VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ EVOLUČNÍCH ALGORITMŮ PŘI UČENÍ NEURONOVÝCH SÍTÍ

EVOLUTIONARY ALGORITHMS FOR NEURAL NETWORKS LEARNING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID VOSOL**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. FRANTIŠEK ZBOŘIL, CSc.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Vosol David**
Program: Informační technologie
Název: **Využití evolučních algoritmů při učení neuronových sítí**
Evolutionary Algorithms for Neural Networks Learning
Kategorie: Umělá inteligence

Zadání:

1. Prostudujte zadanou literaturu.
2. Proveďte přehled možné spolupráce evolučních algoritmů a neuronových sítí.
3. Navrhněte demonstrační program pro některou ze zjištěných možností.
4. Navržený program implementujte.
5. Proveďte potřebné experimenty s cílem porovnání rychlosti učení a kvality odezvy pro klasickou neuronovou síť (například backpropagation) a neuronovou síť optimalizovanou evolučním algoritmem.
6. Zhodnoťte výsledky.

Literatura:

- García-Pedrajas, N., Ortiz-Boyer, D., Hervás-Martínez, C.: An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization, Elsevier, Neural Networks 19, 2006
- Chiroma, H., Mohd Noor, A.S., Abdulkareem, S., Abubakar, A., Hermawan, A., Qin, H., Hamza, M.F., Herawan, T.: Neural Networks Optimization through GeneticAlgorithm Searches: A Review, Applied Mathematics & Information Sciences, 2017
- Khan, M.M., Ahmad, A.M., Khan, G.M., Miller, J.F.: Fast learning neural networks using Cartesian genetic programming, Elsevier, Neurocomputing 121, 2013

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Zbořil František V., doc. Ing., CSc.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato práce má za úkol nalézt a porovnat možnosti spolupráce evolučních algoritmů při učení neuronové sítě a také jejich následné porovnání s klasickým přístupem učení pomocí backpropagation. Toto porovnání je demonstrováno na hluboké dopředné síti, která je využita při klasifikačních úlohách. Optimalizace probíhá na úrovni hledání optimálních hodnot vah a biasů sítě při zachování její stejné topologie. Jako evoluční algoritmy pro tuto optimalizaci jsou vybrány tři metody. Jedná se o genetický algoritmus, diferenciální evoluci a optimalizaci hejnem částic. Demonstrační program je implementován v programovacím jazyce Python3 a to bez použití knihoven pro strojové učení.

Abstract

Main point of this thesis is to find and compare posibilities of cooperation between evolutionary algorithms and neural network learning and their comparison with classical learning technique called backpropagation. This comparison is demonstrated with deep feed-forward neural network which is used for classification tasks. The process of optimalization is via search of optimal values of weights and biases within neural network with fixed topology. We chose three evolutionary approaches. Genetic algorithm, differential evolution and particle swarm optimization algorithm. These three approaches are also compared between each other. The demonstrating program is implemented in Python3 programming language without usage of any third parties libraries focused on deep learning.

Klíčová slova

neuroevoluce, evoluční algoritmy, genetický algoritmus, diferenciální evoluce, optimalizace hejnem částic, neuronová síť, hluboké učení, strojové učení, Python

Keywords

neuroevolution, evolutionary algorithms, genetic algorithm, differential evolution, particle swarm optimization, neural network, deep learning, machine learning, Python

Citace

VOSOL, David. *Využití evolučních algoritmů při učení neuronových sítí.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. František Zbořil, CSc.

Využití evolučních algoritmů při učení neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Františka Zbořila, CSc.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Vosol
13. května 2019

Poděkování

Tímto bych chtěl velmi poděkovat mému vedoucímu práce Doc. Ing. Františku Zbořilovi, CSc., za jeho ochotu, pomoc, cenné rady a investovaný čas, který mi poskytl při odborném vedení této bakalářské práce.

Obsah

Kapitola 1

Úvod

Umělá inteligence je v dnešní době jedna z nejrychleji se rozrůstajících oblastí informatiky. Často se užívá k řešení velice komplexních problémů jako rozpoznávání obrazu, mluvené řeči, predikci počasí, autonomnímu řízení vozidel nebo na zpracování velkých dat. Na tyto problémy obecně běžné exaktní algoritmy nestačí. S dnešním nárůstem výpočetního výkonu počítačů nabývá umělá inteligence mnohem většího významu a využití, než v 50. letech minulého století, kdy začala být poprvé zkoumána. Umělou inteligenci můžeme rozdělit na několik hlavních oblastí. Mezi nejdůležitější patří zejména strojové učení, expertní systémy, dolování dat, umělé neuronové sítě a optimalizační evoluční algoritmy.

V této práci se zaměřujeme na poslední dvě zmiňované oblasti. Každá slouží primárně k jinému účelu a fungují na absolutně rozdílných principech. Evoluční algoritmy primárně slouží k vyhledání optimálního řešení nějakého lineárně neseparovatelného problému a umělé neuronové sítě naopak ke klasifikačním problémům a rozpoznávání dat.

Naším cílem je navrhnout a implementovat nástroj, který tyto dvě oblasti spojuje. Konkrétněji se budeme snažit o spojení evolučních přístupů s hlubokou neuronovou sítí. Moderní literatura toto spojení v posledních letech začala nazývat *Neuroevolucí*. Jedná se o mocný nástroj, který spojuje výhody obou metod a je zde prostor pro jeho budoucí výzkum a využití.

V praxi budeme za pomocí genetického algoritmu, diferenciální evoluce a optimalizací hejnem částic evolučně optimalizovat hlubokou neuronovou síť, tak aby dosáhla co nejvyšší kvality výsledků na různých klasifikačních problémech. To bude provedeno takovým způsobem, že evoluční algoritmus bude hledat optimální hodnoty vah spojení mezi jednotlivými vrstvami neuronů uvnitř sítě. Jednotlivé metody budou mezi sebou porovnány a na základě jejich výsledků budou provedeny případné změny s cílem zvýšit kvalitu klasifikace a rychlost učení neuronové sítě.

Druhou částí této práce bude následné porovnání těchto navržených optimalizačních metod s klasickou metodou učení neuronové sítě, zvané *Back-propagation* neboli algoritmus zpětného šíření chyby, která je dosud nejvíce používanou metodou pro učení neuronových sítí. Budeme sledovat rychlost učení, celkovou odezvu sítě, ale také časovou náročnost zmíněných algoritmů. Vše bude demonstrováno při experimentech na několika vybraných klasifikačních problémech.

Kapitola 2

Evoluční algoritmy

Jednou z důležitých oblastí umělé inteligence jsou bezpochyby evoluční algoritmy. Jsou to algoritmy inspirované biologickými procesy v přírodě. Toto odvětví umělé inteligence začalo vznikat již v 50. letech dvacátého století, nicméně až ke konci století byly představeny moderní varianty těchto algoritmů. Mezi tzv. klasické evoluční algoritmy můžeme zařadit genetický algoritmus, diferenciální evoluci, evoluční strategie či genetické programování. Tyto metody vycházejí z biologických procesů v přírodě a byly poprvé souhrnně popsány biologem Charlesem Darwinem v 19. století. Často se označují jako Darwinova evoluční teorie nebo darwinismus [5].

Tato teorie vychází z předpokladu, že v přírodě přežijí jen nejsilnější jedinci v rámci daného druhu, a ti pak mají právo se křížit a dále šířit svůj genetický materiál. V evolučních algoritmech se tak pracuje s pojmy jako populace, generace, křížení a jiné. V rámci dané populace hledají nejsilnější jedince (jedince s nejlepší hodnotou řešení daného problému), kteří se následně stávají rodiči a jejich genetický materiál se zkříží s dalšími nejsilnějšími jedinci. To nám zaručuje jistou pravděpodobnost vzniku opět o něco kvalitnějšího jedince. Naopak slabí jedinci jsou z populace vyřazeni a dále se evolučně nevyvíjí. Tímto iterativním procesem se postupně prohledává stavový prostor řešení zadaného problému, který má tendenci konvergovat k optimálním řešením [22].

Evoluční algoritmy se používají v typu úloh, u kterých nelze nebo není znám exaktní algoritmus pro jejich optimální řešení. Jednou z jejich nejsilnějších stránek je fakt, že potřebují jen velmi málo informací o zadaném problému a jsou tak vhodné pro hledání řešení problémů v nejrůznějších oblastech. Prakticky jediné, co musí evoluční algoritmus znát je tzv. *fitness* funkce, na jejímž základě optimalizujeme řešení zadaného problému a která hodnotí kvalitu takového řešení. Evoluční algoritmy se staly velice silnou zbraní v oblastech, ve kterých známe optimální hodnotu řešení, ale nemáme k dispozici prakticky žádná vstupní data.

V této kapitole vycházíme z těchto zdrojů: [36], [31], [12], [8], [37].

2.1 Pojmy v evolučních algoritmech

Prakticky všechny evoluční algoritmy jsou založené na stejném principu. Na počátku je vygenerována počáteční populace a to buď náhodně nebo za užití určité heuristiky. Tato populace se skládá z jedinců, kteří představují jednotlivá řešení daného problému. Každý z těchto jedinců je následně ohodnocen *fitness funkcí*, která určí kvalitu jejich řešení. Na základě tohoto ohodnocení jsou z populace nebo můžeme říci nulté generace následně vybráni

jedinci, kteří se budou podílet na vzniku generace nové, obvykle o stejné fixní velikosti. S množinou takto vybraných jedinců provedeme křížení, čímž nám vzniknou jedinci noví. Záleží na zvoleném algoritmu, zda-li se do budoucí generace dostanou pouze noví jedinci vzniklí křížením nebo i ti z minulé generace. Tomuto pojmu se říká generační obměna. Na takto vybrané jedince je aplikován proces mutace, kdy simulujeme určité nepředvídatelné změny v jejich genech a tím zajišťujeme větší diverzitu populace. Bylo dokázáno, že mutace je klíčovou součástí evolučních algoritmů, bez které mohou algoritmy uváznout pouze v lokálních optimech řešení. Takto získanou populaci můžeme nazývat novou generací, kterou opět posíláme na ohodnocení funkcí fitness. Tento proces opakujeme až do splnění ukončujících podmínek. Může se jednat o hodnotu chyby, počet iterací aj. Cílem tohoto procesu je, aby s postupem času hodnoty funkce fitness aplikované na jedince konvergovaly k jejímu optimálnímu řešení, a tak byl nalezen jedinec s nejlepší hodnotou řešení, kterého posléze prohlásíme za řešení nalezené daným evolučním algoritmem.

Genotyp

Genotyp představuje jedince z populace, který je složen z jednotlivých genů. S touto reprezentací pracuje proces křížení a mutace. Jako příklad reprezentace genotypu může být přirozené číslo zakódované do binární podoby, kdy jednotlivý gen představuje hodnotu 0 nebo 1 a genotyp pak celé binární číslo. S touto binární podobou bude následně provedeno křížení i mutace v podobě změny jednotlivých bitů jedince.

Fenotyp

Fenotyp představuje taktéž jedince z populace, ale s touto reprezentací naopak pracuje fitness funkce. Jako příklad fenotypu můžeme uvést reálné číslo, které bude vstupem do funkce fitness, která hledá maximum nějaké matematické funkce.

Tedy můžeme říci, že genotypem označujeme jedince složeného z určitých částí a fenotypem jeho reálnou hodnotu, na základě které určujeme kvalitu takového jedince, tedy hodnotu jedincova řešení.

2.2 Principy evolučních algoritmů

V této části si shrneme jednotlivé části evolučních algoritmů jako je princip selekce rodičů, způsob křížení těchto vybraných rodičů a následný vznik potomků, způsob jakým je prováděna mutace takto nově vzniklých jedinců a v neposlední řadě typy kódování a obvyklé ukončující podmínky celého evolučního algoritmu.

Princip obecného fungování evolučního algoritmu, který mají všechny varianty společný můžeme vidět v následujícím pseudokódu 1:

1. Náhodně inicializuj populaci A(s=0);
while *ukončující podmínka není splněna* **do**
    1. Vypočítej fitness každého jedince z A(s);
    2. Vyber jedince s nejlepším fitness jako rodiče B(s) z A(s);
    3. Vytvoř potomky C(s) z B(s);
    4. A(s+1) = C(s);
end

Algorithm 1: Pseudokód Evolučních algoritmů

Funkce Fitness

Správný výběr fitness funkce je klíčovým prvkem celého evolučního algoritmu. Při jejím nesprávném výběru je prakticky nemožné nalézt optimální řešení daného problému. Jedinec z populace je vstupem takové funkce a na základě jejího výstupu je jedinec ohodnocen. Toto ohodnocení jedince představuje kvalitu řešení daného problému, jež se evoluční algoritmus snaží optimalizovat.

Ukončující podmínky

Pro ukončující podmínky evolučních algoritmů mohou být zvoleny různé přístupy. Jedním z nich je ukončení po předem zvoleném počtu generací, tedy iterací algoritmu. Dalším pak dosažení určité hodnoty fitness u nejlepších jedinců, případně hodnota diference nejlepších jedinců v rámci dvou nebo více po sobě jdoucích generacích. Se snižující se diferencí mezi generacemi nám algoritmus naznačuje, že již konverguje do nějakého optima (ať už globálního nebo lokálního) a již se hodnota fitness dramaticky měnit nebude. Z tohoto důvodu je vhodné algoritmus předčasně ukončit.

2.2.1 Selekce

Operátorem selekce vybíráme z celkové populace ty jedince, kteří se budou podílet na vzniku nové generace. Obecně se ukázalo, že prostý výběr pouze nejlepších jedinců nevede k lepším výsledkům a rychlejší konvergenci algoritmu ke správnému řešení. Je potřeba určitá diverzita jedinců uvnitř populace, a proto by neměly být upřednostněni při výběru pouze ti jedinci, kteří dosáhli nejvyššího fitness. Existuje mnoho strategií při procesu selekce rodičů, my si zde popíšeme tři nejrozšířenější.

