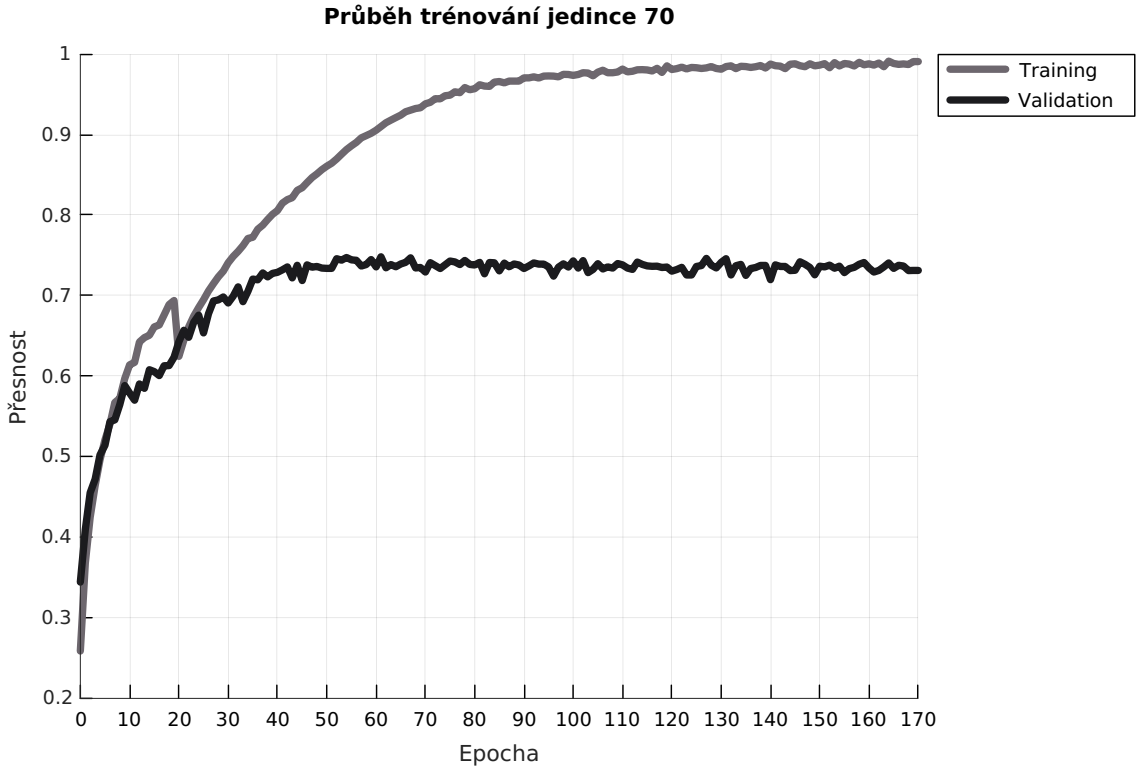
7.4 Výstupy programu

V průběhu provádění programu je ukládáno mnoho dat, která jsou určena pro pozdější vizualizaci. Mezi zaznamenávané informace patří jak průběh učení jedinců, tak i průběh evoluce.

Výstupy pro každý běh programu jsou ukládány do složky s názvem, který odpovídá času, kdy byl program spuštěn. Tato složka je dále rozdělena na tři hlavní podsložky `graphs/`, `logger/` a `TensorBoard/`.

7.4.1 Výstupy učení

Výstupem procesu učení jsou grafy, které zachycují změny trénovací a testovací přesnosti po každé epoše (tedy po jednom průchodu celé datové sady). Pro záznam těchto údajů byla použita knihovna *TensorBoard* a klasický zápis do souboru. Výstupy trénování jedinců, které se skládají z trénovacích a testovací přesností, se nacházejí ve složce `logger/`. Zaznamenané údaje o průběhu učení jedince s identifikátorem `ID` lze vykreslit pomocí volání Octave funkce `plotAccuracy(ID)`, uložené v souboru `plotAccuracy.m`. Příklad této vizualizace je uveden na obrázku 7.4.

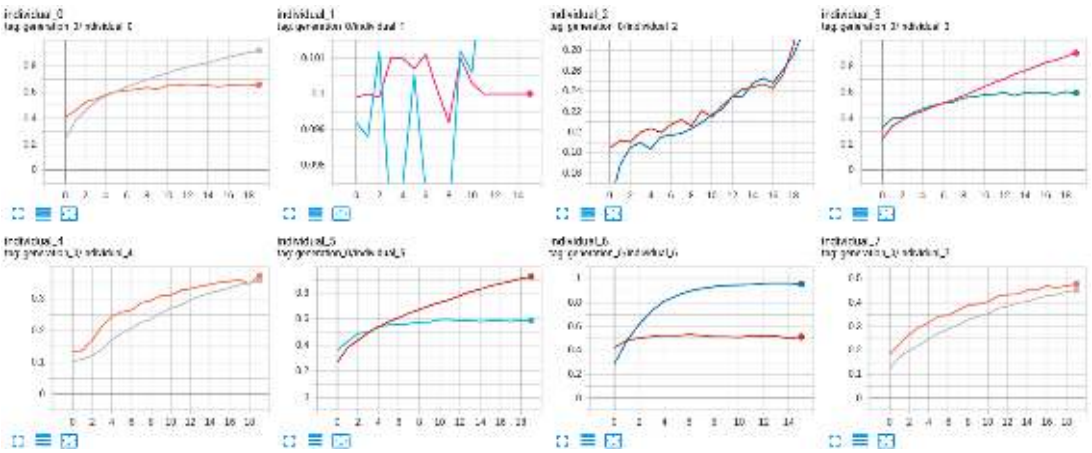


Obrázek 7.4: Ukázka vizualizace průběhu učení jedince `70` pomocí Octave skriptu. Pokles přesnosti v 20. epoše je způsoben tím, že jedinec byl prvních 20 epoch trénován na mini trénovací sadě a dalších 150 epoch na plné trénovací sadě (dotrénování).

Údaje o průběhu učení jedinců v průběhu evoluce jsou rovněž zaznamenávány pomocí knihovny TensorBoard, a to do složky `TensorBoard/`. Vizualizaci údajů o průběhu učení všech jedinců v generaci `i` lze získat spuštěním programu:

```
TensorBoard --logdir=generation_i
```

Ukázka výstupu tohoto programu je zachycena na obrázku 7.5.



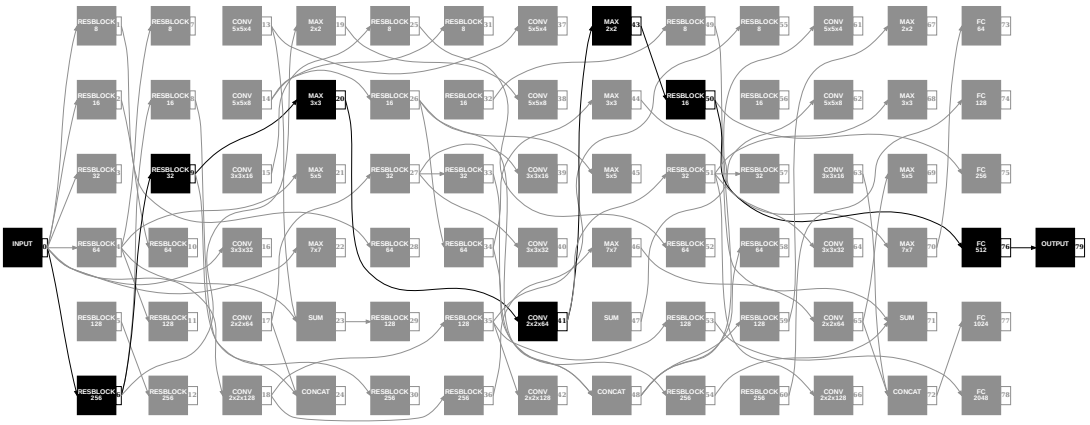
Obrázek 7.5: Ukázka vizualizace průběhu učení celé populace první generace s využitím knihovny TensorBoard.

7.4.2 Výstupy procesu evoluce

V průběhu evoluce jsou ukládány informace o architekturách jedinců, ve formě *DAG*. Pro vykreslení těchto grafů byla použita knihovna *Graphviz* [11]. Složka `graphs/` obsahuje binární soubory, ve kterých jsou uloženy informace o jednotlivých jedincích a jejich architekturách. Spuštěním programu:

```
python graph_viz.py /path/to/graphs
```

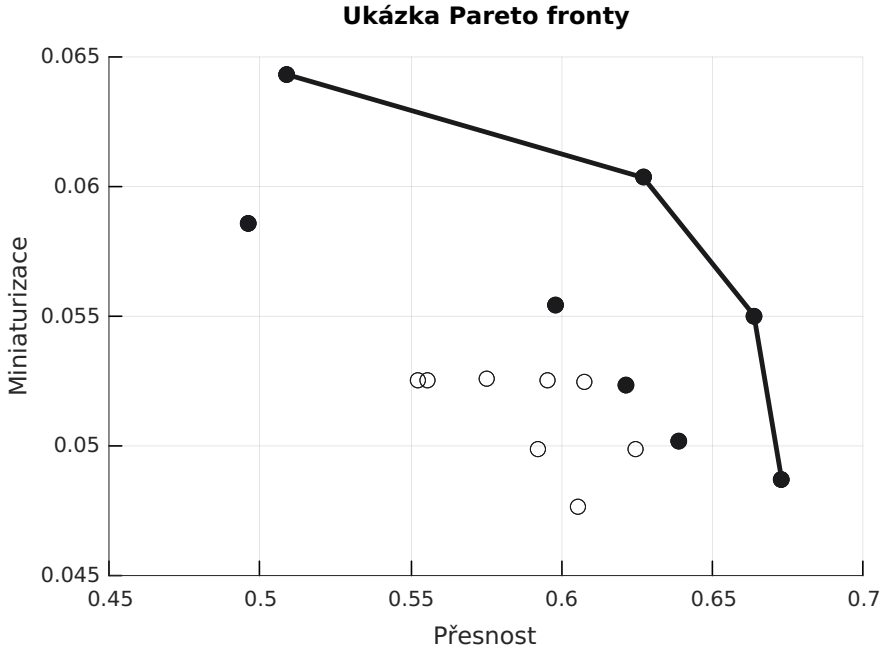
lze vytvořit vizualizaci architektury jedince, která je uvedena na obrázku 7.6.



Obrázek 7.6: Ukázka vizualizace architektury jedince pomocí knihovny Graphviz.

Při evoluci jsou rovněž ukládány hodnoty účelových funkcí každého jedince, díky čemuž lze později vykreslit grafy, zachycující Pareto fronty v jednotlivých generacích. Pro tyto účely byl použit program *Octave*. Ve složce `logger/` se nachází soubor `evolution_log`, který obsahuje informace o změnách v populaci napříč generacemi. Konkrétně jsou zde

zachyceny hodnoty účelových funkcí všech jedinců v každé generaci, přičemž záznamy v tomto souboru jsou strukturovány tak, že z nich je možné vytvořit vizualizaci v podobě grafu s jednotlivými Pareto frontami. Pro vizualizaci těchto informací slouží Octave funkce `plotPareto()`, uložená v souboru `plotPareto.m`, která vykreslí jedince do grafu, jak je ukázáno na obrázku 7.7, kde se na horizontální ose nacházejí hodnoty účelové funkce f_1 (zde označené jako *přesnost*) a na vertikální ose jsou umístěny hodnoty účelové funkce f_2 (zde označené jako *miniaturizace*).



Obrázek 7.7: Ukázka vizualizace populace v jedné generaci. Plnou čarou jsou spojeni jedinci, kteří se nacházejí v Pareto frontě dané generace. Plnými kolečky jsou označeni jedinci, kteří byli vybráni do další generace. Prázdnými kolečky pak jedinci, kteří se do další generace nedostali.

7.5 Využití aproximačních násobiček

Implementovaný program podporuje návrh architektury CNN pro zařízení, která používají aproximační výpočetní jednotky. Tyto jednotky na úkor přesnosti snižují požadavky na příkon nebo plochu na čipu vestavěných zařízení, která jsou často limitována z pohledu dostupných zdrojů. Postup návrhu architektury CNN pro zařízení, využívající aproximační výpočetní jednotky, je založen na předpokladu, že vestavěné zařízení bude obsahovat naučený model CNN, který bude používat pro inferenci (tedy jen v dopředném směru).

Pro implementaci využití aproximačních jednotek bylo použito rozšíření knihovny TensorFlow *tf-approximate* [64], založené na práci [48], na kterém pracuje výzkumná skupina *Evolvable Hardware*. Toto rozšíření implementuje konvoluční vrstvu, která pomocí 8 bitové kvantizace převádí násobení v plovoucí řádové čárce na násobení s pevnou řádovou čárkou. Pro samotné násobení v aritmetice s pevnou řádovou čárkou je použita aproximační násobička, která je v uvedené knihovně realizována pomocí LUT, jelikož GPU pracuje v aritmetice plovoucí řádové čárky.

Aplikace aproximačních násobiček je do programu implementována v podobě přepínače `--mult_file=approx_mult.bin`, který jako parametr očekává cestu k binárnímu souboru s implementací aproximační násobičky. Tento přepínač pak způsobí, že v průběhu evoluce je validace navržených sítí prováděna na modelech, které používají aproximační násobičku `approx_mult.bin`. Výsledkem evoluce je pak množina jedinců, kteří představují nejlepší nalezená řešení z pohledu přesnosti a počtu parametrů pro zadanou násobičku, která bude použita ve všech konvolučních vrstvách.

Kapitola 8

Experimenty a vyhodnocení

Tato kapitola obsahuje experimenty, které byly provedeny za účelem vyhodnocení implementovaného programu. Na úvod této kapitoly je uvedeno nastavení jednotlivých experimentů s popisem cílů, které si kladou. Následují výsledky a vyhodnocení provedených experimentů. Na závěr této kapitoly je uvedeno srovnání dosažených výsledků s jinými, stejně zaměřenými algoritmy.

8.1 Použitá datová sada

Experimenty byly prováděny na datové sadě CIFAR-10, která se skládá z 60 000 barevných obrázků o velikosti 32×32 pixelů, rozdělených do 10 tříd. Jedná se o jednu z nejpoužívanějších standardních datových sad, která se používá pro srovnání výkonnosti různých algoritmů, zaměřených na klasifikaci obrazových dat. Ukázka této datové sady je zobrazena na obrázku 8.1.

Obrázek 8.1: Ukázka datové sady CIFAR-10.

8.2 Nastavení experimentů

Experimenty byly spuštěny na školním GPU serveru výzkumné skupiny SC@FIT, disponujícím čtyřmi grafickými kartami NVIDIA GTX 1080 (Pascal), 8GB RAM, které byly použity pro akceleraci procesu učení CNN. Všechny ostatní výpočty byly prováděny na CPU (Intel Xeon CPU E5-2620 v3 @ 2.40GHz, six cores). Uvedené experimenty byly prováděny se shodným nastavením parametrů, které je uvedeno v tabulce 8.1.

V provedených experimentech se pomocí termínu *přesnost* jedince/řešení rozumí účelová funkce f_1, uvedená v kapitole 6.4, představující přesnost klasifikace modelu konvoluční neuronové sítě, kterou daný jedinec reprezentuje. Obdobně se označením *miniaturizace* jedince/řešení rozumí účelová funkce f_2, uvedená v kapitole 6.4, představující převrácenou hodnotu logaritmu z počtu parametrů modelu konvoluční neuronové sítě, kterou daný jedinec reprezentuje. Cílem je tedy obě tyto účelové funkce maximalizovat.

Nastavení experimentů	
Parametry evolučního algoritmu	
Počet generací	20
Velikost populace	8
Parametry evaluace jedince	
Velikost trénovací sady	30 000
Velikost testovací sady	10 000
Počet epoch	20
Parametry CGP	
Počet řádků	6
Počet sloupců	13
L-back	6
Parametry dotrénování	
Velikost trénovací sady	50 000
Velikost testovací sady	10 000
Počet epoch	200

Tabulka 8.1: Tabulka s parametry použitými při experimentech.

Parametry Inception modulů						
Název	R_1	R_2	R_3	C_1	C_2	C_3
Inception_1	16	16	32	16	64	64
Inception_2	32	16	48	32	96	96
Inception_3	64	32	64	48	128	128
Inception_4	64	32	96	64	192	192
Inception_5	64	48	128	96	256	256
Inception_6	128	48	196	128	320	320

Tabulka 8.2: Tabulka s parametry jednotlivých inception modulů. Názvy parametrů odpovídají těm na obrázku 6.9a

8.3 Kontrolní experiment

Pro ověření funkčnosti a stability implementovaného programu byl proveden kontrolní experiment, který spočíval v realizaci 10 běhů programu se shodnými parametry. Z důvodů výpočetní náročnosti takovéhoto experimentu byl program spuštěn s následujícími parametry. Počet epoch byl nastaven na 20, přičemž populace se skládala ze 4 jedinců. V průběhu evaluace byly jedinci trénováni na podmnožině trénovací sady o velikosti 20000 vzorků, zatímco pro testování bylo použito všech 10000 vzorků původní testovací sady CIFAR-10. Jelikož se jedná o kontrolní experiment, zaměřený na otestování základní funkcionality implementovaného evolučního algoritmu, tak finální dotrénování nejlepších nalezených jedinců nebylo nutné. Cílem tohoto experimentu tak bylo pouze zjistit, zda se nalezená řešení v průběhu generací zlepšují. Výpočetní mřížka tohoto experimentu se skládala ze všech implementovaných uzlů (základních, reziduálních a inception), uspořádaných do mřížky s 6 řádky a 13 sloupci. Hodnota parametru l-back byla nastavena na 6.

Výsledek tohoto experimentu je ukázán na obrázku 8.2, který zachycuje graf jedinců v první (prázdná kolečka) a poslední (plná kolečka) generaci každého běhu programu. Průměrná doba evoluce byla 5 hodin a 10 minut.

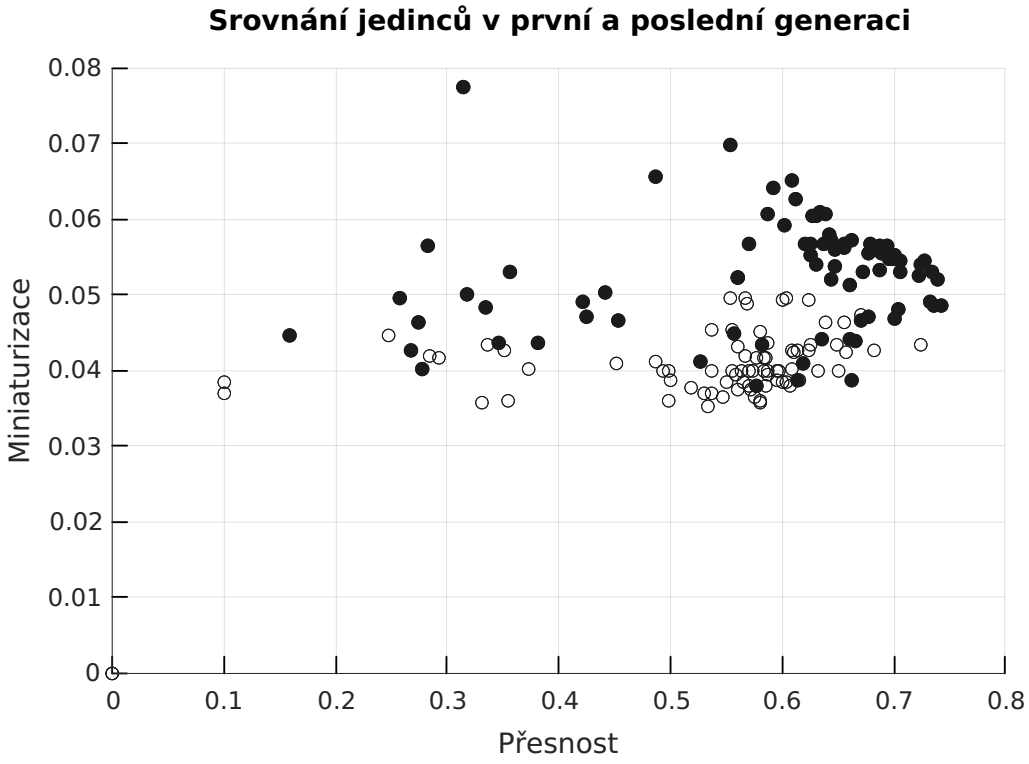
Z uvedených výsledků je vidět zlepšení (posun ve směru doprava a nahoru) jedinců ve 20. generacích vůči jedincům z 1. generací. V uvedeném grafu je rovněž možné si všimnout jedince s nulovou miniaturizací (ve skutečnosti s nekonečným počtem parametrů) a nulovou přesností, který vznikl tak, že navržená architektura přesáhla dostupné hardwarové prostředky (paměť), důsledkem čehož nebylo možné odpovídající model sestrojit. Takovýto jedinec pak samozřejmě představoval řešení s nulovou přesností.

8.4 Experimenty

Celkem bylo provedeno pět experimentů, jejichž cílem bylo demonstrovat výkonnost implementovaného programu. Cílem prvních čtyř experimentů bylo ukázat, že program je schopen nalézt řešení, využívající různé výpočetní uzly, které byly definovány v zadané výpočetní mřížce. Vzhledem ke značné výpočetní náročnosti, byl proveden vždy pouze jeden běh programu.

V prvním experimentu byla zadána výpočetní mřížka, obsahující pouze základní vrstvy CNN modelu, jako jsou konvoluční, seskupující, sčítací a konkatenační výpočetní uzly. Ve druhém experimentu byly do výpočetní mřížky přidány reziduální výpočetní uzly. V rámci třetího experimentu obsahovala výpočetní mřížka mimo základní vrstvy CNN modelu i inception uzly. Výpočetní mřížka čtvrtého experimentu byla složena ze všech implementovaných výpočetních uzlů, tedy ze základních, reziduálních a inception. Přesnost je uvedena po dotrénování nejlepších nalezených řešení.

Cílem posledního provedeného experimentu bylo ukázat, že program je schopen nalézt řešení pro zařízení, využívající aproximační násobičky. Pro tyto účely bylo provedeno několik běhů programu, které se lišily v použité aproximační násobičce. Ve všech bězích tohoto experimentu byla použita výpočetní mřížka se základními výpočetními uzly, jako u prvního experimentu. V průběhu evoluce pak byla pro účelovou funkci reprezentující přesnost nalezeného modelu použita přesnost s využitím aproximační násobičky. Použité aproximační násobičky jsou uvedeny v tabulce 8.3.



Obrázek 8.2: Graf výsledků kontrolního experimentu. Jednotlivé body reprezentují jedince, nalezené během evoluce. Prázdnými kolečky jsou znázorněni jedinci první generace a plnými kolečky jedinci 20. generace. Tento graf zachycuje jedince ze všech 10 běhů.

8.5 Výsledky experimentů

Výstupem každého experimentu je množina nejlepších nalezených jedinců, kteří jsou tvořeni jedinci Pareto fronty v poslední generaci evolučního algoritmu. Přesnosti (po dotrénování), počty parametrů a doby běhu (fáze dotrénování) nejlepších nalezených jedinců každého experimentu jsou zachyceny v tabulkách 8.5, 8.6, 8.7 a 8.8. Z uvedených výsledků plyne, že nejlepší řešení bylo nalezeno v experimentu, který využíval reziduální a inception výpočetní uzly. Z tohoto důvodu jsou v této sekci uvedeny výstupy pouze pro tento experiment. Průběh trénování nejlepších nalezených jedinců experimentu s reziduálními a inception výpočetními uzly je uveden na obrázku 8.4, přičemž ukázka Pareto front v první a poslední generaci tohoto experimentu je ilustrována na obrázku 8.3. Výstupy ostatních provedených experimentů jsou uvedeny v příloze A pro experiment s jednoduchými výpočetními uzly, B pro experiment s reziduálními výpočetními uzly a C pro experiment s inception výpočetními uzly.