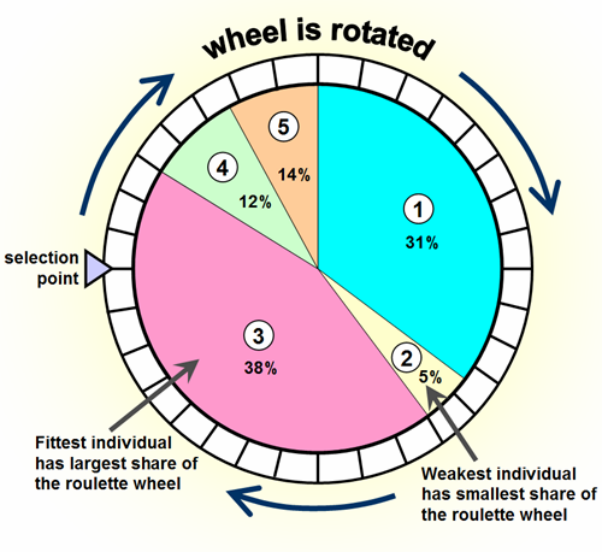
Ruleta

Selekce založená na principu ruletového kola, které si můžeme představit jako koláčový graf. Každý jedinec zaujímá takovou část grafu, která představuje poměrově pravděpodobnost jeho zvolení mezi ostatními jedinci. Tato úměrná část je dána jedincovou hodnotou fitness funkce. Tedy kvalitnější jedinci zaujímají větší výseč grafu a naopak. Matematicky lze tento vztah vyjádřit následovně:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Kde p_i je pravděpodobnost zvolení jedince i, f_i je jedincovo fitness a N je velikost populace.

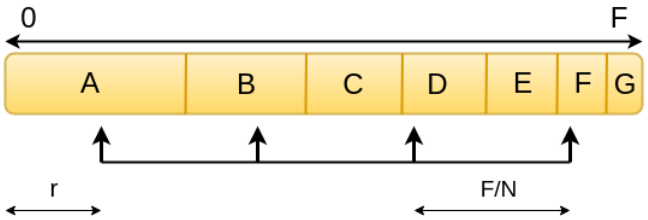
Jedinec je zvolen na základě roztočení kola, kdy ručička na okraji kola po svém zastavení se určí jedince, který je vybrán. Tato strategie ovšem není ideální při velkých rozdílech v kvalitě jedinců. Pak jsou přednostně vybíráni pouze ti nejúspěšnější a rychle tak zaberou místo jedincům s menším fitness. Takto může algoritmus poměrně rychle a v brzké fázi konvergovat k lokálnímu optimu. Naopak při vyrovnaném fitness je selekční tlak na zvolení nejlepších jedinců nízký, a tak je větší prostor pro diverzitu v nové generaci.



Obrázek 2.1: Ruletové kolo - 1 selekční bod, kolo se otáčí ve směru šipek a vidíme poměrové zastoupení jedinců v závislosti na jejich hodnotě fitness. Nejlepší jedinec (růžový) zabírá největší část výseče kola. [Převzato z: [34]]

Stochastické univerzální vzorkování

Podobným přístupem je také metoda SUS (Stochastic universal sampling), kde si opět můžeme představit ruletové kolo, nicméně tentokrát je náhodně zvolena hodnota r z rozsahu $(0 - F/N)$, kde F označuje celkovou sumu fitness hodnot všech jedinců a N celkový počet jedinců, které chceme vybrat. Tato náhodná hodnota poté představuje vzdálenost od které začínáme řadit ukazatele. Takových ukazatelů nebo chceme-li ručiček je na rozdíl od klasické rulety více a jsou od sebe poté vzdáleny přesně F/N. Tím zajistíme nižší selekční tlak, a tak i jedinci s menším fitness mohou být spíše vybráni do nové generace. Více nám přiblíží princip výběru následující obrázek:



Obrázek 2.2: Metoda SUS, výběr 4 jedinců do nové generace: $N = 4$, $r\epsilon(0, F/N)$, F je suma fitness všech jedinců

Turnaj

Tato strategie nevyžaduje, abychom znali hodnotu fitness celé populace, ale jen určitého jejího množství. Tato vlastnost je výhodná zejména při velkém počtu jedinců v populaci. Pracuje na principu, kdy se náhodně vybere N jedinců z populace, těm je následně spočtena hodnota fitness a na jejím základě je vybrán nejlepší z nich. Toto je opakováno, dokud

nedosáhneme požadovaného počtu rodičů. Selekční tlak můžeme jednoduše upravit změnou velikosti turnaje. Tedy jedinec s nižší hodnotou fitness bude pravděpodobněji vybrán při menší velikosti turnaje.

Elitismus

Nejtriviálnějším způsobem je strategie založená na pořadí jedinců podle jejich hodnoty fitness. Tato zdánlivě nevýhodná metoda, kdy jsou logicky upřednostněni pouze ti nejlepší a může tak docházet ke ztrátě diverzifikace, a tak předčasnému konvergování k pouhému lokálnímu optimu se dá snadno upravit. Pro zvýšení úspěšnosti této metody je možné například vybrat pouze určitou část nejlepších, čímž i urychlujeme činnost evolučních algoritmů a zároveň zachováváme dobrý genetický materiál. Zbylá část je pak vybrána například jedním z výše zmíněných způsobů.

2.2.2 Křížení

Rodiče vybraní procesem selekce jsou nyní obvykle po dvojicích spolu zkříženi (*crossover*). Vznikají tak dva noví potomci, se zděděnými geny po obou rodičích, pro budoucí generaci. Operátor křížení představuje hlavní složku evolučních algoritmů a princip, kterým vznikají noví, povětšinou kvalitnější jedinci. Typy křížení můžeme rozdělit podle reprezentace chromozomu a to na typy pro binární reprezentaci a na reprezentaci pomocí reálných hodnot.

2.2.3 Binární reprezentace

Binární reprezentace chromozomu se více hodí pro proces křížení u jedinců představujících přirozené celé číslo nebo například, kdy jsou geny nějakým výčtem možných hodnot. Je totiž velmi snadné takové číslo nebo podobu jedince převést na jeho binární podobu a následně zkřížit dva takové jedince. Jednoduše, podle typu křížení, prohodíme dané úseky binárních hodnot obou jedinců. Takovéto křížení je výpočetně jednoduché, a tak je i z tohoto důvodu preferováno. Nejužívanější typy křížení v binární reprezentaci jsou následující.

Jednobodové křížení

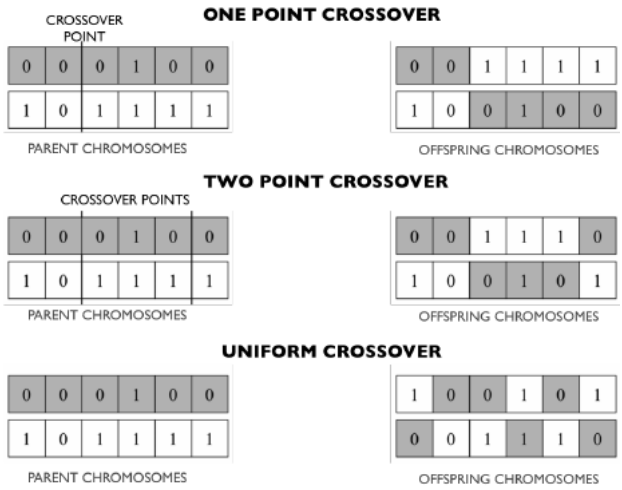
Jednobodové křížení je nejprimitivnějším typem křížení, kdy jsou náhodně 2 chromozomy určené ke vzájemnému křížení rozděleny ve stejném bodě na dvě části. Tyto části genů jsou poté mezi sebou vyměněny, takže 2 nově vzniklí jedinci obsahují část z každého z rodičů.

Dvoubodové křížení

Dvoubodové křížení je velice podobné tomu jednobodovému, s tím rozdílem že jsou zvoleny 2 body v rámci obou chromozomů a dvě z těchto tří částí jsou mezi rodiči prohozeny. Takto opět vzniknou dva noví jedinci, kteří obsahují úseky genů z obou svých rodičů.

Uniformní křížení

Uniformní křížení funguje na principu, že pro každý gen v chromozomu je vygenerována náhodná hodnota v rozsahu $(0, 1)$ a pokud je pravděpodobnost $i < 0.5$ zůstává tento gen v chromozomu 1. potomka a pokud je $i > 0.5$, pak tento gen obdrží 2. potomek. Takto je náhodná hodnota vygenerována pro všechny geny v chromozomu a ty jsou tak postupně prohozeny nebo zachovány.



Obrázek 2.3: Typy genetického křížení při binární reprezentaci [Převzato z: [25]]

2.2.4 Reprezentace pomocí reálných hodnot

Jelikož nelze spolehlivě a jednoduše zakódovat reálná hodnota do binární reprezentace bez ztráty přesnosti nebo za použití náročných operací případně při reprezentaci jedince více takovými hodnotami, je vhodné zvolit jiný přístup. Nedochází zde k doslovnému křížení jedinců z pohledu pozic genů, ale křížení na základě určitého poměru hodnoty každého z rodičů. Nejužívanějšími typy procesu křížení při reprezentaci chromozomu pomocí reálných hodnot jsou aritmetické a heuristické křížení.

Aritmetické křížení

Probíhá způsobem, kdy se každý jednotlivý gen z chromozomu pro nového jedince utváří na základě tohoto vzorce:

$$g_i^{O1} = g_i^{P1} * a_i + g_i^{P2} * (1 - a_i)$$
$$g_i^{O2} = g_i^{P2} * (1 - a_i) + g_i^{P2} * a_i$$

Kde g_i^{O1} je i-tý gen prvního potomka a g_i^{P1} je i-tý gen prvního rodiče. Důležitý je koeficient a_i, který určuje podíl obou hodnot daného genu z obou rodičů. Obvykle se nachází v rozsahu $a_i \in \langle 0.25, 1.25 \rangle$.

Heuristické křížení

U heuristického křížení je princip velice podobný tomu aritmetickému. Zde nám ale záleží na hodnotě fitness jedince a pouze jeden ze dvou nově vzniklých jedinců je zkřížen. Opět využíváme koeficientu, v tomto případě r, který se zapojuje do křížení a určuje vliv horšího jedince na nově vzniklého. Vzorec pro toto křížení je následující:

$$O_1 = P_{better} + r * (P_{worse} - P_{better})$$
$$O_2 = P_{better}$$

Obvykle je koeficient v rozsahu $r \in \langle 0, 1 \rangle$ a zůstává po celý proces křížení dvou jedinců neměnný. Tj. pro každý gen v chromozomu nabývá stejné hodnoty [37].

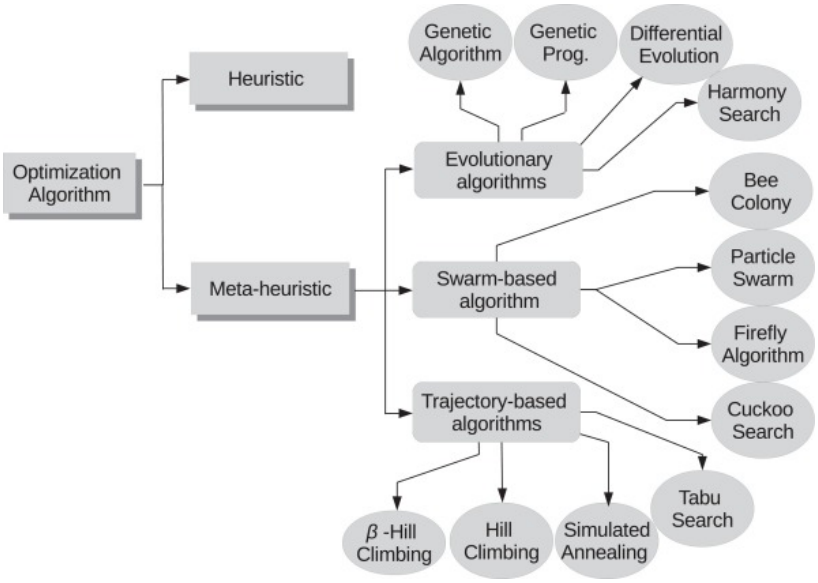
2.2.5 Mutace

Operátorem mutace se rozumí, že se s určitou malou pravděpodobností *mutation rate* změní hodnota genu v chromozomu. Aplikuje se na každého jedince zvlášť. U binární reprezentace procházíme celý chromozom a na základě hodnoty *mutation rate* negujeme hodnotu genu na dané pozici. U reprezentace chromozomu reálnými hodnotami přičteme náhodnému jedinci v populaci náhodnou hodnotu, obvykle se jedná maximálně o 10% z hodnoty chromozomu a tato hodnota může být i záporná.

Operátor mutace je velice důležitou součástí evolučních algoritmů, kdy díky ní získáváme pro jedince unikátní vlastnosti, které se jinak v populaci nemusejí nenacházet. Případně slouží také ke zvětšení diverzity jedinců v rámci populace. Mimo jiné může pomoci vyváznutí algoritmu z případného lokálního optima.

2.3 Varianty evolučních algoritmů

Do třídy evolučních algoritmů můžeme zařadit takové, které svým principem vycházejí z biologické populační evoluce, principem jinak známým jako darwinismus [5]. Mezi nejpoužívanější a zároveň nejznámější evoluční algoritmy můžeme zařadit Genetické algoritmy, Evoluční strategie, Genetické programování a Diferenciální evoluci. Všechny pracují na podobném principu, který zahrnuje konečnou velikost populace, iterativní proces tvorby populací - generace, křížení nebo replikace jedinců - rodičů a tvorbu nových potomků z těchto rodičů a jejich následná mutace. Trochu jiným přístupem jsou například metaheuristické algoritmy PSO (Particle Swarm Optimization) a ACO (Ant Colony Optimization Algorithms), které jsou inspirovány pohybem a vzájemnou interakcí organismů v rámci větší skupiny. Podle některých názorů však také spadají do kategorie evolučních algoritmů [16].



Obrázek 2.4: Rozdělení optimalizačních algoritmů [Převzato z: [7]]

2.3.1 Genetický algoritmus

Jedním z nejstarších a zároveň nejpoužívanějším je genetický algoritmus. V tomto algoritmu je jedinec reprezentován nejčastěji binárně. Pro výběr rodičů je užíván princip rulety případně turnaje, založené na základě hodnoty fitness a do další generace postupují pouze nově vzniklí potomci na základě tzv. generační obměny. Křížení bývá nejčastěji jednobodové, případně vícebodové a jsou k němu zapotřebí dva rodiče. Mutace je řešena uniformně a to případnou negací mutujícího bitu. Ukončující podmínky bývají obvykle dosažení určité hodnoty řešení [25], [20]. Následující pseudokód zobrazuje základní činnost genetického algoritmu 2:

while *není splněno ukončující kritérium* **do**
  1. Dekóduj všechny jedince n v populaci na odpovídající kandidátní řešení a vypočítej jejich fitness;
  2. Vyber n jedinců z populace s pravděpodobností úměrnou jejich fitness a umísti je do společného prostoru rodičů;
  while *není vygenerováno n potomků* **do**
    1. Vyber náhodně 2 rodiče ze společného prostoru rodičů a s pravděpodobností p_{cross} proveď křížení těchto rodičů, čímž vzniknou 2 potomci;
    2. Pokud křížení neproběhlo, tak jsou potomci pouhé kopie jejich rodičů;
    3. S pravděpodobnostní p_{mut} zmutuj každý prvek (gen) potomků (chromozomů);
  end
  Starou populaci nahraď potomky, čímž vzniká nová generace;
end

Algorithm 2: Pseudokód Genetického algoritmu [36]

2.3.2 Evoluční strategie

Evoluční strategie byly vyvinuty v šedesátých letech v Německu. Původně byly vytvořeny jako prohledávací heuristiky pro optimalizační problémy v oblasti strojírenství. Od genetických algoritmů se liší především tím, že fenotyp není nijak kódován, čili křížení a mutace probíhá změnou reálných hodnot atributů jedinců a ne jejich zakódované reprezentace. Parametry specifikující vlastnosti mutace jsou nazývány *parametry strategie* a atributy řešení jako *parametry rozhodování*. Evoluční strategie optimalizují oba zmíněné.