Doba běhu evolučního algoritmu (označovaná jako *doba evoluce*) pro jednotlivé experimenty je uvedena v tabulce 8.4.

8.6 Výsledky experimentu s aproximačními násobičkami

Výsledky tohoto experimentu jsou uvedeny v tabulce 8.9, která obsahuje výčet nejpřesnějších nalezených řešení pro vybrané aproximační násobičky. Tato tabulka rovněž obsahuje

Aproximační násobičky					
Označení násobičky	Příkon (mW)	Plocha (μm^2)	EP (%)	MAE (%)	WCE (%)
mul8u_1JFF	0.391	709.6	0.00	0.00	0.00
mul8u_NGR	0.276	511.5	96.37	0.065	0.25
mul8u_1AGV	0.095	228.5	99.05	0.67	2.94
mul8u_JV3	0.034	110.8	99.16	2.15	8.21

Tabulka 8.3: Tabulka s parametry použitých aproximačních násobiček, kde *EP* označuje Error Probability, *MAE* značí Mean Absolute Error a *WCE* představuje Worst-Case Absolute Error. Uvedené hodnoty aproximačních násobiček byly převzaty z knihovny EvoApprox8b [47].

Doby evoluce provedených experimentů	
Experiment	Doba evoluce ($hh:mm:ss$)
Exp. 1 se základními uzly	02:20:53
Exp. 2 s reziduálními uzly	07:36:02
Exp. 3 s inception uzly	16:54:46
Exp. 4 s reziduálními a inception uzly	11:19:05

Tabulka 8.4: Tabulka zachycující doby evoluce jednotlivých experimentů. Doba evoluce zde vyjadřuje čas od začátku evolučního algoritmu až po jeho konec. Do této doby se nepočítá doba strávená ve finálním dotrénování nejlepších jedinců.

srovnání přesností nalezených řešení s použitím uvedených aproximačních násobiček a s použitím přesných násobiček.

Na obrázcích 8.6, 8.7, 8.8 a 8.9 jsou naznačena nalezená řešení, v prostoru definovaném účelovými funkcemi představujícími přesnost a počet parametrů, pro první a poslední generaci evolučního algoritmu. Každé nalezené řešení je vyznačeno dvěma body, kde prázdným kolečkem je naznačeno umístění nalezeného řešení s využitím aproximační násobičky a plným kolečkem umístění odpovídajícího řešení s využitím přesné násobičky.

8.7 Zhodnocení výsledků

Z prvních čtyř provedených experimentů se jako nejlepší ukázala varianta, která využívala všechny implementované výpočetní uzly. Z tohoto experimentu vzešlo řešení, které dosáhlo přesnosti 86.5 % s počtem parametrů 3604414. Jedno z nejlepších současných řešení, nazvané *DenseNet* [27], dosáhlo na datové sadě CIFAR-10 přesnosti 96.54 % s 25.6M parametry. Pro srovnání je zde uvedena i tabulka 8.10, která obsahuje přesnosti a počty parametrů konvolučních neuronových sítích, navržených různými evolučními algoritmy.

Z uvedených informací plyne, že řešení, nalezené pomocí implementovaného programu, zatím nedosahuje výsledků srovnatelných s těmi v tabulce 8.10, každopádně se jim poměrně blíží. Dále je nutné si uvědomit, že implementovaný program nebyl zaměřen pouze na hledání nejpřesnějšího řešení, ale na nalezení množiny řešení, která jsou výhodná z pohledu přesnosti a počtu parametrů, důsledkem čehož se v množině nejlepších řešení budou nacházet jedinci z různých konců Pareto fronty. Z toho plyne, že mezi nejlepšími jedinci bude

Výsledky experimentu s jednoduchými uzly			
Označení jedince	Přesnost (%)	Počet parametrů	Doba běhu (GPU hodiny)
Individual_117	71.4	298238	00:29:50
Individual_58	70.6	86538	00:31:00
Individual_166	68.5	68782	00:27:00
Individual_56	67.5	66690	00:57:00
Individual_165	65.6	62470	00:58:00
Individual_98	62.6	15166	00:28:17
Individual_150	61.7	6674	00:29:23
Individual_161	54.2	4610	00:27:48

Tabulka 8.5: Tabulka s výsledky nejlepších nalezených jedinců z experimentu, který využíval pouze jednoduché výpočetní uzly.

Výsledky experimentu s reziduálními uzly			
Označení jedince	Přesnost (%)	Počet parametrů	Doba běhu (GPU hodiny)
Individual_71	75.4	6144186	02:47:33
Individual_157	73.5	813658	01:42:05
Individual_149	72.1	518538	00:55:03
Individual_75	72.0	531298	00:48:51
Individual_134	71.4	330258	00:48:30
Individual_90	71.0	513498	00:45:56
Individual_159	69.2	306730	00:37:08
Individual_80	66.4	303570	00:39:16

Tabulka 8.6: Tabulka s výsledky nejlepších nalezených jedinců z experimentu, který využíval reziduální výpočetní uzly.

řešení s nejmenším počtem parametrů (a pravděpodobně s nejmenší přesností) a řešení s největší přesností (a pravděpodobně s největším počtem parametrů). Implementovaný program ale k těmto řešením přistupuje stejně (například z pohledu regularizace nebo velikosti učícího kroku), což má za následek, že trénování těchto jedinců není optimální. Toho je možné si všimnout na průběhu učení nalezených řešení, v grafech 8.4, kde validační křivka některých řešení klesá, zatímco jiných stoupá. Řešením tohoto problému by byla implementace adaptivního nastavení různých parametrů nalezených řešení tak, aby bylo trénování CNN modelů, které tato řešení představují, co nejvíce efektivní.

Poslední provedený experiment, zaměřený na hledání nejlepších řešení s využitím aproximačních násobiček, ukázal, že implementovaný program je schopen nalézt taková řešení, která dosahují nejlepších výsledků na zařízeních, která využívají netradiční výpočetní prostředky. Pro ověření funkcionality tohoto rozšíření implementovaného programu byl proveden experiment s využitím pouze základních výpočetních uzlů (tedy bez reziduálních a inception modulů), skládající se ze čtyř běhů programu, kde v každém běhu byla použita jiná aproximační násobička. Jedinci dosahující největší přesnosti z každého běhu programu jsou uvedeni v tabulce 8.9, ve které je pro srovnání zachycena i přesnost daného řešení s využitím přesné násobičky.

Výsledky experimentu s inception uzly			
Označení jedince	Přesnost (%)	Počet parametrů	Doba běhu (GPU hodiny)
Individual_165	79.6	10501598	08:20:02
Individual_31	79.3	10275838	07:03:20
Individual_158	77.0	920722	03:30:03
Individual_152	76.7	8376206	10:49:13
Individual_114	74.3	435434	02:05:30
Individual_71	73.4	919642	03:55:46
Individual_142	71.0	86250	00:23:13
Individual_11	68.4	134682	00:23:41

Tabulka 8.7: Tabulka s výsledky nejlepších nalezených jedinců z experimentu, který využíval inception výpočetní uzly.

Výsledky experimentu s reziduálními a inception uzly			
Označení jedince	Přesnost (%)	Počet parametrů	Doba běhu (GPU hodiny)
Individual_102	86.5	3604414	05:14:53
Individual_116	83.8	3559758	03:54:05
Individual_166	83.6	3260334	04:24:56
Individual_133	75.3	901794	03:03:49
Individual_164	73.9	236186	01:03:20
Individual_74	69.3	144642	00:25:30
Individual_153	68.9	41322	00:36:22
Individual_137	67.4	14730	00:20:25

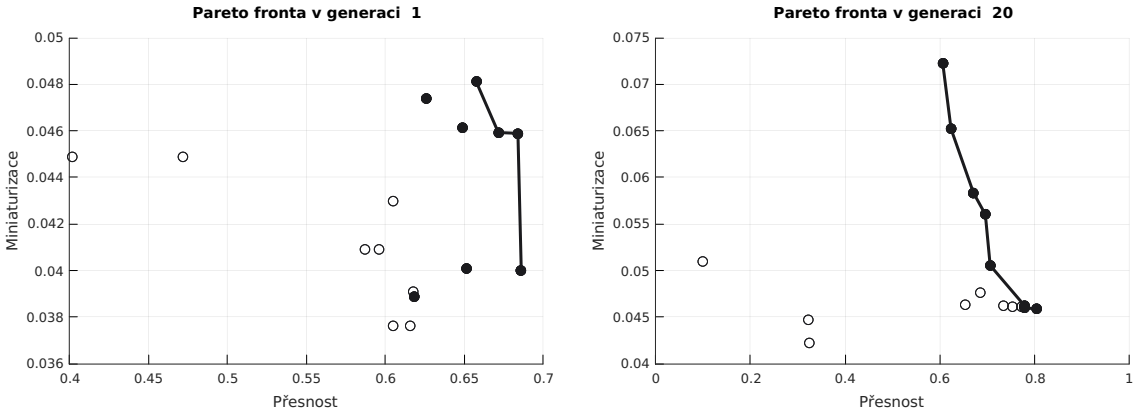
Tabulka 8.8: Tabulka s výsledky nejlepších nalezených jedinců z experimentu, který využíval reziduální a inception výpočetní uzly.

8.8 Pokračování práce

Z provedených experimentů je patrné, že implementovaný program je schopen s využitím uživatelem definovaných omezení (v podobě zadané CGP výpočetní mřížky, parametrů programu nebo použité násobičky) nalézt takové architektury CNN, které jsou výhodné z pohledu přesnosti či počtu parametrů. Tento postup tak uživateli nabízí možnost parametrizovat si proces hledání architektur CNN a zaměřit se tak na určité architektury, které jsou z jeho pohledu zajímavé. Implementovaný program tímto otevírá prostor jak pro výzkumnou činnost, tak pro praktické využití.

Z výzkumného pohledu představuje implementovaný program nástroj pro tvorbu a následnou analýzu navržených architektur, které by například jiným způsobem nevznikly. Program nemusí být nutně použit pro návrh kompletních architektur CNN, ale může být použit pro návrh určitých částí (modulů) CNN architektur. Takto navržené moduly by pak mohly být použity jako stavební bloky výsledné CNN architektury (jako jsou například reziduální nebo inception moduly).

Jak již bylo zmíněno, tak implementovaný program představuje zajímavé řešení i z hlediska praktického využití v reálném světě, jak dokazuje experiment s aproximačními náso-



(a) (b)

Obrázek 8.3: Ukázka Pareto front ve vybraných generacích experimentu, který využíval reziduální a inception výpočetní uzly.

Výsledky experimentu s aproximačními násobičkami			
Použitá násobička	Přesnost s násobičkou (%)	Opravdová přesnost (%)	Počet parametrů
mul8u_1JFF	72.5	73.8	2569258
mul8u_NGR	65.7	72.6	1033994
mul8u_1AGV	54.0	67.0	263914
mul8u_JV3	23.8	42.0	16874

Tabulka 8.9: Tabulka s výsledky nejlepších nalezených jedinců z experimentu, který využíval aproximační násobičky. V tabulce jsou uvedeny přesnosti (po dotrénování) nejpřesnějších nalezených řešení. Pro srovnání je zde uvedena i přesnost (po dotrénování) při použití přesné násobičky.