Jedinec je představován vektorem reálných čísel. Selekce rodičů je provedena uniformně. Operátor křížení se vyskytuje až u pozdějších variant tohoto algoritmu a bývá uskutečněn uniformně, avšak většinou s více rodiči. Pro mutaci byla využita tzv. *Gaussovská mutace*, kdy k potomkovi byla přičtena hodnota z Gaussova pravděpodobnostního rozdělení.

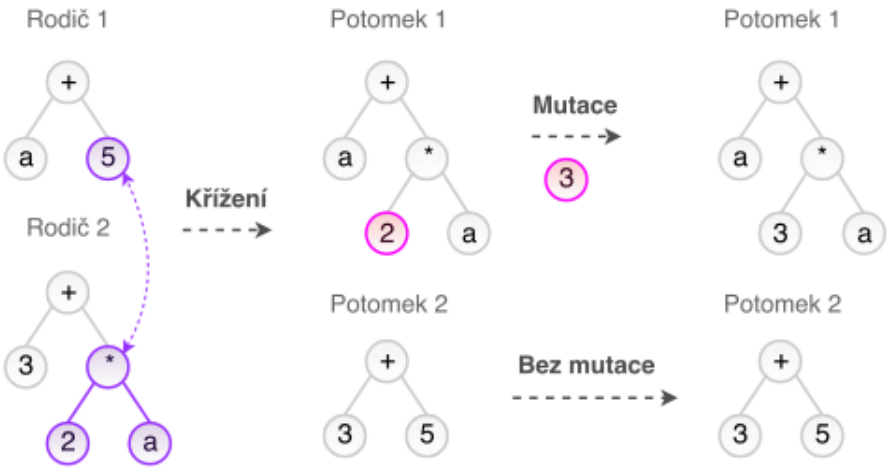
U evolučních strategií můžeme také poprvé vidět tzv. *self-adaptation*, kdy si algoritmus v průběhu může měnit své parametry podle potřeby. Pro výběr jedinců do nové populace je užit elitismus - určitý počet nejlepších jde do další generace. Existují dvě varianty tohoto výběru: $(\mu + \lambda)$ - kdy μ značí velikost populace a λ počet vygenerovaných potomků a výběr probíhá ze sjednocení těchto dvou množin. Druhou variantou je (μ, λ), kdy je výběr proveden pouze z λ potomků, kterých je zpravidla mnohonásobně více než je hodnota μ, tedy velikost populace [9].

2.3.3 Genetické programování

Tento přístup vznikl v 80.letech a jako první byl aplikován v programovacím jazyce LISP pro návrh programů. LISP je vysokoúrovňový jazyk, jehož základním stavebním kamenem je seznam, z toho název (*LISt Processor*). Genetické programování vychází z předpokladu, že jedinec je představován programem, který je zapsán jako stromová struktura. Tento program je složen z různých bloků, tj. podstromů. Ohodnocení jedince funkcí fitness pak záleží na výstupu daného programu. Čili pro zjištění hodnoty fitness je program spuštěn a jeho výstupní hodnota značí fitness tohoto jedince.

Program je tedy reprezentován stromovou strukturou, kdy jeho vnitřní nelistové uzly s minimální aritou 1 představují nejčastěji matematické funkce a boolovské operace a jeho listové uzly představující terminály, které mají aritu 0. Obvykle se jedná o proměnné a konstanty. Selekce jedinců probíhá na principu turnaje. Proces křížení spočívá v záměně podstromů dvou rodičů a mutace jako náhodná změna podstromu za nově vygenerovaný podstrom, nebo změna určitého uzlu.

Do nové generace jsou vybírání pouze nově vzniklí potomci, tzv. generační obměna. Vzhledem k velmi velkému stavovému prostoru možných stromů bývá obvyklá velmi velká populace (1000 jedinců a více), ale naopak malé množství počtu generací (50 a méně). Také vzhledem k neomezené délce stromové struktury, tj. velikosti jedince (zejména při křížení se velikost jedince značně zvětšuje, protože relativně malý podstrom jednoho rodiče může být vyměněn za mnohem větší podstrom druhého rodiče), často dochází k tzv. *Bloat* problému, kdy při nadměrném zvětšování šířky a hloubky stromu se jeho fitness prakticky nemění, případně degraduje. To zapříčiňuje vysokou výpočetní náročnost tohoto algoritmu, jelikož provedení programu a tím zjištění jeho fitness se stává stále více náročnější. Existuje mnoho přístupů, jak tento problém řešit, například omezením hloubky stromu jedince a následným ořezáním všeho, co překračuje tuto hloubku. Nebo například penalizací příliš dlouhých jedinců. [22] Princip reprezentace jedince a jeho následné křížení je znázorněno na obrázku 2.5.



Obrázek 2.5: Princip křížení a mutace u genetického programování. Křížení podstromů dvou rodičů a mutace náhodného uzlu potomka

2.3.4 Diferenciální evoluce

Poslední variantou z významných evolučních algoritmů je tzv. diferenciální evoluce. Její název není odvozen od diferenciálního počtu, ale od obyčejné diference (rozdílu) dvou vektorů. V tomto algoritmu jsou jedinci reprezentováni vektorem reálných hodnot. Mírně se odkloňuje od principu fungování výše zmíněných algoritmů, avšak dosahuje jedněch z nejlepších výsledků napříč evolučními algoritmy. Mezi výhody patří její vysoká rychlost i při vícerozměrném stavovém prostoru, snadná implementace a v neposlední řadě také fakt, že dokáže velice obstojně uniknout z lokálních extrémů a v případě, že má funkce více globálních extrémů, obvykle je s velkou pravděpodobností nalezne.

Principiálně funguje následujícím způsobem. (Počty vybraných jedinců z populace a způsob tvorby šumového vektoru odpovídají variantě algoritmu *Rand/1.*) Nultá generace je vygenerována zcela náhodně. Jedinec pro následující generaci je tvořen ze 3 jedinců aktuální generace. Nejprve se vyberou 3 náhodní jedinci a k jednomu z nich je přičten rozdíl zbylých dvou vynásobený mutační konstantou F. Tento proces se nazývá diferenciální mutace a vznikne ním tzv. *šumový vektor*.

Tento šumový vektor se následně zkříží s vybraným rodičem (aktivním jedincem). Křížení je podle binomického nebo exponenciálního rozdělení a práh křížení CR nabývá náhodné hodnoty z intervalu $(0, 1)$. Takto nově vytvořený jedinec se nazývá tzv. *zkušební vektor*, na který je aplikována funkce fitness a jeho výsledná hodnota je porovnána s fitness *aktivního jedince*. Pokud má *zkušební vektor* lepší hodnotu fitness, je vybrán do nové generace. Pokud má horší hodnotu, je naopak zachován *aktivní jedinec*, který beze změny postupuje jako nový jedinec do budoucí generace. Tento postup se opakuje do velikosti populace N. Tzn. každý jedinec z aktuální generace vytvoří spolu s pomocnými jedinci jeden zkušební vektor se kterým je následně porovnán. Tímto je provedena jedna iterace algoritmu. Celý algoritmus končí splněním ukončovací podmínky a jedinec s nejlepším fitness je prohlášen za řešení dané optimalizační úlohy [17], [33].

Existuje mnoho variant diferenciální evoluce a každá vyniká v jiném typu úloh. Můžeme je dále rozdělit podle mutačních schémat a schémat křížení.

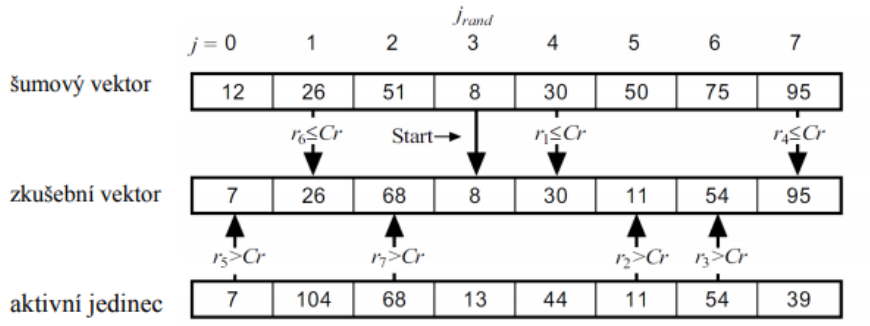
Některé varianty mutačních schémat:

- *Rand/1/XXX:* $x_{mut} = x_1 + F(x_2 - x_3)$
- *Rand/2/XXX:* $x_{mut} = x_1 + F(x_2 - x_3 + x_4 - x_5)$
- *Best/1/XXX:* $x_{mut} = x_{best} + F(x_2 - x_3)$
- *Best/2/XXX:* $x_{mut} = x_{best} + F(x_2 - x_3 + x_4 - x_5)$
- *Rand-to-best/1/XXX:* $x_{mut} = x_1 + F_1(x_2 - x_3) + F_2(x_{best} - x_1)$

Kde F_i označuje mutační konstantu, x_{mut} šumový vektor, x_i vybrané jedince pro tvorbu šumového vektoru a x_{best} nejlepšího jedince z aktuální populace.

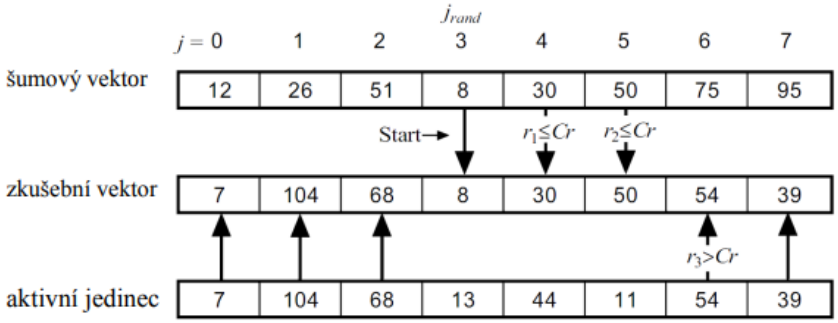
První parametr označuje typ výběru aktivního jedince (*Rand, Best*), druhý je počet rozdílů dvou vektorů (*1, 2*) a poslední *XXX* označuje typ křížení, tj. podle jakého rozložení určujeme které geny se u obou jedinců vymění s geny mutujícího jedince (šumového vektoru) a které nikoliv. Tato rozdělení jsou:

- *Binomické (Bin)* - křížení na základě binomického rozdělení probíhá tím způsobem, že každý gen je s pravděpodobnostní CR změněn na korespondující gen mutujícího jedince (šumového vektoru). Počáteční gen je zvolen náhodně.



Obrázek 2.6: Binomické křížení u DE, r_j jsou náhodná čísla, CR je práh křížení, šipky naznačují z kterého vektoru byl daný gen vybrán do zkušebního vektoru [Převzato z: [17]]

- *Exponenciální (Exp)* - jedná se o obdobu dvoubodového křížení, kdy se náhodně vybere gen, od kterého se při $r_i \leq CR$ kopírují hodnoty z šumového vektoru až do té doby, dokud nepřestane tato podmínka platit, poté je zbylá část genů zkopírována z aktivního jedince



Obrázek 2.7: Exponenciální křížení u DE, r_j jsou náhodná čísla, CR je práh křížení, šipky naznačují z kterého vektoru byl daný gen vybrán do zkušebního vektoru [Převzato z: [17]]

2.3.5 Optimalizace hejnem částic

Algoritmus PSO *Particle swarm optimization* je optimalizační metoda inspirovaná biologickým chováním hejn ptáků při letu. Můžeme ji zařadit do oblasti technik nazývaných inteligence hejna. Byla poprvé publikována v roce 1995. Jedná se o implementačně velmi jednoduchou metodu. Její síla mimo jiné také spočívá v tom, že poměrně rychle dochází ke konvergenci řešení k optimální hodnotě.

Každá částice v rámci populace je definována svou pozicí, směrovým vektorem *inertia*, který ukazuje na směr lepšího řešení a pamětí svého nejlepšího řešení. Globálně si potom

algoritmus pamatuje také pozici a hodnotu nejlepšího řešení v rámci celé populace. Nová pozice každé částice je tedy závislá jak na svém dosavadním nejlepším řešení a hodnotě aktuálního řešení, tak globálním nejlepším řešení v rámci celé populace. Při tomto algoritmu nedochází k operacím křížení ani mutace. Pouze je v každém kroku spočítána hodnota fitness všech jedinců, následně jsou pro ně vypočítány nové hodnoty směrového vektoru rychlosti a na jeho základě je vypočítána jejich nová pozice [16].

for *každou částici i* **do**
  for *každou dimenzi d* **do**
    Inicializuj pozici x_{id} náhodně v rámci povoleného rozmezí;
    Inicializuj rychlost v_{id} náhodně v rámci povoleného rozmezí;
  end
end
Iterace $k = 1$;
while *Nesplněna ukončující kritéria (chyba, počet iterací)* **do**
  for *každou částici i* **do**
    Spočítej hodnotu fitness;
    if *Hodnota fitness je lepší než* p_i^{Best} **then**
      Nastav stávající hodnotu fitness jako p_i^{Best};
    end
  end
  Vyber částici s nejlepší hodnotou fitness jako g^{Best};
  for *každou částici i* **do**
    for *každou dimenzi d* **do**
      Vypočítej směrový vektor podle rovnice:;
      $v_{id}(k+1) = w * v_{id}(k) + c_1 * rand_1(p_{id}^{Best} - x_{id}) + c_2 * rand_2(g_d^{Best} - x_{id})$;
      Aktualizuj pozici částice podle rovnice:;
      $x_{id}(k+1) = x_{id} + v_{id}(k+1)$;
    end
  end
  $k = k + 1$;
end

Algorithm 3: Pseudokód algoritmu PSO - kde v_i je směrový vektor, x_i je pozice částice, p_i je nejlepší řešení částice i a g je nejlepší řešení v rámci populace. Parametry w určuje sílu vlivu směrového vektoru, c_1 sílu vlivu nejlepšího řešení aktuální částice a c_2 sílu vlivu nejlepšího řešení v populaci. $rand_{1,2}$ jsou náhodné hodnoty.

Kapitola 3

Neuronové sítě

Umělé neuronové sítě jsou oblastí umělé inteligence, která má za cíl napodobovat principy fungování lidského mozku a centrální nervové soustavy. Lidský mozek dokáže zpracovat obrovské množství informací v poměrně malém časovém intervalu. Jeho základní stavební jednotku nazýváme neuron. Těchto neuronů obsahuje kolem 10 - 15 miliard a každý je propojen přibližně s tisícovkou dalších. Dnešní umělé neuronové sítě jsou stále velmi vzdálené svou komplexností a velikostí od těch reálných. Nicméně již dnes se úspěšně využívají v oblastech zaměřených především na rozpoznávání, klasifikaci a predikci hlasových, obrazových nebo audiovizuálních dat.

Umělé neuronové sítě pracují na principu tzv. černé skříňky *black box*, kdy nevíme, co přesně se děje uvnitř sítě při její činnosti. Na začátku je potřeba takovou síť jednorázově naučit na vstupních datech z datové sady *datasetu*. Taková data označujeme jako trénovací sada (training set) a typicky představují data podobná těm, která bude naučená síť zpracovávat. Toto učení spočívá typicky v úpravách hodnot spojení, dále jen (vah) mezi jednotlivými neurony. Můžeme říci, že neuronové sítě pracují jako funkce. Na základě nějakého vstupu produkují určitý výstup. Tímto výstupem rozumíme např. v případě neuronové sítě vytvořené pro klasifikaci vstupních dat, určení jedné klasifikační kategorie do které spadá konkrétní vstupní vzorek. Fáze učení je zpravidla velmi výpočetně i časově náročná operace, jelikož zde dochází k velkému množství výpočtů nad každým vzorkem. Síla sítě je zpravidla úměrná velikosti vstupního trénovacího setu a také různorodosti těchto vzorků.

Po fázi učení, kdy jednotlivé váhy mezi neurony postupně konvergují na svou optimální hodnotu, je neuronová síť připravena ke svému používání. Samotné používání naučené sítě již nepředstavuje výraznější výpočetní zátěž na procesoru a je zpravidla velmi rychlé.

Tato kapitola vychází z těchto zdrojů: [35], [24], [21], [37].

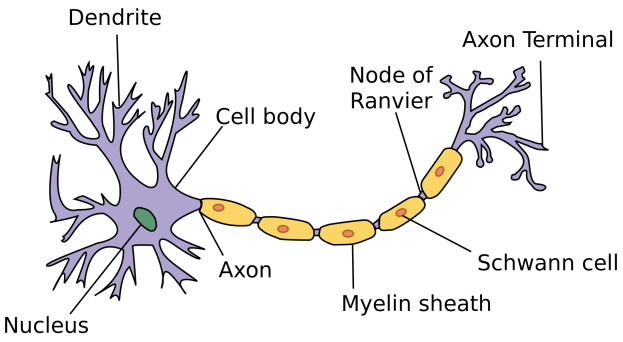
3.1 Model neuronu

Neuron je základní stavební jednotka v lidském mozku. Stejně tak je umělý neuron základní stavební jednotkou umělých neuronových sítí. Oba mají nějaké vstupy, ze kterých po přijetí informací vysílají na výstup svoji odezvu. Bývají složeny do komplexních sítí, ve kterých pak pracují mnohem efektivněji.

3.1.1 Biologický neuron

Biologický neuron na svém vstupu obdrží elektrický vzruch, který *receptorem* příjme, zpracuje a následně *neutransmiterem* odešle svou odezvu opět jako elektrický vzruch dalším

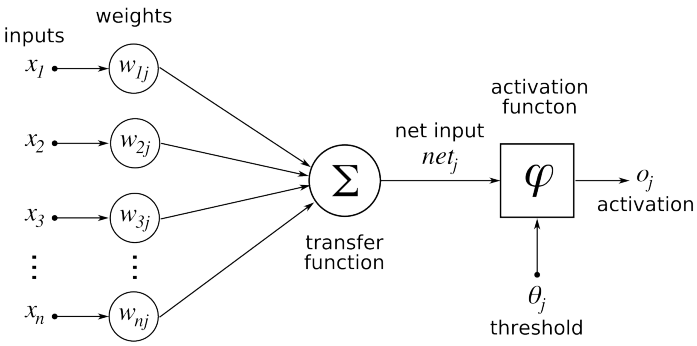
neuronům se kterými je spojen. Posílání těchto signálů mezi neurony je přenášeno přes speciální spoje zvané *synapse*. V těchto synapsích je také uložena informace. Takto neurony mezi sebou komunikují a vytvářejí obrovské okruhy a sítě spolu navzájem propojených neuronů. Typický neuron se skládá ze svého těla (*soma*), ze svého výstupu (*axon*), který je zpravidla pouze jeden (případně žádný, pokud neuron slouží pouze jako propojující) a ze svého vstupu zvaného *dendrit*. Na dendrity jsou připojeny synapse z ostatních neuronů [6]. Blíže je stavba neuronu popsána na obrázku níže. 3.1



Obrázek 3.1: Biologický model neuronu Převzato z: [6]

3.1.2 Umělý neuron

Model umělého neuronu viz Obrázek 3.2 má na vstupu n vstupů, každému z těchto vstupů xi je přiřazena určitá váha w_i. Vstupní informace ze vstupu i je tedy rovna hodnotě $w_i x_i$. Tyto vstupní hodnoty neuronu jsou poté vstupem přenosové funkce, kterou nejčastěji bývá funkce sumy, která má na výstupu pouze jednu hodnotu. Ta je vstupem do aktivační funkce, na základě které je určena hodnota výstupu celého neuronu. Aktivačních funkcí existuje celá řada a více si je rozebereme v sekci 3.1.3.



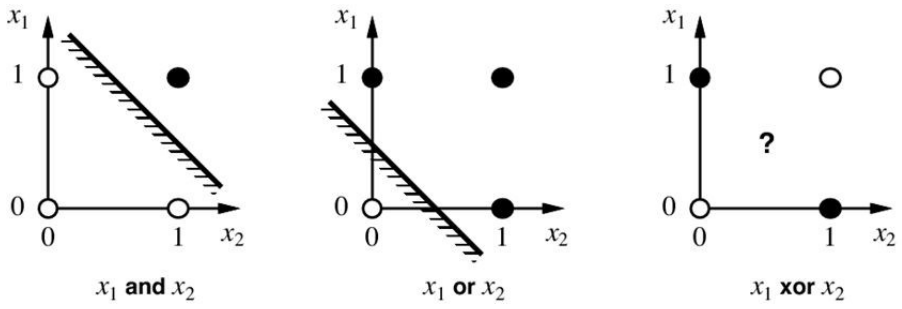
Obrázek 3.2: Umělý model neuronu

Perceptron

Perceptron je nejjednodušší umělá neuronová síť sestávající se pouze z jednoho neuronu. Tj. vstup neuronu je zároveň vstupem celé sítě a jeho výstup je taktéž výstupem celé sítě. Jednoduchou výstupní funkci můžeme definovat následovně:

$$f(x) = \begin{cases} 1 & \text{pro } \sum_{i=1}^{n} w_i x_i \geq threshold \\ 0 & \text{pro } \sum_{i=1}^{n} w_i x_i < threshold \end{cases}$$

Často chceme hodnotu prahu (*threshold*) snadno měnit, proto se využívá jeho hodnota jako vstupní hodnota neuronu na pozici $x_0 = -1$ a $w_0 = 0$. Díky tomu můžeme sumační operátor indexovat od nuly. Váhy perceptronu se mění v závislosti na rozdílu výstupní hodnoty perceptronu a výstupní hodnoty předem očekávané. Tímto určujeme chybu sítě. Během procesu učení se jednotlivé váhy perceptronu mění, dokud nekonvergují na své stálé hodnoty. Tento vektor hodnot vah pak představuje globální minimum funkce všech vah sítě. Nicméně toto tvrzení platí pouze u lineárně separovatelných problémů. U složitějších problémů (například funkce XOR) nám již k nalezení řešení nestačí využití pouze jednoho neuronu, a tak je potřeba přidat další vrstvy. Tím nám vznikají vícevrstevné neuronové sítě viz 3.2.



Obrázek 3.3: Jednotkový graf lineární separovatelnosti jednoduchých logických funkcí AND a OR a nelineárně separovatelné funkce XOR

3.1.3 Aktivační funkce

Aktivační funkce slouží k aktivaci neuronu. Obecně mohou být tyto funkce skokové nebo spojité a udávají, při jaké hodnotě vstupu je neuron aktivován a propaguje tak svou hodnotu dále. Nejčastěji užívanou funkcí při ukázkách a jednoduchých experimentech s neuronovými sítěmi bývá funkce *sigmoid*. V poslední době se však ukázala jako lépe vyhovující funkce ReLU 3.1.3 (mj. z toho důvodu, že může nabývat funkčních hodnot z většího intervalu), proto již dnes u většiny serióznějších neuronových sítí je využíváno právě této funkce. Přehled nejčastějších aktivačních funkcí nalezneme v obrázku 3.6 níže.

Sigmoid

Asi nejznámější aktivační funkcí je funkce Sigmoid. Jedná se o nelineární funkci, která dává na výstupu hodnoty z intervalu $(0, 1)$. Tímto se velice podobá reálnému fungování biologického neuronu. Při vysokých hodnotách na vstupu funkce se výstupní hodnoty velmi blíží

jedničce a naopak při vysokých záporných hodnotých k nule. Tímto se na tuto funkci váže také problém *vanishing gradient*, neboli mizející gradient. V praxi to znamená, že derivace v bodech vzdálených od osy y jsou velmi blízké nule, a tak je například pro algoritmus Back-propagation velmi pomalé a náročné síť učit. Tento problém se řeší většinou použitím výkonnějšího stroje nebo naopak prvotním naučením sítě za použití *Unsupervised learning* (učení bez učitele) a následném doladění vah pomocí algoritmu Back-propagation nebo použitím jiné aktivační funkce. Funkce Sigmoid je definována předpisem:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Případně obdobně známá aktivační funkce Hyperbolický Tangent, liší se od funkce Sigmoid prakticky jen rozsahem na ose y tj $y\eta(-1, 1)$, nabízí tak větší rozsah pro aktivaci neuronu. Funkce je definována takto:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

ReLU

Aktivační funkce *Rectified Linear Unit* je nejužívanější funkcí v moderních neuronových sítích. Je to zejména proto, že významně urychluje chod sítě, protože používá méně výpočetních operací než funkce Sigmoid nebo Hyperbolický tangent. ReLU při kladných hodnotách x propaguje stejnou hodnotu na výstup a naopak záporné hodnoty propaguje jako nulu. Díky tomu tak ušetříme průměrně 50% početních operací oproti výše zmíněným funkcím. Jediným nedostatkem této funkce je fakt, že při záporných hodnotách na vstupu je výsledný gradient nulový, což s sebou přináší problém zvaný *dying ReLU*. Tento problém ovšem řeší mírně upravená funkce *Leaky ReLU*, která pouze místo nuly na výstupu při záporném vstupu dává hodnoty velmi blízké nule, tak aby gradient nebyl nulový a tím daný neuron nepřestal zcela reagovat. Obecná funkce ReLU je tedy definována takto:

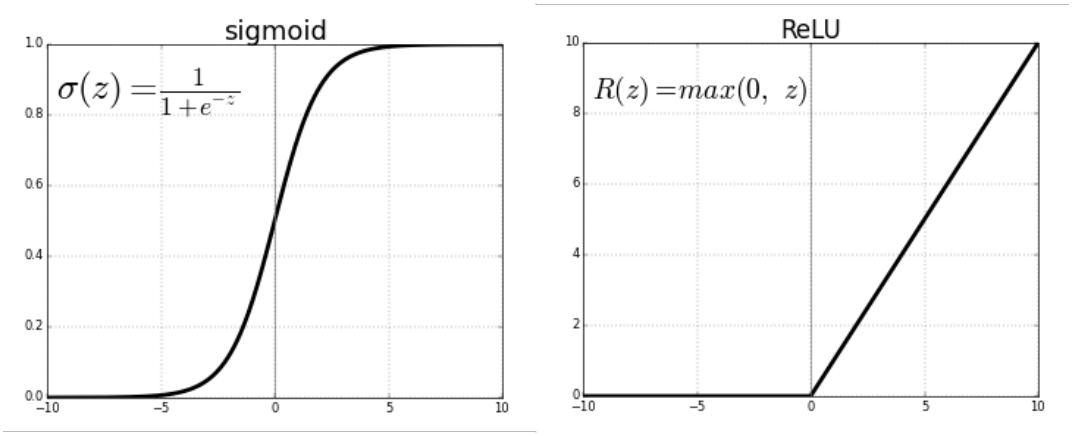
$$f(x) = \begin{cases} 0 & \text{pro} \quad x < 0 \\ x & \text{pro} \quad x \geq 0 \end{cases}$$

Softmax

Funkce softmax se obvykle užívá u sítí řešících klasifikační problémy. Máme předem připraven konečný počet definovaných tříd klasifikace a zkoumaný vzorek odpovídá pouze jediné z nich. Funkce softmax se tak užívá prakticky pouze v poslední vrstvě neuronové sítě, kdy zajišťuje, že suma hodnot neuronů ve výstupní vrstvě je rovna jedné. Čili představuje pravděpodobnost, s jakou právě zkoumaný vzorek odpovídá daným klasifikačním třídám. Funkce softmax je obecně definována takto:

$$SM(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

Kde $j\epsilon(1, N)$ a N určuje rozměr vektoru. Vektor SM tak dává v součtu hodnotu 1.



Obrázek 3.4: Aktivační funkce Sigmoid a ReLU

3.1.4 Chybové funkce

Chybové funkce se využívají v poslední vrstvě neuronové sítě a slouží jednak k určení přesnosti sítě, ale také ke zpětné propagaci této chyby přes všechny neurony až k vrstvě vstupní. Šířením právě hodnot z chybové funkce se neuronová síť učí.