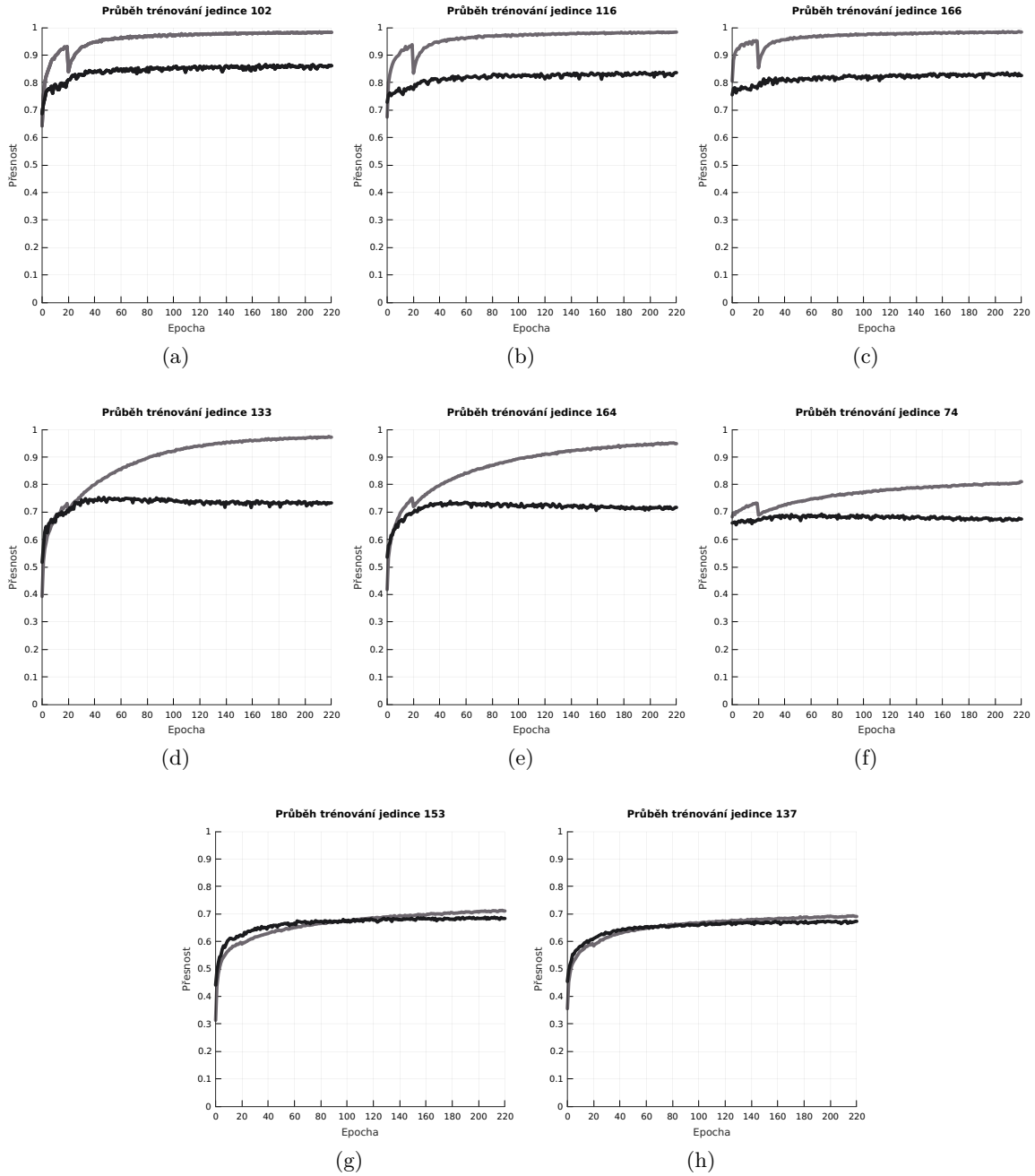
bičkami. Navržené a implementované řešení by tak mohlo sloužit jako nástroj pro automatizovaný návrh architektury CNN pro nějaké specifické (například vestavěné) zařízení, které je omezené dostupnými prostředky (výpočetními, paměťovými, prostorovými nebo jinými), a které je určené k plnění konkrétní úlohy.

Jelikož cílem této práce bylo navrhnout a ověřit řešení využívající CGP pro návrh architektur CNN, tak implementované řešení zatím představuje první krok na cestě k realizaci plnohodnotného nástroje, určeného k automatizovanému návrhu konvolučních neuronových sítí. Jako takové, toto řešení skýtá možnost rozšíření v mnoha ohledech. V budoucnu se tak lze zaměřit na rozšíření implementovaného programu o návrh nejen architektur CNN, ale obecně o návrh neuronových sítí, určených k jiným účelům, než je jen klasifikace obrázků. Dále by bylo možné zaměřit se na návrh různých, nejen dopředných, neuronových sítí. Pro celkové zlepšení nalezených řešení by bylo vhodné v budoucnu využít pokročilé techniky učení neuronových sítí, jako je využití regularizace nebo augmentace dat.

Dále by bylo vhodné provést detailnější analýzu vlivu nastavení parametrů CGP na kvalitu výsledku. Vzhledem k časové náročnosti těchto experimentů tato analýza nemohla být v této práci provedena.

Srovnání algoritmů pro CIFAR-10			
Název modelu	Přesnost (%)	Počet parametrů	Doba běhu (GPU dny)
DenseNet [27]	96.54	25.6M	–
NAS [73]	93.99	2.5M	22 400
CGP-CNN [59]	94.02	1.68M	27
CNN-GA [60]	95.22	2.9M	35

Tabulka 8.10: Srovnání parametrů CNN, které byly získány pomocí různých přístupů (jeden GPU den znamená, že algoritmus strávil jeden den na jednom GPU).

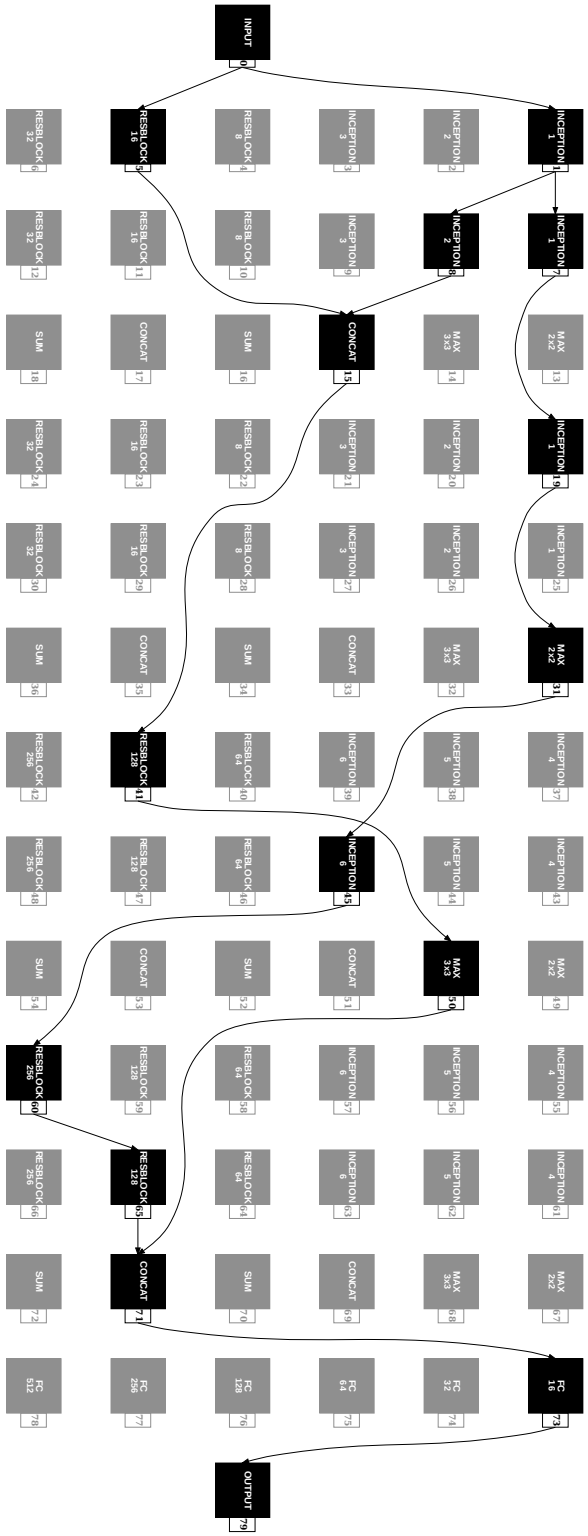


(a) (b) (c)

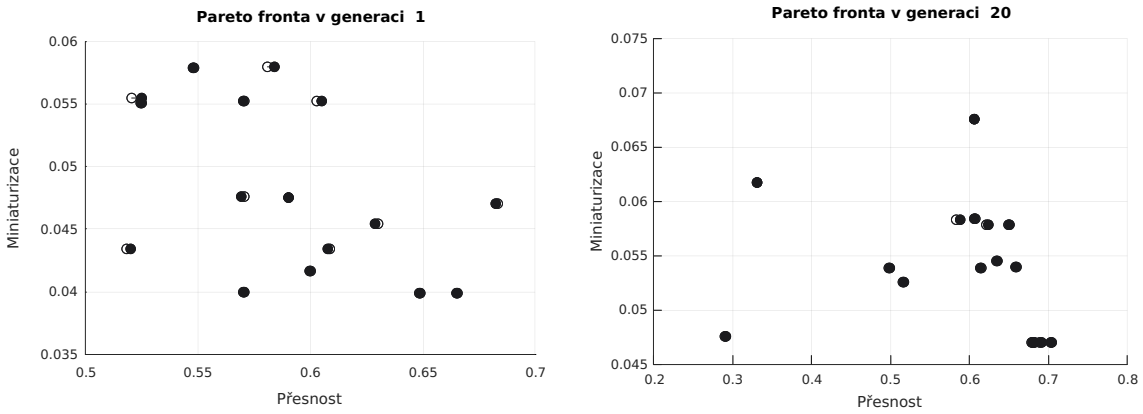
(d) (e) (f)

(g) (h)

Obrázek 8.4: Průběh učení nejlepších jedinců z experimentu, který využíval reziduální a inception výpočetní uzly.

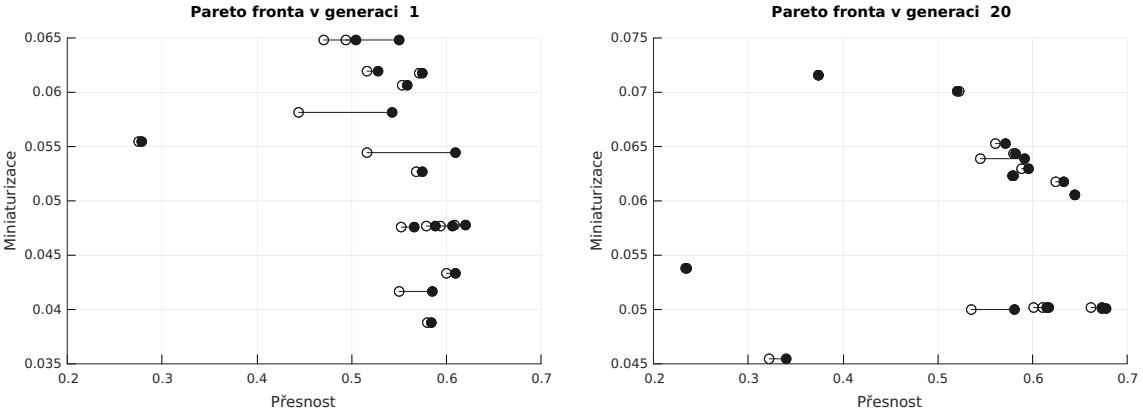


Obrázek 8.5: Architektura nejpřesnějšího jedince, označeného jako Individual_102, vzešlého z experimentu s reziduálními a inception výpočetními uzly.



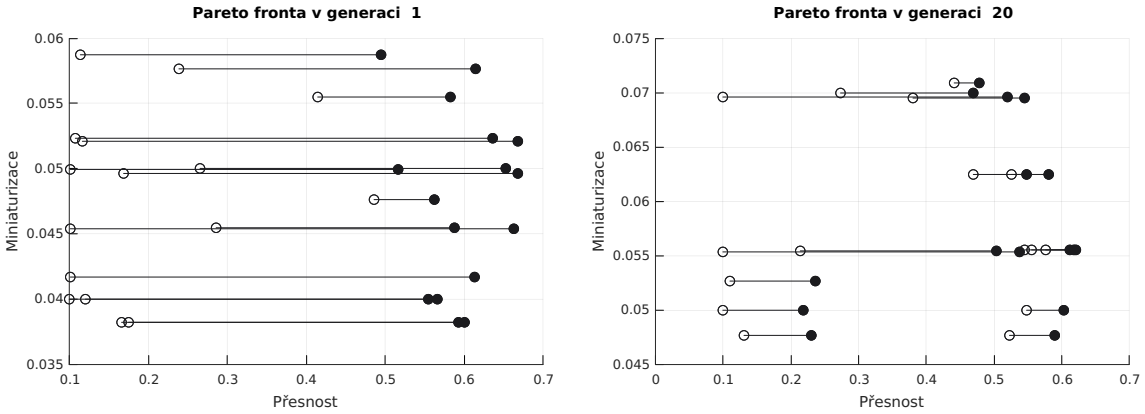
(a) (b)

Obrázek 8.6: Srovnaní nalezených řešení pro aproximační násobičku 1JFF v první a poslední generaci evolučního algoritmu. Prázdnými kolečky jsou naznačena umístění řešení s využitím aproximační násobičky a plnými kolečky umístění odpovídajících řešení s využitím přesné násobičky.



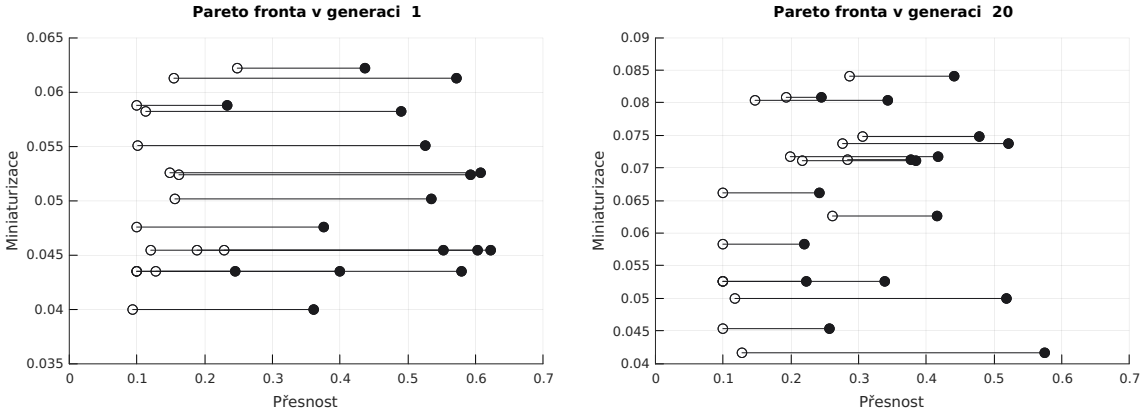
(a) (b)

Obrázek 8.7: Srovnaní nalezených řešení pro aproximační násobičku NGR v první a poslední generaci evolučního algoritmu. Prázdnými kolečky jsou naznačena umístění řešení s využitím aproximační násobičky a plnými kolečky umístění odpovídajících řešení s využitím přesné násobičky.



(a) (b)

Obrázek 8.8: Srovnaní nalezených řešení pro aproximační násobičku 1AGV v první a poslední generaci evolučního algoritmu. Prázdnými kolečky jsou naznačena umístění řešení s využitím aproximační násobičky a plnými kolečky umístění odpovídajících řešení s využitím přesné násobičky.



(a) (b)

Obrázek 8.9: Srovnaní nalezených řešení pro aproximační násobičku JV3 v první a poslední generaci evolučního algoritmu. Prázdnými kolečky jsou naznačena umístění řešení s využitím aproximační násobičky a plnými kolečky umístění odpovídajících řešení s využitím přesné násobičky.

Kapitola 9

Závěr

Cílem této práce bylo prozkoumat, navrhnout, implementovat a pomocí experimentů vyhodnotit úspěšnost využití evolučních výpočetních technik v návrhu konvolučních neuronových sítí. Pro účely neuroevoluce byla zvolena kombinace techniky kartézského genetického programování a genetického algoritmu NSGA-II.

V první kapitole byla popsána historie, základy a hlavní myšlenky evolučních výpočetních technik. Hlavní pozornost zde pak byla věnována kartézskému genetickému programování, které představuje základ této práce. Druhá kapitola se zabývala popisem umělých neuronových sítí, jejichž základy jsou postaveny na modelu umělého neuronu, nazvaného *perceptron*. Zbytek této kapitoly následně tvořil detailní rozbor algoritmu zpětného šíření chyby, který je často používán pro učení umělých neuronových sítí. Třetí kapitola byla zaměřena na popis konvolučních neuronových sítí, často používaných pro klasifikaci obrazových dat. Teoretickou část této práce završuje čtvrtá kapitola, která je dedikovaná popisu neuroevoluce, tedy spojení evolučních výpočetních technik a neuronových sítí. V této kapitole se nachází popis evolučního návrhu různých částí umělých neuronových sítí, přičemž hlavní důraz je kladen na návrh architektur konvolučních neuronových sítí. V kapitole číslo pět byl uveden návrh řešení, včetně popisu použitých prostředků, které zahrnovaly knihovnu TensorFlow pro práci s neuronovými sítěmi a multikriteriální optimalizační genetický algoritmus NSGA-II, na kterých bylo celé řešení postaveno. Šestá kapitola shrnovala implementaci a použití realizovaného řešení, včetně rozšiřujících prostředků pro vizualizaci výsledků. Završení této práce je shrnuto v poslední kapitole, která obsahovala provedené experimenty, vyhodnocení výsledků a nastínění případného pokračování práce.

Řešení, navržené a implementované v této práci, se podařilo úspěšně realizovat, čehož důkazem jsou výsledky dosažené v provedených experimentech. Implementovaná metoda automatizovaného návrhu architektur konvolučních neuronových sítí se ukázala být stabilní, jak dokazují experimenty provedené na datové sadě CIFAR-10. Nejlepší nalezené řešení, z pohledu přesnosti a počtu parametrů, dosáhlo přesnosti 86.5 % s počtem parametrů 3.6M.

V rámci této práce bylo rovněž implementováno rozšíření základní funkcionality navrženého programu o využití aproximačních násobiček. Ty byly do výsledného programu zakomponovány ve formě přepínače, který implementovanému programu umožňuje se zaměřit na návrh architektur konvolučních neuronových sítí, běžících na zařízeních, používajících aproximační násobičky.

Literatura

[1] Abadi, M.; Barham, P.; Chen, J.; aj.: TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, s. 265–283.
URL
https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Bremermann, H. J.; aj.: *Optimization through evolution and recombination.* In Self-organizing systems, ročník 93, Washington, DC: Spartan, 1962, s. 93–106.

[3] Caudill, M.: *Neural Nets Primer, Part I.* AI Expert, ročník 2, č. 12, Únor 1989: str. 46–52, ISSN 0888-3785.

[4] Clegg, J.; Walker, J. A.; Miller, J. F.: *A New Crossover Technique for Cartesian Genetic Programming.* In Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO '07, New York, NY, USA, 2007, s. 1580–1587.

[5] Corne, D. W.; Lones, M. A.: *Evolutionary Algorithms.* 2018.
URL http://arxiv.org/abs/1805.11014

[6] Cybenko, G.: *Approximation by superpositions of a sigmoidal function.* Mathematics of Control, Signals and Systems, ročník 2, č. 4, Prosinec 1989: s. 303–314.

[7] Darwin, C.: *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life.* John Murray, London, 1859.
URL https://www.biodiversitylibrary.org/item/135954

[8] Deb, K.; Agrawal, S.; Pratap, A.; aj.: A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, Springer Berlin Heidelberg, 2000, ISBN 978-3-540-45356-7, s. 849–858.

[9] Dorigo, M.; Stützle, T.: *Ant Colony Optimization.* Scituate, MA, USA: Bradford Company, 2004, ISBN 0262042193.

[10] Duman, E.; Uysal, M.; Alkaya, A.: *Migrating Birds Optimization: A New Metaheuristic Approach and Its Application to the Quadratic Assignment Problem.* Information Sciences, ročník 217, Duben 2011: s. 254–263.

[11] Ellson, J.; Gansner, E.; Koutsofios, L.; aj.: Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, Springer-Verlag, 2001, s. 483–484.

[12] Elsken, T.; Metzen, J. H.; Hutter, F.: *Neural Architecture Search: A Survey.* Journal of Machine Learning Research, ročník 20, č. 55, 2019: s. 1–21.

[13] FDominec: Nervové buňky. 2006, Wikimedia Commons.
URL https://commons.wikimedia.org/wiki/File:Neuron_(cesky)-1.svg

[14] Fogel, L. J.; Owens, A. J.; Walsh, M. J.: *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, 1966, 170 s.

[15] Friedberg, R. M.: *A Learning Machine: Part I*. IBM Journal of Research and Development, ročník 2, č. 1, Leden 1958: s. 2–13.

[16] Fukushima, K.: *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*. Biological Cybernetics, ročník 36, č. 4, Duben 1980: s. 193–202.

[17] Gershgorn, D.: *The inside story of how AI got good enough to dominate Silicon Valley*. Červen 2018, [Online; navštíveno 5.12.2019].
URL https://qz.com/1307091/the-inside-story-of-how-ai-got-good-enough-to-dominate-silicon-valley/

[18] Gibiansky, A.: *Convolutional Neural Networks*. 2014.
URL http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/

[19] Goldberg, D. E.: *Genetic Algorithms and Rule Learning in Dynamic System Control*. Proc. of the International Conference on Genetic Algorithms and Their Applications, 1985: s. 8–15.

[20] Goldberg, D. E.: *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., první vydání, 1989, ISBN 0201157675.

[21] Gwardys, G.: *Convolutional Neural Networks backpropagation: from intuition to derivation*. 2016.
URL https://grzegorzgwardys.wordpress.com/2016/04/22/8/

[22] Hardesty, L.: *Explained: Neural networks*. MIT News Office, Duben 2017.
URL
http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

[23] Hart, E.: *Evolutionary Computation*. 1993–present, ISSN 1063-6560.
URL https://www.mitpressjournals.org/loi/evco

[24] Hebb, D. O.: *The Organization of Behavior*. Wiley, 1949, ISBN 978-0805843002.

[25] Holland, J. H.: *Outline for a Logical Theory of Adaptive Systems*. Journal ACM, ročník 9, č. 3, Červenec 1962: s. 297–314, ISSN 0004-5411.

[26] Hornik, K.: *Approximation Capabilities of Multilayer Feedforward Networks*. Neural Networks, ročník 4, č. 2, Březen 1991: str. 251–257.

[27] Huang, G.; Liu, Z.; v. d. Maaten, L.; aj.: Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, ISSN 1063-6919, s. 2261–2269.

[28] Hubel, D. H.; Wiesel, T. N.: *Receptive fields and functional architecture of monkey striate cortex*. The Journal of Physiology, ročník 195, č. 1, 1968: s. 215–243.

[29] Jacobs, R. A.: *Increased rates of convergence through learning rate adaptation*. Neural Networks, ročník 1, 1987: s. 295–307.

[30] Jefkine: *Backpropagation In Convolutional Neural Networks*. 2016.
URL https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

[31] Khan, G. M.; Miller, J. F.; Halliday, D. M.: *Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture*. Evolutionary Computation, ročník 19, č. 3, 2011: s. 469–523.

[32] Khan, M. M.; Khan, G. M.: A Novel NeuroEvolutionary Algorithm: Cartesian Genetic Programming Evolved Artificial Neural Network (CGPANN). In *Proceedings of the 8th International Conference on Frontiers of Information Technology*, New York, NY, USA: Association for Computing Machinery, 2010, ISBN 9781450303422.