Čtvercová chyba

Nejznámnější chybovou funkcí je funkce MSE (*Mean squared error*) neboli čtvercová chyba. Odečte reálnou hodnotu na výstupu sítě od očekávané hodnoty a pro odstranění záporných hodnot je tento rozdíl umocněn na druhou. Tímto se uměle zvyšuje rozdíl chyby mezi správně a nesprávně klasifikovaným vzorkem, a tím tak dopomáhá k rychlejšímu natrénování. Funkce čtvercové chyby je definována takto:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(d_i - y_i)^2$$

Kde N je počet vzorku z trénovací sady, d_i je reálná hodnota a y_i je hodnota na výstupu sítě.

Křížová entropie

Další známou chybovou funkcí je funkce Křížová entropie (Cross-Entropy). Využívá se při klasifikaci do tříd, kdy ve výstupní vrstvě sítě je počet neuronů $C \geq 2$.

Obvykle se využívá ve spojení s aktivační funkcí Softmax, která rozloží hodnoty na výstupu poslední vrstvy do pravděpodobnostní náležitosti do jednotlivých tříd v součtu tvořících jedničku, tedy 100% pravděpodobnost celku.

Pro $C > 2$ je cross-entropy definována takto:

$$CE = -\sum_{i}^{C} d_i \log(y_i)$$

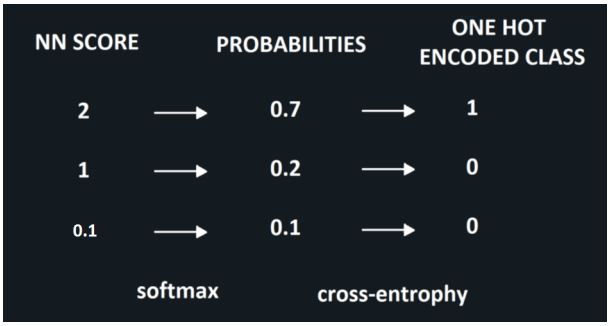
Kde d_i je reálná hodnota a y_i je hodnota na výstupu sítě.

Pro binární počet klasifikačních tříd, tedy $C = 2$ je definována takto:

$$CE = -\sum_{i=1}^{C=2} d_i \log(y_i) = -d_i \log(y_i) + (1 - d_i) \log(1 - y_i)$$

Kde opět d_i je reálná hodnota a y_i je hodnota na výstupu sítě

Čím nižší je hodnota křížové entropie, tím je síť lépe naučena a produkuje přesnější predikce. Při dokonale naučené síti je hodnota křížové entropie rovna nule. Tedy pokud je u daného vzorku hodnota správné klasifikační třídy rovna jedné a ostatní třídy rovny nule [27].



Obrázek 3.5: Zpracování vzorku v poslední vrstvě neuronové sítě obsahující 3 klasifikační třídy. Pomocí funkcí softmax (převede na pravděpodobnostní hodnoty tvořící v sumě jedničku) a cross-entropy (nejsilnější třídu převede na jedničku, zbylé třídy na nulu) [Převzato z: [26]]

3.1.5 Back-propagation

Back-propagation neboli zpětné šíření chyby je algoritmus patřící do třídy gradientních metod a zajišťuje u vícevrstevných sítí jejich učení, tj. změnu hodnot vah a biasů jednotlivých vrstev. Patří mezi nejpoužívanější algoritmy pro učení dopředných neuronových sítí.

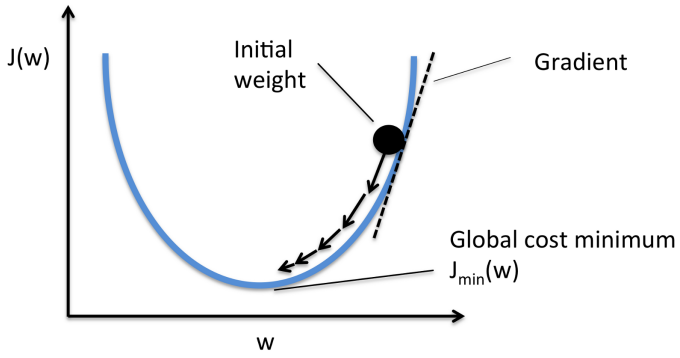
Back-propagation využívá tzv. *chain-rule*, řetězového pravidla právě pro postupné šíření chyby gradientu u jednotlivých vrstev, směrem od výstupní vrstvy až po vrstvu vstupní. Postupně tedy upravuje hodnoty jednotlivých vah, dokud nedojde k lokálnímu, případně globálnímu minimu. Algoritmus obsahuje také parametr *learning rate* (míra učení) v intervalu $\langle 0, 1\rangle$, který určuje rychlost učení sítě. Při jeho vyšší hodnotě se hodnoty vah modifikují výrazněji, avšak díky tomu mohou uváznout v lokálním minimu, ze kterého se již nemusejí být schopny vymanit. Obvyklá hodnota parametru je kolem $0,2$. Konkrétní fungování algoritmu ukazuje pseudokód níže 4.

Jednou z podmínek pro fungování algoritmu je, že aktivační funkce každého neuronu musí být diferencovatelná, tzn. musíme být schopni spočítat její derivaci. Algoritmus využívá postupného sestupu, neboli *gradient descent*. Tento optimalizační algoritmus se využívá pro hledání minima nějaké funkce. Využívá toho, že ve směru nejvyššího gradientu roste i nejvíce chyba. Čili pro hledání minima funkce, algoritmus postupuje přesně proti směru tohoto gradientu. Algoritmus postupuje po krocích, které jsou přímo úměrné velikosti gradientu [35].

Úpravy vah se řídí dle následujícího vzorce:

$$w_{i+1} = w_i - \eta \nabla G(w_i)$$

Kde w je hodnota váhy, η *learning rate* neboli míra učení a ∇G je hodnota gradientu.[26]



Obrázek 3.6: Gradient descent - iterativní postup hodnot vah směrem k jejich globálnímu minimu. (w je váha, J_w je hodnota chyby) [Převzato z: [14]]

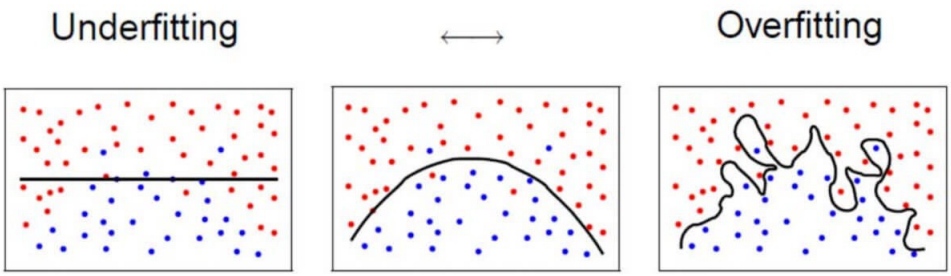
Inicializuj všechny váhy mezi neurony v síti na náhodnou hodnotu z rozmezí $\langle -1; 1 \rangle$;
while *Dokud není splněno ukončovací kritérium (počet iterací, hodnota chyby)* **do**
  for *Každý vzorek z trénovací množiny* **do**
    //Propagace vstupu směrem dopředu od vstupu k výstupu;
    for *Každou vrstvu v síti* **do**
      for *Každý neuron ve vrstvě* **do**
        1. Spočítej sumu vah se vstupy přicházejícími do neuronu;
        2. Přidej bias do sumy;
        3. Spočítej aktivační funkci pro daný neuron;
      end
    end
    //Propagace chyby směrem zpět od výstupu ke vstupu;
    for *Každý neuron ve výstupní vrstvě* **do**
      Spočítej hodnotu chyby;
    end
    for *Každou skrytou vrstvu* **do**
      for *Každý neuron ve skryté vrstvě* **do**
        1. Vypočítej celkovou chybu neuronu;
        2. Aktualizuj hodnoty všech vah daného neuronu;
      end
    end
    //Výpočet globální chyby;
    Spočítej chybovou funkci;
  end
end

Algorithm 4: Pseudokód algoritmu Back-propagation

Overfitting

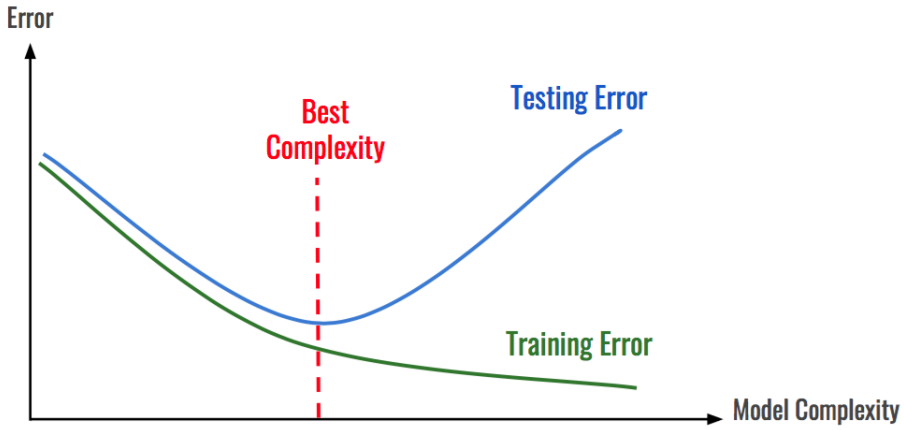
S algoritmem back-propagation se pojí i pojem přetrénování (*overfitting*). Jedná se o stav, kdy je neuronová síť naučena až příliš přesně, tedy její hodnoty vah jsou na svých optimálních hodnotách, a tak síť vykazuje přesnost klasifikace blížící se 100% přesnosti. Nicméně tato přesnost se vztahuje pouze k trénovací množině vzorků, pro kterou se síť až příliš specifikovala a naopak přestala zobecňovat na neznámých vzorcích. Na validačním setu pak dochází k jevu, že takto přeučená síť vykazuje mnohem menší přesnost, než síť, která dokazovala daleko menší přesnosti při trénovacím setu [21].

Problém přílišné generalizace a naopak přetrénování při učení neuronové sítě nám blíže ukazuje následující Obrázek 3.7.



Obrázek 3.7: Generalizace vs ideální klasifikace vs přetrénování - černá křivka vyjadřuje rozdělení vzorků do dvou klasifikačních tříd při jednotlivých problémech při učení neuronových sítí a celkově algoritmech zaměřených na klasifikaci

Řešením tohoto problému je obvykle dostatečně různorodý a početně velký trénovací set. Brzké zastavení procesu učení při počínajícím se zhoršování přesnosti sítě na validačním setu. Nebo tzv. metodou *dropout*. Spočívá v tom, že se při fázi učení některé neurony ve skrytých vrstvách náhodně vyřazují z činnosti. Váhy vedoucí z těchto neuronů jsou poté vynásobeny pravděpodobností s jakou byly dané neurony vyřazeny z činnosti při fázi učení [37].



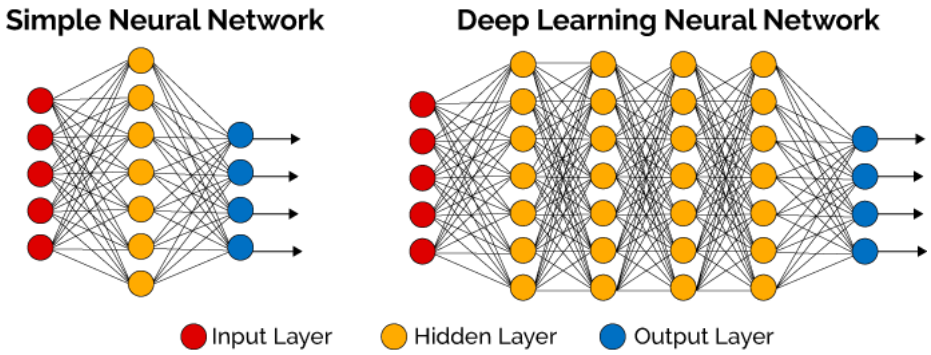
Obrázek 3.8: Přetrénování - červená svislá čára označuje ideální moment pro ukončení fáze učení, při zachování nejvyšší přesnosti sítě. Modrá křivka ukazuje průběh přesnosti sítě na validačních datech. Zelená křivka ukazuje průběh přesnosti na trénovacích datech, kdy postupně s nabývající komplexností modelu - tedy až příliš optimálními hodnotami vah sítě sice můžeme dosáhnout minimalizace chyby na těchto datech, nicméně chyba na validačních datech se dramaticky zvýší.

3.2 Vícevrstevné sítě

Topologie neuronové sítě značí způsob propojení jednotlivých neuronů ve vícevrstevné síti. Vrstvy sítě existují 3 základní typy a to **vstupní vrstvy**, **skryté vrstvy** a **vrstvy výstupní**.

Dále rozlišujeme dva hlavní typy sítí, podle typu propojení a to dopředné a rekurentní. U dopředných sítí jsou vždy výstupy neuronu jedné vrstvy připojeny jako vstupy vrstvy následující. U rekurentních sítí je možnost propojení výstupu jednoho neuronu na svůj vlastní vstup, případně na vstup neuronu v jeho vrstvě. Tzn. výstupy jedné vrstvy mohou být ovlivněny svým vlastním výstupem, někdy též nazýváno jako zpětnovazební přenos. Jako obecný příklad využití rekurentních sítí jsou oblasti, kdy záleží na pořadí vstupů, tzn. zajímá nás časová složka dat. Patří do nich oblasti klasifikace, rozpoznávání nebo predikce v obrazových (video) či hlasových datech.

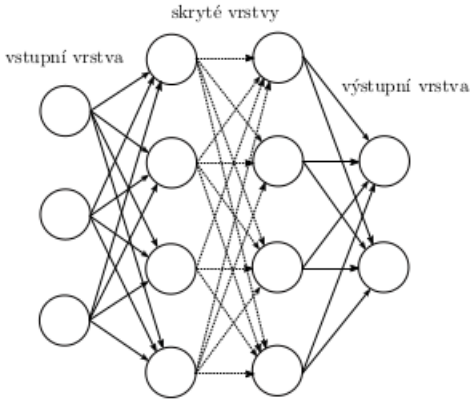
Sítě také rozdělujeme na husté a řídké, podle hustoty propojení mezi jednotlivými vrstvami. Husté sítě znamenají, že všechny výstupy neuronů jedné vrstvy jsou propojeny se všemi vstupy neuronů následující vrstvy. Naopak u řídkých sítí mohou být propojeny mezi sebou pouze určité neurony.



Obrázek 3.9: Vícevrstevné sítě - rozdíl mezi jednoduchou a hlubokou sítí je v počtu skrytých vrstev L, pro které musí platit $L \geq 2$ pro hlubokou síť

3.2.1 Dopředné sítě

Dopředné sítě neboli *Feed-Forward* jsou nejběžnější typ neuronových sítí. Jsou konstruovány takovým způsobem, že výstupy jedné vrstvy jsou zároveň vstupy vrstvy následující, tudíž na rozdíl od rekurentních sítí neobsahují žádné zpětné smyčky. U těchto sítí tak není problém využití algoritmu back-propagation pro její natrénování. V této práci se budeme zabývat trénováním a optimalizací pouze těchto dopředných sítí. Následující obrázek ukazuje základní topologii takové dopředné sítě 3.10.



Obrázek 3.10: Příklad dopředné neuronové sítě se dvěma skrytými vrstvami

Výpočet aktivace jednotlivých neuronů v rámci sítě je potom spočítán podle vzorce:

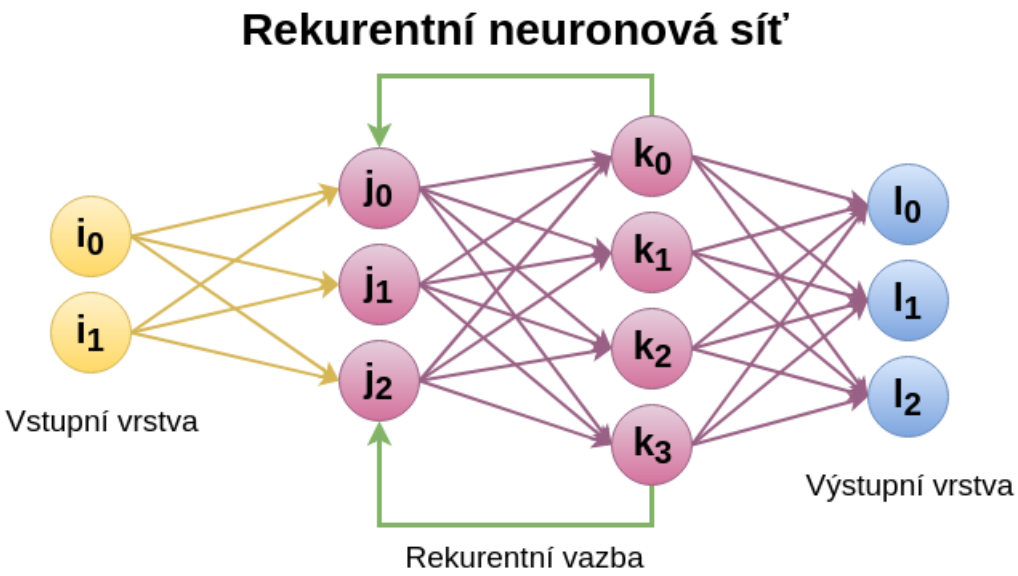
$$a_j = \sigma(w_{i,j} * x_i + b_j)$$

Kde a_j je neuron ve vrstvě j, $w_{i,j}$ je váha mezi vrstvami i a j, b_j je bias pro vrstvu j a σ je aktivační funkce - obvykle funkce *Sigmoid*.

3.2.2 Rekurentní sítě

Rekurentní sítě, jak již bylo zmíněno obsahují smyčky, kdy výstupy jedné vrstvy jsou zároveň svými vstupy. V nejjednodušším případě je tedy výstup vrstvy i závislý na výstupu

vrstvy $i + 1$. Tyto sítě umožňují použití u mnohem více komplexnějších úloh než obyčejné dopředné sítě, avšak ne vždy se hodí pro jakýkoliv problém. Při použití rekurentních sítí se také mnohem více zvětšuje prohledávaný prostor řešení, a tak jsou rekurentní sítě daleko více náročné na výpočetní čas. Často se využívají u architektur obsahujících umělé agenty nebo roboty, jelikož rekurentní spojení neuronů umožňují využití paměti při rozhodovacím procesu. Využití takovýchto sítí tedy silně záleží na jejich aplikaci a je důležité si položit otázku, jestli je jejich použití nezbytné a vhodné pro daný problém. V neposlední řadě je také nutná vhodně zvolená reprezentace vstupních dat.

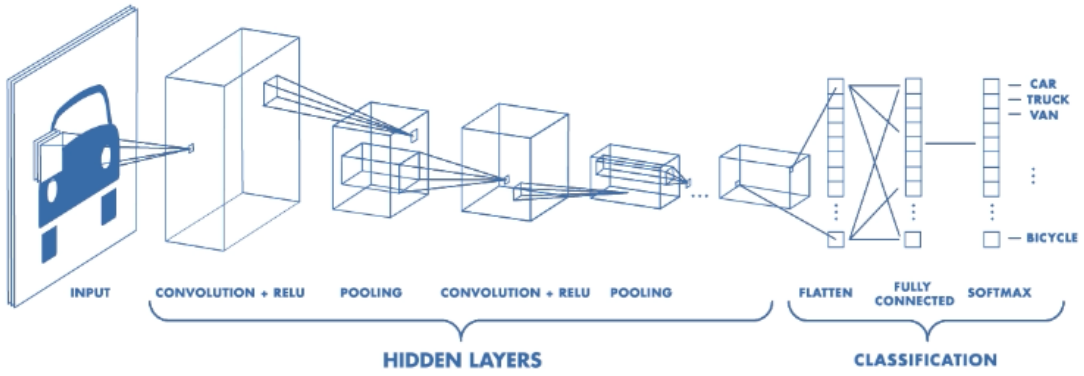


Obrázek 3.11: Rekurentní síť - můžeme si všimnout zpětných vazeb (zeleně) - díky těmto vazbám tak výstup skryté vrstvy j závisí na výstupu následující skryté vrstvy k, a tak není možné užít klasický učící algoritmus Back-propagation

Díky zpětným smyčkám není možné užít klasický Back-propagation algoritmus, ale je potřeba upravené metody zvané Back-propagation through time (BPTT), která již tyto zpětné smyčky zohledňuje.

3.2.3 Konvoluční sítě

Jiným druhem dopředných neuronových sítí jsou sítě, které obsahují mimo klasických neuronových vrstev také vrstvy složené z konvolučních filtrů. Tyto sítě slouží především pro detekci nebo rozpoznávání objektů v obraze, kdy jejich konvoluční vrstvy dokáží extrahovat ze vstupního obrazu určité vzory, se kterými tato síť dále pracuje.



Obrázek 3.12: Příklad postupu práce konvoluční neuronové sítě. Můžeme si všimnout ve skrytých vrstvách mimo konvolučních vrstev také tzv. *pooling vrstev*, které slouží k postupnému snižování dimenze vstupních dat z neuronů předchozích vrstev do jediného neuronu vrstvy následující. Jedná se o klíčový prvek konvolučních sítí. [Převzato z: [11]]

Kapitola 4

Spojení Evolučních algoritmů a Neuronových sítí

Oblast umělé inteligence, kde dochází ke spojení umělých neuronových sítí spolu s evolučními algoritmy se nazývá *neuroevoluce*. Nejčastěji jsou tyto dvě oblasti spojeny v tom smyslu, že evoluční algoritmus zajišťuje způsob natrénování, případně výstavbu topologie neuronové sítě, která následně pracuje jako klasická neuronová síť [15].

Využití neuroevoluce můžeme tedy hledat všude tam, kde je využití klasických neuronových sítí. Jde tedy o oblasti rozpoznávání, klasifikace, predikce atd. Velká síla neuroevoluce však spočívá v tom, že alespoň teoreticky by natrénování sítě pomocí evolučních algoritmů nemělo býti tak výpočetně a časově náročné, jako u standardního algoritmu back-propagation. Dále také při využití neuroevoluce by nemělo docházet při fázi učení k uváznutí v některém z lokálních optim, tak jako je to u back-propagation poměrně často viditelné.

Také jednou z budoucích výhod neuroevoluce bude jistě i to, že z procesu návrhu neuronové sítě zcela vytěsní lidský faktor, který je doposud potřeba při návrhu složitých neuronových sítí a všech jejich parametrů. Tento proces je zdlouhavý a spíše připomíná metodu pokus omyl. Zejména lze toto vidět u návrhu konvolučních neuronových sítí, které zpracovávají různá obrazová data a obsahují velké množství různě velkých vrstev, kdy každá plní jinou funkci. Tím se nalezení optimální konfigurace sítě pro člověka stává téměř nemožná. V tomto ohledu tedy v budoucnu minimálně takové využití neuroevoluce v praxi zcela jistě uvidíme.

V této kapitole si rozebereme základní principy možné spolupráce a následně si krátce chronologicky představíme již existující algoritmy. Budeme vycházet z těchto zdrojů: [10], [32], [7], [18], [19], [13].

4.1 Příklady spolupráce

Jako vhodné možnosti spolupráce evolučních algoritmů a neuronových sítí se nabízí více variant, které můžeme rozdělit na varianty s fixní a na varianty s proměnlivou topologií neuronové sítě.

4.1.1 Fixní topologie

Variant spolupráce evolučních algoritmů a neuronových sítí s fixní topologií může být hned několik. Jelikož se bude muset programátor zaobírat tím, jak vystavět neuronovou síť, kolik

skrytých vrstev by měla síť obsahovat a také, jak hustě by měly jednotlivé vrstvy být propojeny, je tento přístup do jisté míry omezující a prakticky pouze nahrazuje proces učení, případně ovlivní kvalitu výsledků u takovéto topologie. Tímto přístupem ovšem nedosáhneme optimální sítě pro zadanou úlohu, ale pouze námi již navrženou síť naučíme, případně vylepšíme její chování [15].

Mezi hlavní varianty evoluce neuronové sítě můžeme zařadit tyto přístupy:

- **Změna vah mezi jednotlivými neurony** - jedinec v evolučním algoritmu bude síť reprezentovaná jako matice jejich vah, ty bude evoluce optimalizovat. Jako fitness funkce by byla použita odezva sítě na validačních datech. Toto řešení tedy nahrazuje algoritmus back-propagation [28].

- **Změna aktivačních funkcí u jednotlivých neuronů** - jedincem v evolučním algoritmu by byla opět síť, avšak jejími chromozomy by byly aktivační funkce jednotlivých neuronů. Tento přístup ovšem nenahrazuje proces učení, a tak pouze napomáhá teoretickému zlepšení vlastností sítě. Může se stát velice časově výpočetně náročným, neboť pro každého jedince v populaci je potřeba jej pro zjištění fitness naučit na trénovacích datech.

- **Změna evolučního algoritmu nebo jeho parametrů** - (např. velikost populace, typ křížení, pravděpodobnost mutace, typ kódování sítě, změna ukončující podmínky..) - tento přístup napomáhá pouze k lepšímu, případně rychlejšímu nalezení sítě s nejlepšími vlastnostmi a typicky by se pojil s jednou z výše zmíněných variant. Teoreticky by bylo možné, aby evoluční algoritmus hledal optimální nastavení parametrů jiného evolučního algoritmu, který by teprve optimalizoval naši neuronovou síť. To by ovšem znamenalo rapidní nárůst výpočetní složitosti celého mechanismu. Tento přístup by se spíše hodil jako jednorázová akce k dlouhodobému zjištění optimálních parametrů evolučního algoritmu pro další experimentování s učením sítě.

4.1.2 Proměnlivá topologie

Varianty s proměnlivou topologií neuronové sítě samy o sobě zadanou síť nenatrénují, nicméně velice nám napomohou ke zjištění optimálního rozestavení jednotlivých vrstev a propojení mezi nimi. Z praxe můžeme říci, že správné navrhnutí topologie sítě je velice náročná a neméně důležitá úloha a dosud neexistují žádné stoprocentní postupy, jak by toto mělo být správně provedeno. Většina doporučení se odvíjí od předchozích zkušeností, kdy nezbývalo nic jiného, než manuálně zkoušet různá nastavení a změny topologií sítí. Proto tento přístup, kdy evoluční algoritmus navrhne i samotnou strukturu sítě, je mnohem zajímavější než pouhé jeho užití k natrénování sítě s fixní topologií. V tomto ohledu se v posledních letech uchytil algoritmus *NEAT*, který velmi zdatně zvládá tuto úlohu, avšak za cenu velmi vysoké výpočetní náročnosti.

Varianty úprav topologie neuronové sítě můžeme rozřadit do těchto kategorií:

- **Přidávání nebo odebírání neuronů** - změna topologie sítě, například u algoritmu NEAT, který začíná s nejjednodušší topologií a postupně ji zvyšuje nebo později naopak zmenšuje, podle úspěšnosti sítě

- **Přidávání nebo odebírání celých vrstev neuronů** - to je například vhodné při evoluci konvolučních sítí, kdy celé vrstvy představují neurony se stejnými vlastnostmi. Například vrstvy s konvolučními filtry nebo naopak pooling vrstvy pro snížení dimenze vstupních dat.

- **Přidávání nebo odebírání spojení mezi neurony** - tvorba řídkých sítí, opět je tato varianta k vidění u algoritmu NEAT, kdy se vazba mezi dvěma neurony může deaktivovat, avšak stále v síti zůstává pro pozdější možné znovuobnovení

4.2 Přímé kódování

Všechny výše zmíněné postupy evoluce neuronové sítě vycházejí z přímého kódování. Přímé kódování tedy znamená, že reprezentace genotypu odpovídá mapování 1:1 s fenotypem. To znamená, že každá část genomu odpovídá určité části neuronové sítě. Velkou výhodou takové reprezentace je to, že můžeme snadno vidět, jak je síť z takové reprezentace genotypu sestavena. Jednou z nevýhod takového kódování je ovšem to, že při velkých sítích roste stejnou měrou velikost reprezentace genotypu, což s sebou přináší nepříjemný fakt v podobě vysoké výpočetní náročnosti v průběhu evoluce. Opačným přístupem je nepřímé kódování, kam mezi nejznámější zástupce řadíme například algoritmus *HyperNEAT* [15].

4.2.1 CNE

Většina metod využívající přímého kódování využívá fixní topologie sítě, a tak jedinou úlohou evolučního algoritmu je nalezení optimální hodnoty vah mezi jednotlivými neurony. Tyto přístupy byly v počátcích neuroevoluce v polovině 90.let, kdy síť byla reprezentována vektorem vah s fixní velikostí. Tato skupina neuroevolučních algoritmů se obvykle nazývá CNE (*Conventional Neuroevolution*), jelikož stála u zrodu této oblasti.

4.2.2 CMA-ES

Jedním z dalších pokusů o vylepšení takového přístupu byl v roce 1996 algoritmus CMA-ES (*Covariance Matrix Adaptation - Evolution Strategies*). Tato metoda využívala vektorů reálných čísel představujících váhy a především se osvědčila při užití malých populací. Křížení probíhalo na základě evolučních strategií, tedy bylo využito více rodičů pro vznik jednoho potomka a následná mutace využívala gaussovského rozložení pravděpodobnosti.