[33] Koza, J. R.: *Hierarchical Genetic Algorithms Operating on Populations of Computer Programs*. Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1, 1989: s. 768–774.

[34] Koza, J. R.: *Non-Linear Genetic Algorithms for Solving Problems*. Červen 1990, United States Patent 4935877, filed May 20, 1988.

[35] Koza, J. R.: *Human-competitive results produced by genetic programming*. Genetic Programming and Evolvable Machines, ročník 11, č. 3, Září 2010: s. 251–284.

[36] Kriesel, D.: *A Brief Introduction to Neural Networks* . 2007.
URL http://www.dkriesel.com

[37] Krizhevsky, A.; Sutskeve, I.; Hinton, G. E.: *ImageNet Classification with Deep Convolutional Neural Networks*. 2012.
URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[38] Kushida, J.; Hara, A.; Takahama, T.: *Cartesian Genetic Programming with Module Mutation for Symbolic Regression*. IEEE International Conference on Systems, Man, and Cybernetics (SMC), Oct 2018: s. 159–164, ISSN 1062-22X.

[39] Lindsay, G.: *Deep Convolutional Neural Networks as Models of the Visual System: Q&A*. 2018, [Online; navštíveno 10.1.2020].
URL https://neurdiness.wordpress.com/2018/05/17/deep-convolutional-neural-networks-as-models-of-the-visual-system-qa/

[40] Lu, Z.; Pu, H.; Wang, F.; aj.: *The Expressive Power of Neural Networks: A View from the Width*. In Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, s. 6231–6239.

[41] Marvin Minsky, S. P.: *Perceptrons - An Introduction to Computational Geometry*. MIT Press, 1969, ISBN 978-0-262-13043-1.

[42] Miller, J. F.: *An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach*. In Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2, GECCO'99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, ISBN 1-55860-611-4, s. 1135–1142.

[43] Miller, J. F.: *Cartesian Genetic Programming*. Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17310-3, s. 17–34.

[44] Miller, J. F.: *Introduction to Evolutionary Computation and Genetic Programming*. Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17310-3, s. 1–16.

[45] Miller, J. F.; Smith, S. L.: *Redundancy and Computational Efficiency in Cartesian Genetic Programming*. IEEE Transactions on Evolutionary Computation, ročník 10, č. 2, Září 2006: s. 167–174.

[46] Miller, J. F.; Thomson, P.; Fogarty, T.: *Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: a case study*. Dept. of Computer Studies, Napier University, Edinburgh, 1997.

[47] Mrazek, V.; Hrbacek, R.; Vasicek, Z.; aj.: EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, s. 258–261.

[48] Mrazek, V.; Vasicek, Z.; Sekanina, L.; aj.: *ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining*. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov 2019, ISSN 1933-7760, s. 1–8.

[49] Pan, X.; Xue, L.; Li, R.: *A new and efficient firefly algorithm for numerical optimization problems*. Neural Computing and Applications, ročník 31, č. 5, Květen 2019: s. 1445–1453, ISSN 1433-3058.

[50] Rahman, M. A.: *Neural Architecture Search (NAS)- The Future of Deep Learning*. Červen 2019, [Online; navštíveno 3.12.2019].
URL https://towardsdatascience.com/neural-architecture-search-nas-the-future-of-deep-learning-c99356351136

[51] Rechenberg, I.: *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart-Bad Cannstatt: Frommann-Holzboog, 1973.

[52] Riedmiller, M.; Braun, H.: *RPROP - A Fast Adaptive Learning Algorithm*. Proceedings of the International Symposium on Computer and Information Science VII, 1992.

[53] Saket Navlakha and Ziv Bar-Joseph: *Algorithms in Nature*. 2012–2015, [Online; navštíveno 6.12.2019].
URL http://www.algorithmsinnature.org/

[54] Schaffer, J. D.: *Multiple Objective Optimization with Vector Evaluated Genetic Algorithms*. Proc. of the International Conference on Genetic Algorithms and Their Applications, 1985: s. 93–100.

[55] Schwefel, H.-P.: *Evolutionsstrategie und numerische Optimierung*. Dissertation, Technische Universität Berlin, Germany, 1975, 370 s.

[56] Siarry, P.; Petrowski, A.; Hamida, S. B.: *Metaheuristics*, kapitola Evolutionary Algorithms. Springer International Publishing, 2016, ISBN 978-3-319-45403-0.

[57] Stanley, K. O.; Clune, J.; Lehman, J.; aj.: *Designing neural networks through neuroevolution*. Nature Machine Intelligence, ročník 1, č. 1, 2019: s. 24–35.
URL https://doi.org/10.1038/s42256-018-0006-z

[58] Stanley, K. O.; Miikkulainen, R.: Evolving neural networks through augmenting topologies. *Evolutionary Computation*, ročník 10, č. 2, 2002: s. 99–127.

[59] Suganuma, M.; Shirakawa, S.; Nagao, T.: A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, New York, NY, USA: Association for Computing Machinery, 2017, str. 497–504.

[60] Sun, Y.; Xue, B.; Zhang, M.; aj.: Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification. 2018.

[61] Szegedy, C.; Liu, W.; Jia, Y.; aj.: Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015: s. 1–9.

[62] Turner, A. J.; Miller, J. F.: *Cartesian Genetic Programming: Why No Bloat?* In European Conference on Genetic Programming, 2014, s. 222–233.

[63] Vassilev, V. K.; Miller, J. F.: *The Advantages of Landscape Neutrality in Digital Circuit Evolution*. In Evolvable Systems: From Biology to Hardware, Springer Berlin Heidelberg, 2000, s. 252–263.

[64] Vaverka, F.; Mrázek, V.; Vašíček, Z.; aj.: TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU. 2020.
URL https://www.fit.vut.cz/research/publication/12072

[65] Wilson, D. G.; Cussat-Blanc, S.; Luga, H.; aj.: *Evolving simple programs for playing Atari games*. 2018.
URL https://arxiv.org/abs/1806.05695

[66] Xin Yao: *Evolving artificial neural networks*. Proceedings of the IEEE, ročník 87, č. 9, Září 1999: s. 1423–1447, ISSN 1558-2256.

[67] Yao, L.; Xu, H.; Zhang, W.; aj.: *SM-NAS: Structural-to-Modular Neural Architecture Search for Object Detection*. 2019.
URL https://arxiv.org/abs/1911.09929

[68] Yu, T.; Miller, J. F.: *Neutrality and the Evolvability of Boolean Function Landscape*. In European Conference on Genetic Programming, Springer Berlin Heidelberg, 2001, s. 204–217.

[69] Yuce, B.; Packianather, M. S.; Mastrocinque, E.; aj.: *Honey Bees Inspired Optimization Method: The Bees Algorithm*. Insects, ročník 4, č. 4, 2013: s. 646–662.

[70] Zbořil, V. F.: *2. Acyklické a dopředné neuronové sítě. Algoritmus backpropagation.* Soft Computing, FIT VUT v Brně, prezentace [pdf].
URL https://www.fit.vutbr.cz/study/courses/SFC/private/index.html.cz

[71] Zdeněk Vašíček: *Cartesian GP in Optimization of Combinational Circuits with Hundreds of Inputs and Thousands of Gates*. In European Conference on Genetic Programming, LNCS 9025, Springer International Publishing, 2015, s. 139–150.

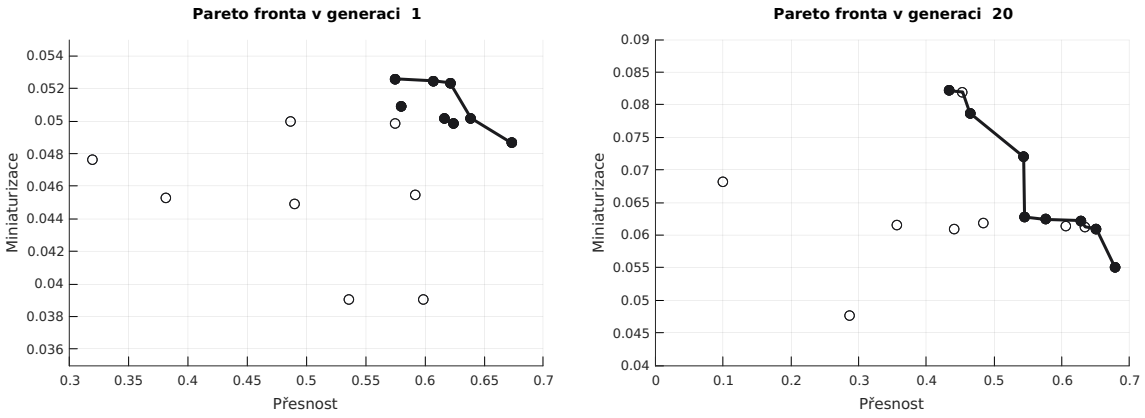
[72] Zhu, N.: *Neural Architecture Search for Deep Face Recognition*. 2019.
URL http://arxiv.org/abs/1904.09523

[73] Zoph, B.; Le, Q. V.: Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations*, 2017.

Příloha A

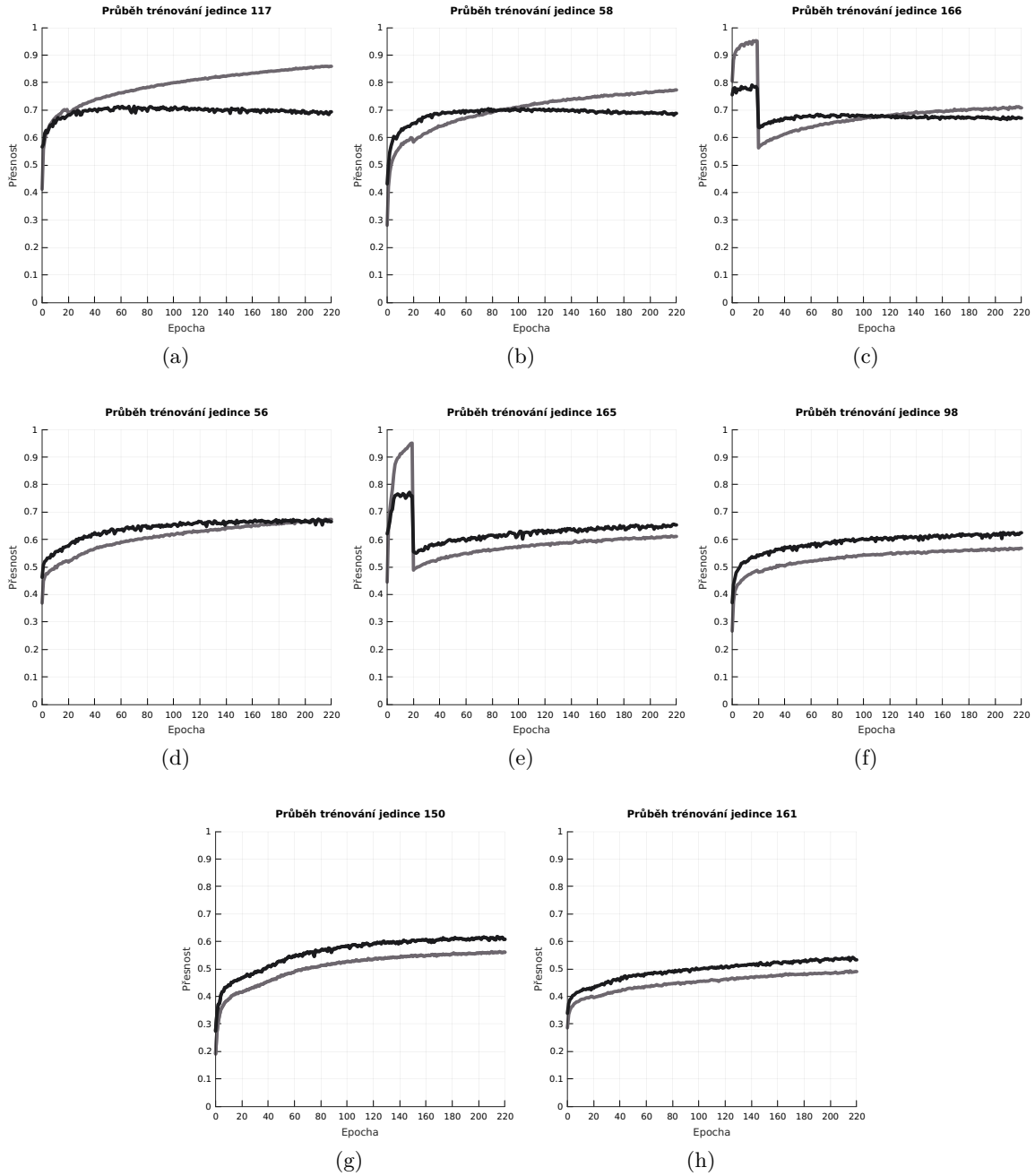
Vizualizace výsledků experimentu s jednoduchými výpočetními uzly

V této příloze se nachází vizualizace výstupů experimentu s jednoduchými výpočetními uzly z kapitoly 8. Ukázka Pareto fronty v první a poslední generaci běhu evolučního algoritmu v tomto experimentu je zachycena na obrázku A.1. Průběh trénování nejlepších nalezených jedinců tohoto experimentu jsou ukázány na obrázku A.2. Architektura nejlepšího nalezeného jedince je vyobrazena na ilustraci A.3.