4.2.3 CoSyNE

Dalším zajímavým přístupem byla metoda CoSyNE. Její princip spočíval v tom, že pro každou jednotlivou váhu v síti byla vytvořena samostatná sub-populace a permutacemi těchto sub-populací se vytvářely a hodnotily jednotlivé sítě. Touto cestou bylo zajištěno, že celková populace sítě byla velmi různorodá.

4.2.4 SANE

Při pokusu o vylepšení různorodosti populací byla v roce 1999 představena metoda SANE (*Symbiotic, Adaptive NeuroEvolution*). Namísto využívání reprezentace jedince v podobě vektoru vah celé sítě, byl jako jedinec v populaci vybrán jediný neuron se svými vahami. Tato metoda je použitelná pouze v dopředné síti s fixní topologií a jedinou skrytou vrstvou. Každý jedinec v populaci tedy představuje neuron ze skryté vrstvy, tj. vektor vah, který obsahuje propojení se všemi neurony vrstvy vstupní a výstupní.

Cílem této techniky je vytvořit různé druhy neuronů v populaci. V optimální neuronové síti každý neuron zastupuje určitou část řešení problému, tj. je specializován na jednu

určitou činnost a právě toto bylo cílem této metody. Ohodnocení jedinců v populaci probíhalo tím způsobem, že bylo náhodně vybráno množství neuronů odpovídající velikosti sítě a takové síti pak bylo změřeno její fitness. Každý neuron si pak takové fitness ukládal. Po několika iteracích s různými sadami neuronů, kdy každý z neuronů měl dostatečné množství nasbíraných fitness hodnot z předešlých působení v různých sítích, bylo fitness každého neuronu vypočítáno jako průměrná hodnota těchto nasbíraných fitness. Tato metoda zajišťuje, že v určitém okamžiku se neurony začnou specializovat na určitý typ úlohy. Následně z takto ohodnocených neuronů bylo vybráno 25% nejlepších a nad těmi bylo uskutečněno křížení. Užívá jednobodové křížení na vektoru reálných čísel obsahující váhy patřící k danému neuronu a tím vznikají dva noví potomci ze dvou rodičů. Mutace pak měnila jednotlivé váhy při 1% pravděpodobnosti změny.

4.2.5 ESP

Vylepšením tohoto algoritmu byla metoda ESP (*Enforced Sub-Populations*), která pouze přidala k tomuto algoritmu sub-populace pro každý jednotlivý neuron. Asi největší výhodou tohoto vylepšení bylo to, že neurony se stejnou specializací byly kříženy pouze mezi sebou, a tak dosahovaly lepších parametrů. Další výhodou tohoto přístupu oproti konvenční neuroevoluci bylo, že sítě se špatným nebo nevyužitým neuronem mohly mít pořád vysoké celkové fitness, zatímco u SANE a ESP by nejspíše k vytvoření nepoužitelného nebo nevyužitého neuronu nikdy nedošlo. Je to vzhledem k tomu, že každý neuron je vyhodnocován zvlášť a jeho fitness je sestaveno z jeho průměrné výkonnosti v rámci několika sítí. Tímto se tedy celá neuronová síť využije s téměř žádnými nadbytečnými neurony.

4.2.6 GNARL

Posledním ze zajímavých přístupů v přímém kódování byla myšlenka zaměřit se na samotnou topologii neuronových sítí. V roce 2002 přišel s touto myšlenkou K.Stanley, který jako první tvrdil, že samotná topologie také výrazně ovlivňuje odezvu sítě a tudíž by i ona měla být součástí evolučních přístupů v neuronových sítích.[30]

Jedním ze zajímavých přístupů bylo propojení evolučního programování s výstavbou topologie neuronové sítě. Evoluční programování je zajímavé tím, že se zde neobjevuje žádný výběr rodičů a tvorba potomků je vázána čistě na proces mutace. Tato metoda dostala název GNARL (*GeNeralized Acquisition of Recurrent Links*).

Prohledávaný prostor v tomto algoritmu je poměrně velký, zejména z toho důvodu, že pro tvorbu sítí byly stanoveny pouze tři omezující podmínky.

- Síť nesmí obsahovat žádná spojení směrem do vstupní vrstvy
- Síť nesmí obsahovat žádná spojení směrem z výstupní vrstvy
- Síť může obsahovat nanejvýš jedno spojení mezi dvěma stejnými neurony

Tato omezení měla za příčinu tvoření poměrně velkého množství neužitečných neuronů, kdy například neurony ve skryté vrstvě mohly existovat bez jakéhokoliv propojení s okolním světem. Váhy mezi jednotlivými neurony byly reprezentovány jako reálná čísla a jejich mutace probíhala na principu gaussovského šumu. Mutace v rámci topologie probíhala na principu přidávání jednotlivých neuronů nebo spojení mezi neurony. Pro zaručení minimální změny hodnoty fitness mezi rodiči a potomky byly nově přidaná spojení ohodnoceny nulovou

váhou. Dalším typem mutace bylo odebírání neuronů, kde ovšem nešlo rozumným způsobem zaručit minimální změnu takovýchto úprav topologie mezi rodičem a potomkem.

Další zajímavostí tohoto algoritmu bylo, že dokázal automaticky zajistit míru mutace T_i jednotlivých potomků. Tato míra mutace byla vypočítána jako:

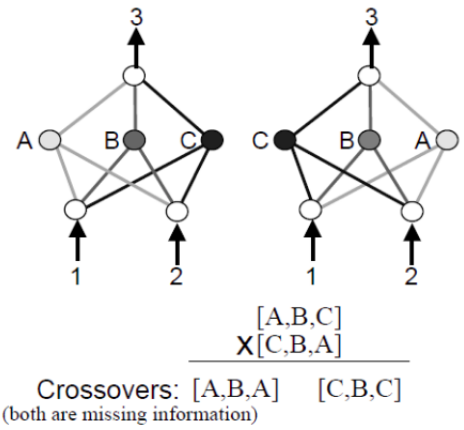
$$T_i = \frac{fitness(i)}{fitness(max)}$$

Tímto bylo zaručeno, že na slabší jedince byla aplikována mutace s větší mírou než na ty kvalitnější a naopak u těch kvalitních mutace prakticky neprobíhala. Tento princip napomáhal celkové konvergenci algoritmu k nalezení jedince s optimální hodnotou fitness.

4.2.7 NEAT

Nejpoužívanějším a také nejzajímavějším algoritmem pro evoluční návrh topologie neuronové sítě dnešní doby je bezesporu algoritmus NEAT (*NeuroEvolution of Augmenting Topologies*). Byl představen roku 2002 K. Stanleym a byl vyvinut s cílem zmírnit poměrně velkou slabinu všech topologicky orientovaných neuroevolučních přístupů. Tímto problémem je *Competing Conventions* [29].

Tento problém znamená, že velké množství různých genotypů představuje ten stejný fenotyp, tj. reprezentují stejnou síť nebo stejnou část sítě. Tento problém má negativní dopad na většinu algoritmů, jelikož například dva rodiče s dobrým fitness, kteří mají náhodou stejný fenotyp, po zkřížení velice často vytvoří dva nepoužitelné potomky s chybějícími informacemi a s velmi degradovaným fitness. Blíže nám tento problém osvětlí následující obrázek 4.1.



Obrázek 4.1: Competing Conventions problém - dvě sítě počítající stejnou funkci pouze s tím rozdílem, že jejich skryté neurony jsou permutovány a tak z pohledu genotypu představují dvě různá řešení. Lze si všimnout, že jednobodové křížení v tomto případě vytváří dvě nové rekombinace, které obě neobsahují jednu ze 3 hlavních složek řešení. Tj. neuron C a A. Obrázek ukazuje pouze 2 ze 6 možných permutací skrytých neuronů.

V algoritmu NEAT jsou proto představeny tři nové inovace zabraňující problému *Competing conventions* se kterým se běžné TWEANN (*Topology and Weight Evolving Artificial Neural Network*) algoritmy setkávají:

- **Dědění ID** - každý gen v sobě mimo informace o neuronu, který reprezentuje obsahuje také unikátní ID. Toto ID je genu přiděleno při jeho vytvoření a po celou dobu běhu algoritmu se nemění. Každý gen má tedy unikátní ID, které se již dále u jiných genů neobjevuje. Toto řešení napomáhá k řešení problému competing conventions, kdy se před křížením jedinců seřadí jejich geny do jednorozměrného pole, kdy dva stejné geny, tj. geny se stejným ID leží na pozici sobě si odpovídající. Tímto docílíme vlastnosti, že již nemůže být ztracen u zkříženého jedince důležitý gen obsahující důležitou část řešení, tj. genu od svých rodičů. Nedochází již tak k degradaci jedince a taktéž nemůže obsahovat dva stejné geny.
- **Nové druhy** - v případě, že již byla nalezena optimální topologie sítě, existuje riziko, že by taková síť mohla projít dalším křížením a byla tak její topologie změněna a degradována. Proto NEAT využívá tzv. vytváření druhů, kdy např. síť s optimální topologií je vyjmuta z druhů, kde se ještě tyto topologie utvářejí a mění a je nastavena na druh, kde již zůstává fixní a pouze se optimalizují její váhy mezi jednotlivými neurony.
- **Minimalizace topologie** - aby se zabránilo vytváření obrovských topologií sítí, nejprve začíná NEAT na nejmenších možných sítích tj. na perceptronech a pouze v případě jednoznačné potřeby teprve přidává nový neuron.

Algoritmus NEAT je sice poměrně výkonný v nalézání optimální topologie sítě a optimálních hodnot vah, avšak za cenu velmi vysoké výpočetní náročnosti. V průběhu algoritmu mohou vznikat i příliš rozsáhlé sítě, které svým natrénováním velmi vytěžují stroj na kterém právě běží a přitom kvalita jejich výsledků nemusí být valná. Přesto se jedná o dosud nejambicioznější a nejzajímavější přístup k evoluční optimalizaci neuronových sítí.

Kapitola 5

Evoluční optimalizace neuronové sítě pro řešení klasifikačních úloh

V této kapitole si rozebereme samotný vlastní návrh na optimalizaci procesu učení klasické dopředné neuronové sítě, který je mimo jiné i tématem této práce. Byly vybrány tři evoluční přístupy, které budou postupně představeny a následně mezi sebou porovnány z hlediska výpočetní náročnosti, samotné odezvy sítě a také doby potřebné k dosažení předem určené přesnosti. Následně budou všechny tyto přístupy porovnány také s klasickým učícím algoritmem pro dopředné neuronové sítě, Back-propagation.

5.1 Klasifikační problém

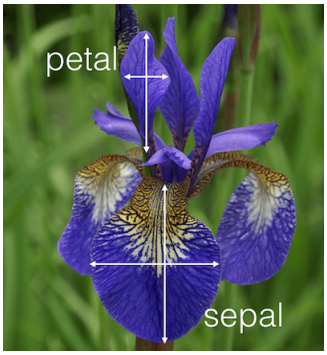
Pro všechny experimenty spojené s evoluční optimalizací neuronové sítě byly zvoleny tři datasety. Všechny spadají do kategorie klasifikačních problémů, kdy každý vzorek z datasetu odpovídá jedné z N klasifikačních tříd. Také všechny atributy každého prvku jsou číselné hodnoty, reprezentované reálnými hodnotami. V těchto klasifikačních problémech se tak nenachází žádné řetězcové literály, výjimkou byly pouze u některých datasetů klasifikační třídy, které ovšem byly převedeny na pravdivostní hodnoty v binární podobě.

Tyto datasety si po řadě od nejtriviálnějšího k nejsložitějšímu postupně představíme a následně si popíšeme jejich úpravu pro další použití během experimentů.

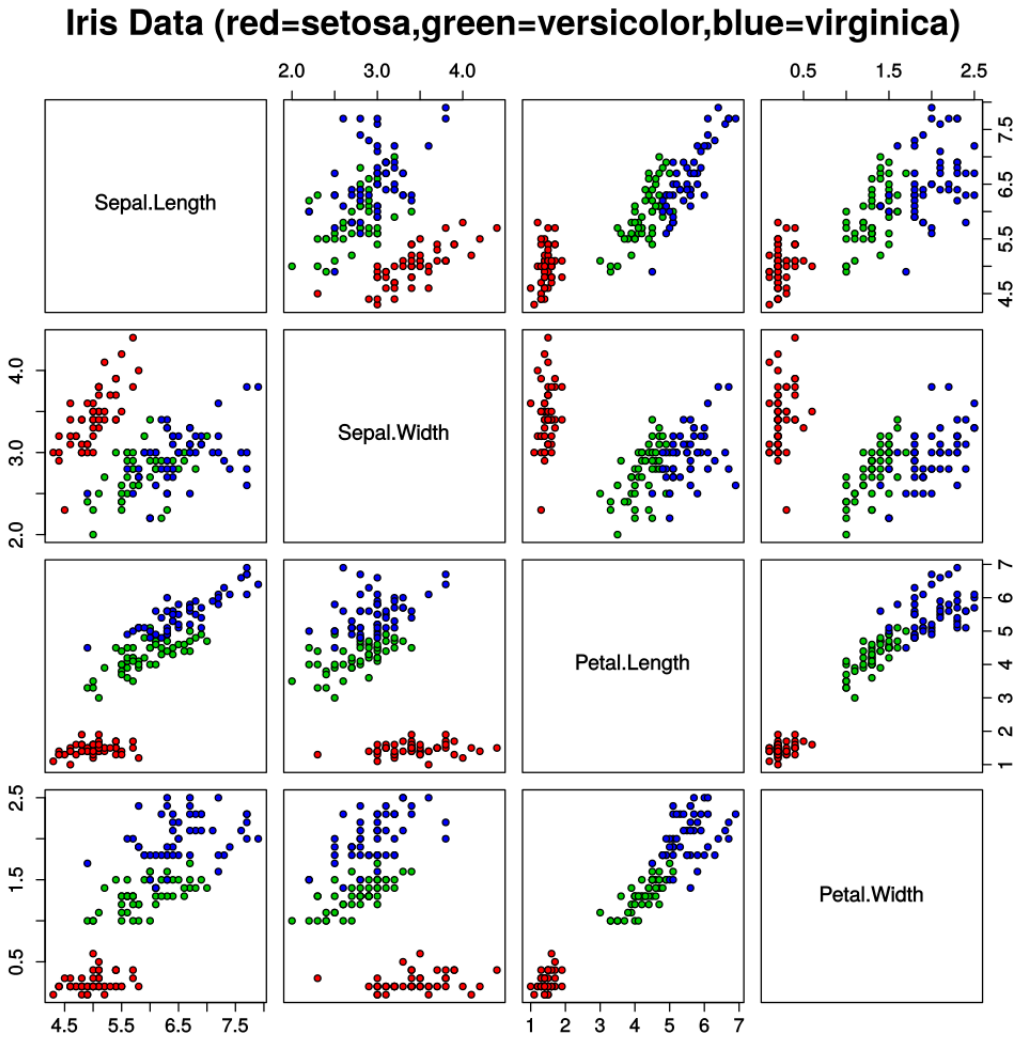
5.1.1 Dataset Iris

Jako první dataset byl použit jeden z nejznámějších datasetů využívaných pro klasifikaci do tříd a to dataset Iris. Jedná se o kolekci 150 záznamů rostliny *Iris* (kosatec). Konkrétněji o rozměry jejích okvětních lístků. Každý vzorek tak obsahuje 4 délkové údaje (šířka, délka) pro 2 typy okvětních lístků pro jednu rostlinu. Klasifikačních tříd obsahuje dataset tři (*Iris Virginica, Iris Setosa, Iris Versicolor*). Dataset je nelineárně separovatelný, avšak stále poměrně jednoduchý. Proto se skvěle hodí na testování funkčnosti a kvality našeho návrhu evoluční optimalizace neuronové sítě.