(a) (b)

Obrázek A.1: Ukázka Pareto front ve vybraných generacích experimentu, který využíval pouze jednoduché výpočetní uzly.

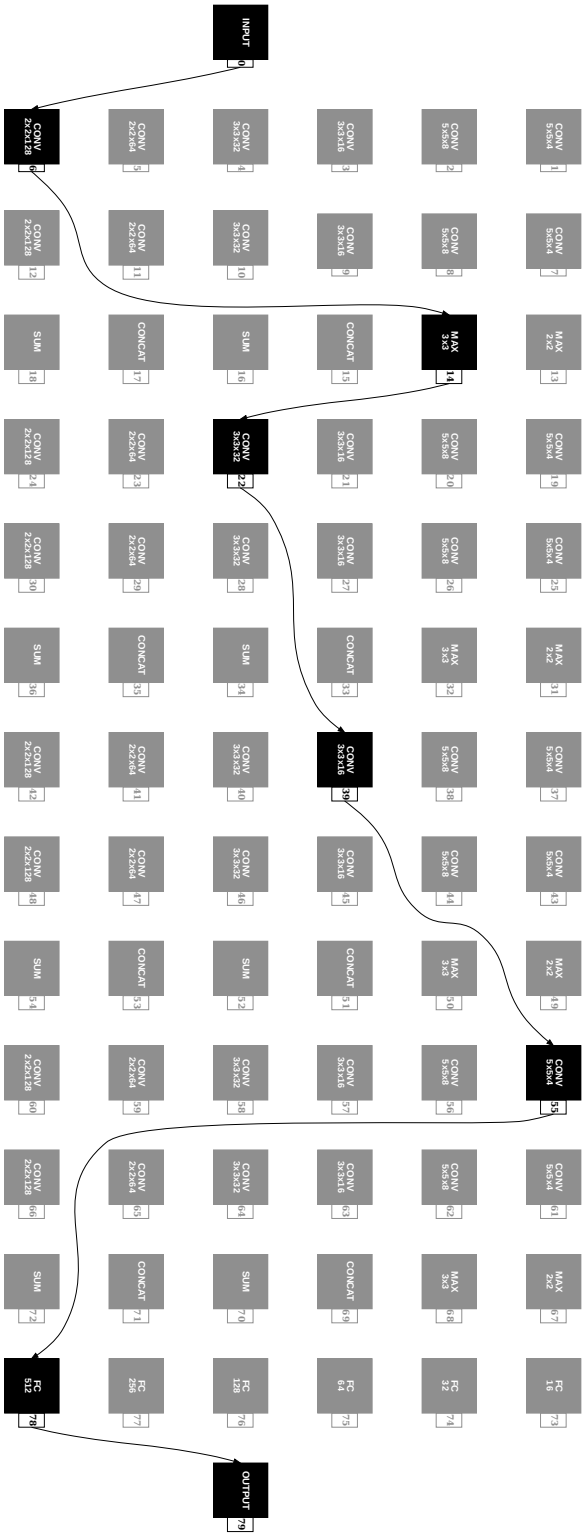


(a) (b) (c)

(d) (e) (f)

(g) (h)

Obrázek A.2: Průběh učení nejlepších jedinců z experimentu, který využíval pouze jednoduché výpočetní uzly.

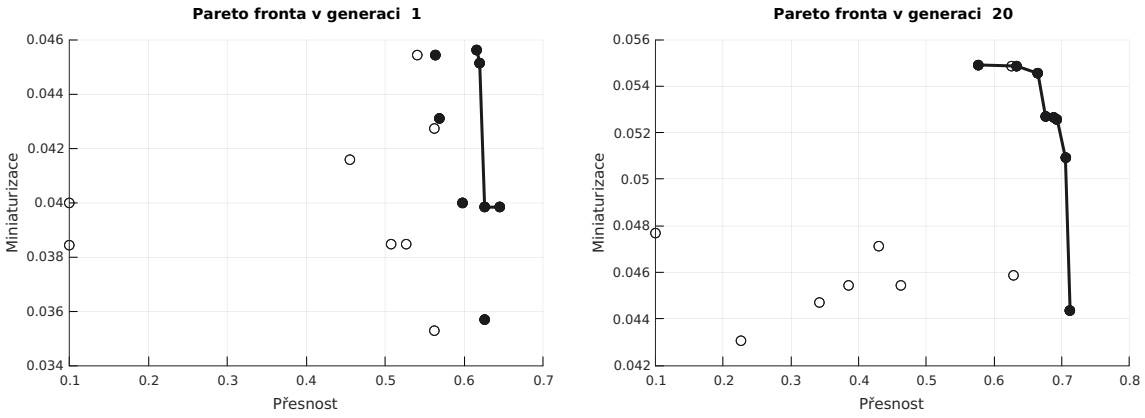


Obrázek A.3: Architektura nejpřesnějšího jedince, označeného jako Individual_117, vzešlého z experimentu s jednoduchými výpočetními uzly.

Příloha B

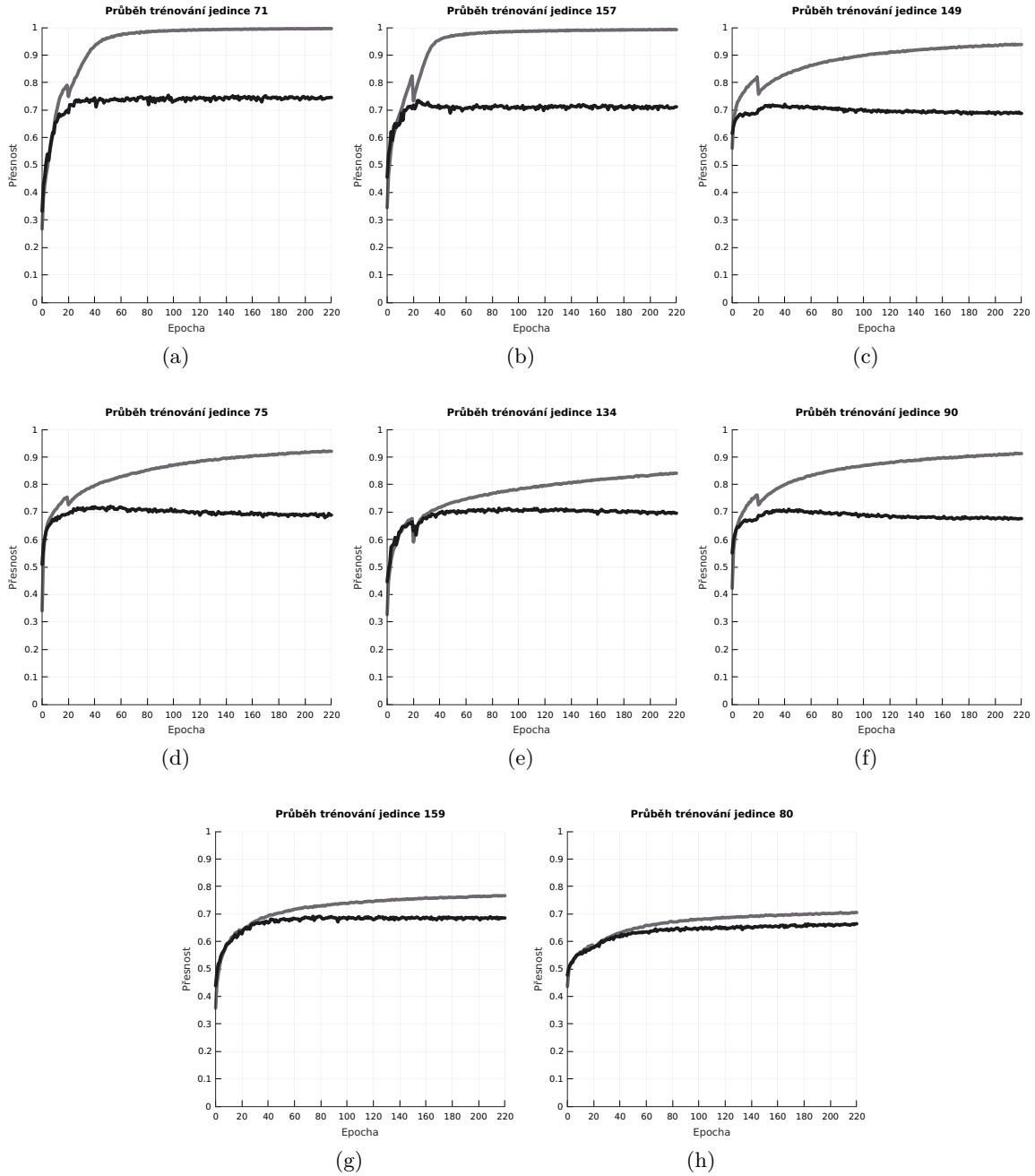
Vizualizace výsledků experimentu s reziduálními výpočetními uzly

V této příloze se nachází vizualizace výstupů experimentu s reziduálními výpočetními uzly z kapitoly 8. Ukázka Pareto fronty v první a poslední generaci běhu evolučního algoritmu v tomto experimentu je zachycena na obrázku B.1. Průběh trénování nejlepších nalezených jedinců tohoto experimentu jsou ukázány na obrázku B.2. Architektura nejlepšího nalezeného jedince je vyobrazena na ilustraci B.3.



(a) (b)

Obrázek B.1: Ukázka Pareto front ve vybraných generacích experimentu, který využíval reziduální výpočetní uzly.

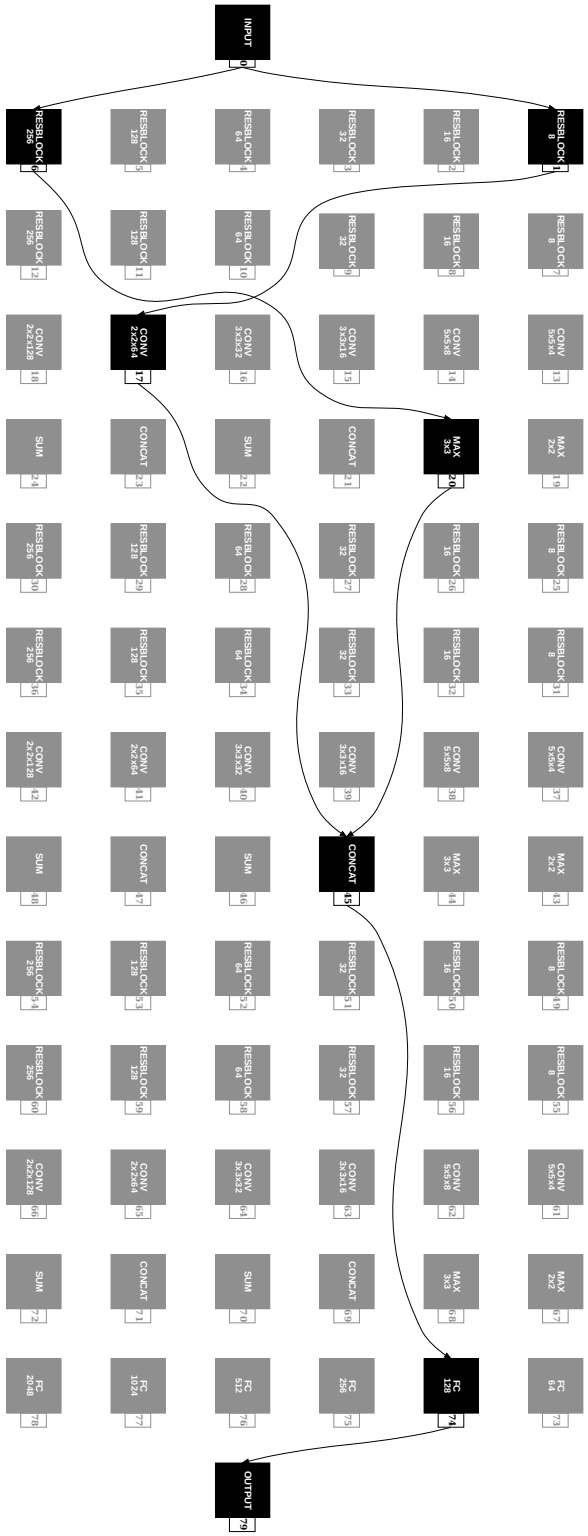


(a) (b) (c)

(d) (e) (f)

(g) (h)

Obrázek B.2: Průběh učení nejlepších jedinců z experimentu, který využíval reziduální výpočetní uzly.

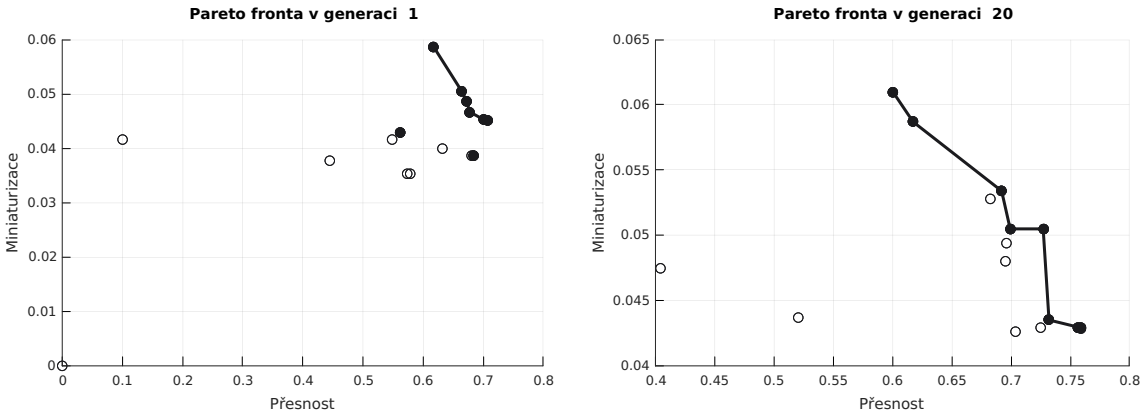


Obrázek B.3: Architektura nejpřesnějšího jedince, označeného jako Individual_71, vzešlého z experimentu s reziduálními výpočetními uzly.

Příloha C

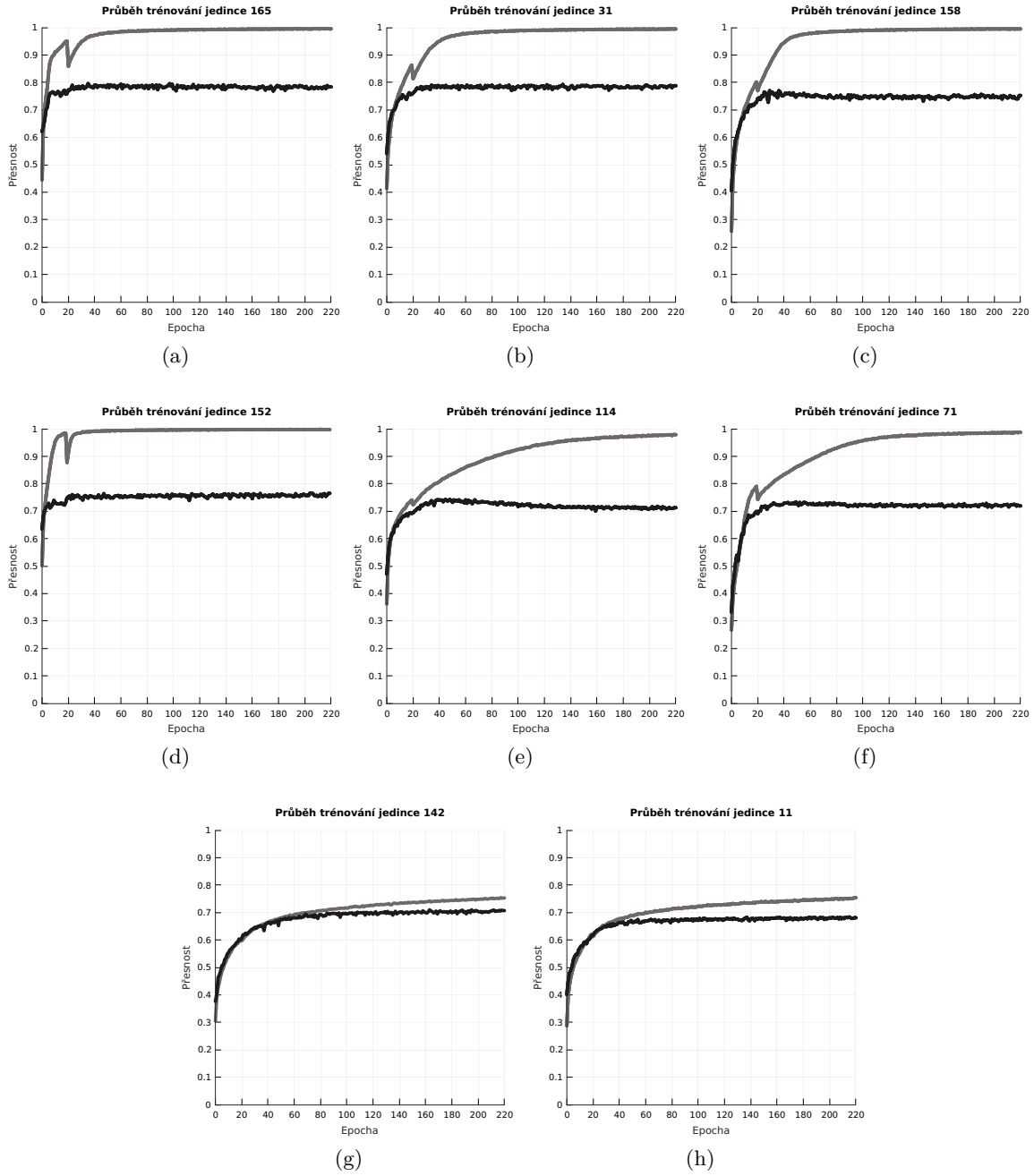
Vizualizace výsledků experimentu s inception výpočetními uzly

V této příloze se nachází vizualizace výstupů experimentu s inception výpočetními uzly z kapitoly 8. Ukázka Pareto fronty v první a poslední generaci běhu evolučního algoritmu v tomto experimentu je zachycena na obrázku C.1. Průběh trénování nejlepších nalezených jedinců tohoto experimentu jsou ukázány na obrázku C.2. Architektura nejlepšího nalezeného jedince je vyobrazena na ilustraci C.3.



(a) (b)

Obrázek C.1: Ukázka Pareto front ve vybraných generacích experimentu, který využíval inception výpočetní uzly.

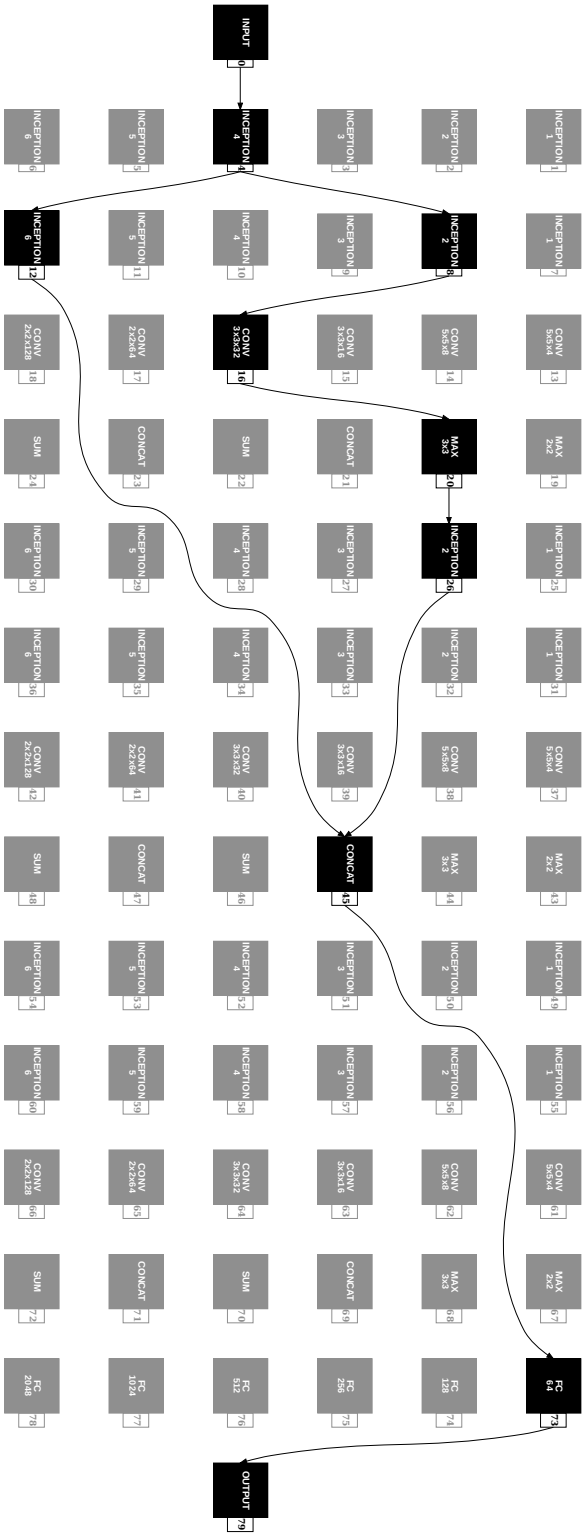


(a) (b) (c)

(d) (e) (f)

(g) (h)

Obrázek C.2: Průběh učení nejlepších jedinců z experimentu, který využíval inception výpočetní uzly.



Obrázek C.3: Architektura nejpřesnějšího jedince, označeného jako Individual_165, vzešlého z experimentu s inception výpočetními uzly.

Příloha D

Obsah přiloženého paměťového média

Na paměťovém médiu, přiloženém k této práci, se nacházejí všechny zdrojové kódy programu, implementovaném v této práci. Dále se zde nacházejí rozšiřující skripty a další pomocné programy. Obsah paměťového média je rozdělen následovně.

Ve složce `src/` se nacházejí zdrojové kódy implementovaného programu v jazyce Python, složka `scripts/` ukrývá pomocné skripty pro vizualizaci výsledků. V adresáři `text/` se nachází text této práce, společně se zdrojovými soubory v jazyce LaTeX. Soubor `README` obsahuje manuál pro práci s implementovaným programem.

Adresářová struktura paměťového média je následující:

```
/
├── src/
├── scripts/
├── text/
└── README
```