Petal označuje okvětní lístky, které náleží samotnému květu. *Sepal* označuje okvětní lístky, které bývají u klasických rostlin obvykle zelené a ohraničují květ z jeho vnějšku [3].



Obrázek 5.1: Květ rostliny Iris - znázorňující, které okvětní lístky jsou septal a které petal.



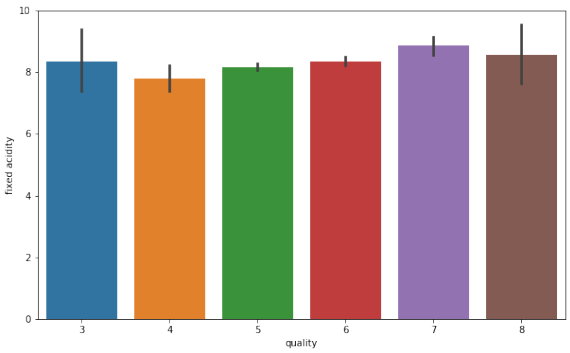
Obrázek 5.2: Rozložení všech 4 parametrů délek a šířek obou typů okvětních lístků (petal, sepal) z datasetu Iris. Vždy vůči ostatním třem parametrům. [Zdroj: http://bit.ly/2LhXZjz]

5.1.2 Dataset kvality červeného vína

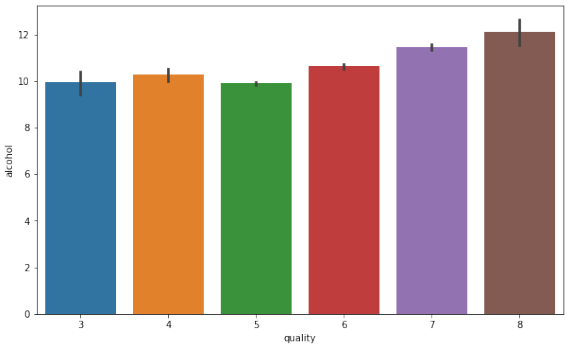
Jako druhý dataset byl vybrán dataset obsahující informace o portugalském červeném víně *"Vinho Verde"* [4]. Obsahuje informace jako hodnotu PH, kyselost, zbytkový cukr a nebo například procenta alkoholu. Celkem se jedná o 11 parametrů. Jako dvanáctý údaj je hodnota značící kvalitu daného vína, která v původním datasetu nabývá hodnot $(0-10)$. Kde hodnota 0 označuje nejhorší a hodnota 10 nejvyšší kvalitu vína. Pro naše účely byly hodnoty parametru určujícího kvalitu zjednodušeny a obsahuje tak nově pouze 3 třídy. Konkrétně špatná (0-3), střední (4-6) a dobrá (7-10) kvalita.

Celkový počet vzorků v datasetu je roven 1599. Tento dataset je náročnější oproti datasetu Iris, jak z hlediska počtu parametrů a jejich hodnotového rozložení, tak z celkového počtu vzorků a můžeme tedy na jeho základě objektivněji zhodnotit sílu daných optimalizačních variant.

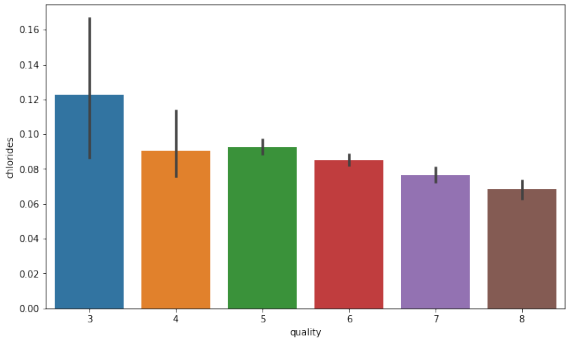
Grafy níže nám ukazují rozložení jednotlivých vzorků datasetu u daných atributů v závislosti na kvalitě vína [1].



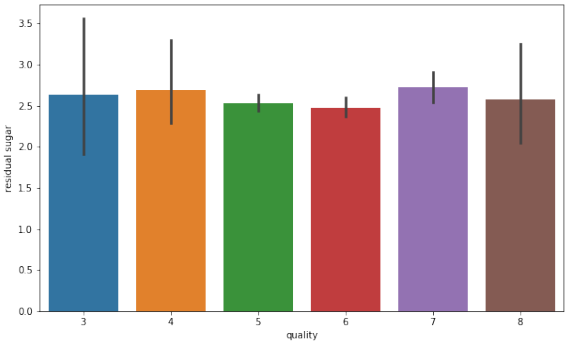
Obrázek 5.3: Kyselost



Obrázek 5.4: Alkohol



Obrázek 5.5: Chloridy



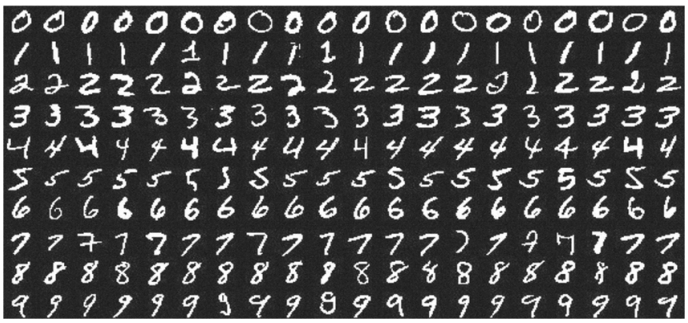
Obrázek 5.6: Cukr

5.1.3 Dataset MNIST

Jako poslední experimentální dataset byl použit dataset obsahující soubor ručně psaných číslic, dataset MNIS (*Modified National Institute of Standards and Technology database*). Jedná se o nejznámější dataset užívaný jako benchmark pro určení kvality neuronových sítí

sloužících pro detekci nebo rozpoznávání objektů v obrazu. Na internetu jsou tedy v hojné míře dostupné výsledky přesnosti klasifikace různých sítí řešících tento problém. Trénovací dataset obsahuje 60 000 ručně psaných číslic, kdy jsou rovnoměrně zastoupeny všechny arabské číslice (0-9) a 10 000 číslic pro validační set.

Každá jednotlivá číslice je reprezentována maticí o velikosti 28x28 pixelů hodnotami RGB (0 - 255) představující různé odstíny šedi [23], [2].



Obrázek 5.7: Grafická ukázka datasetu ručně psaných číslic MNIST [Převzato z: http://bit.ly/2GRjROY]

Z důvodu nízkého výpočetního výkonu a ušetření času byly vyextrahovány z validačního setu pouze tři číslice a to 1, 2 a 3, na kterých byly následně provedeny experimenty. Dataset tedy nově obsahuje pouze tři klasifikační třídy a zhruba 3200 vzorků daných číslic.

5.1.4 Příprava datasetů

Všechny datasety bylo potřeba připravit před jejich následným použitím pro natrénování a vyhodnocení odezvy sítě. Nejprve byly všechny atributy normalizovány podle nejvyšší nalezené hodnoty. Dále bylo potřeba upravit reprezentaci jejich klasifikační třídy. Dekadické hodnoty byly převedeny do jejich binární reprezentace, kdy na hodnotě indexu označujícího danou kategorii byla vložena jednička, na ostatní pak připadla nula.

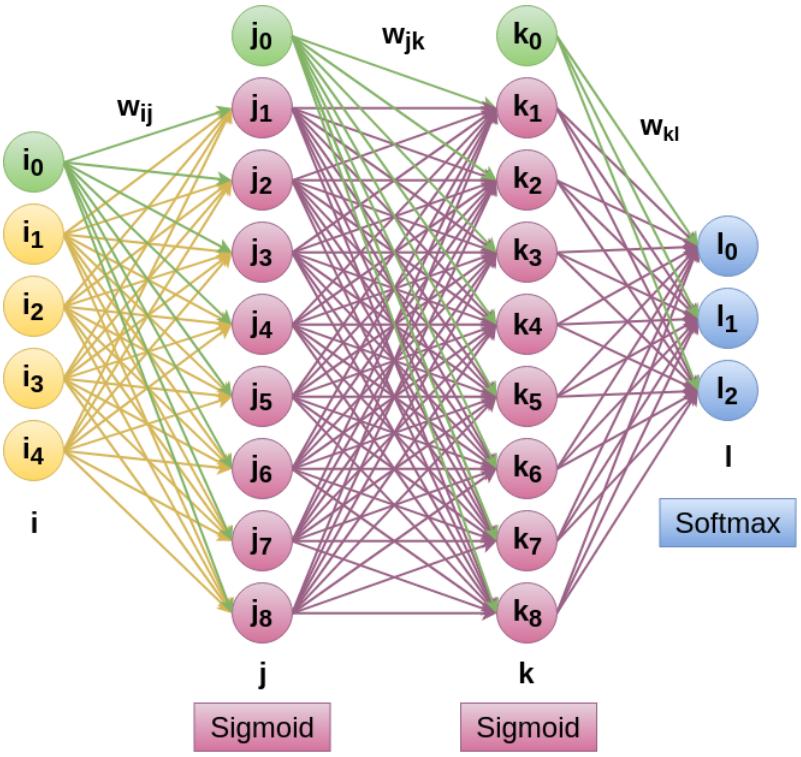
U datasetu MNIST bylo ještě potřeba převést maticovou reprezentaci každé číslice do podoby vektorové. Nově tedy je každá číslice reprezentována vektorem o velikosti 784 pro jednotlivé pixely, rozšířený o klasifikační třídy čítající další 3 hodnoty ke každému vektoru číslice.

Dále byl každý dataset náhodně zamíchán a z tohoto nového uspořádání byl rozdělen na dva celky. První reprezentující trénovací sadu pro samotné trénování sítě a druhý na validační nebo testovací sadu pro otestování kvality odezvy sítě. Toto bylo provedeno v poměru (7:3), kdy pro trénovací sadu bylo použito 70% z celého datasetu a pro validační sadu zbylých 30% vzorků.

5.2 Popis architektury

Pro optimalizaci byl vybrán přístup s fixní topologií neuronové sítě, kdy budeme evolučně upravovat hodnoty jejich vah a biasů. Ostatní parametry neuronové sítě zůstanou fixní. Tedy přístup velice podobný s metodou CNE viz. 4.2.1 nebo CMA-ES viz 4.2.2. Jako aktivační funkce ve skrytých vrstvách byla vybrána funkce *Sigmoid*. Jako aktivační funkce v poslední vrstvě byla zvolena funkce *Softmax* spolu se ztrátovou funkcí *Cross-entropy*. Ta je výhodná

především z toho důvodu, že naše síť bude řešit klasifikační problémy, a tak je vhodné, aby výstupní hodnoty v poslední vrstvě neuronové sítě dávaly v součtu jedničku a tudíž sloužily jako pravděpodobnostní vyjádření náležitosti daného vzorku do daných klasifikačních tříd.



Obrázek 5.8: Topologie neuronové sítě pro dataset Iris. Neurony i_0, j_0, k_0 označeny zeleně představují biasy zapojující se do aktivace v následující vrstvě. Vrstva i označuje vstupní vrstvu (žlutě), vrstva j,k po řadě označují první a druhou skrytou vrstvu (červeně) a vrstva l označuje výstupní vrstvu (modře). Aktivační funkce pro celou vrstvu jsou popsány v rámečku pod nimi. Dataset Iris má 4 vstupní parametry a 3 klasifikační třídy. Vnitřní vrstvy mají po 8 neuronech.

Topologie naší dopředné, plně propojené neuronové sítě sloužící k řešení klasifikačních úloh byla pro každý dataset zvolena experimentálně a po dobu všech experimentů zůstala neměnná. Platí však, že pro každou klasifikační úlohu naše síť obsahovala 2 skryté plně propojené vrstvy, jednu vstupní a jednu vrstvu výstupní. Ve výstupní vrstvě je pak vždy takový počet neuronů, jaký je počet možných klasifikačních tříd daného datasetu.

Aby bylo zabráněno zkreslení výsledků, byly všechny evoluční algoritmy a také algoritmus *Back-propagation* ručně implementovány. Vymažou se tak rozdíly, které by při použití různých knihoven určených pro strojové učení mohly nastat. Každá využívá jiných možností optimalizací výpočtů, případně využívají pro výpočty hardwaru grafické karty. Vlastní implementací jsme tedy do jisté míry zobjektivnili naše výstupní hodnoty u všech experimentů. Ručně byla implementována také samotná neuronová síť. V této práci bylo pro usnadnění implementace využito pouze knihovny *Numpy*, která usnadňuje a optimalizuje početní operace při vektorových a maticových výpočtech.

Pro všechny optimalizační algoritmy byl použit stejný způsob zakódování jedince, tedy neuronové sítě. Jedná se o kódování založené na reálných hodnotách genotypů. Každá jednotlivá váha sítě je vyjádřena reálnou hodnotou, to stejné platí i pro hodnoty biasů. Váhy

jsme tedy nemuseli nijak speciálně převádět, například do binární podoby, která by nám ulehčila práci při operátorech křížení nebo mutace. Na místo toho bylo zvoleno speciální aritmetické křížení, určené právě pro křížení genotypů složených z reálných hodnot. Jedince tedy představuje soubor všech vah a biasů všech vrstev sítě. Přesněji matice vah pro každou vrstvu. Genomy pak představuje každá jednotlivá váha nebo bias. Bias je modelován jako nultý neuron každé vrstvy s pevně danou a neměnnou hodnotou 1.

5.3 Návrh optimalizátorů - (Varianta 1)

Pro optimalizaci vah a biasů byly zvoleny dva evoluční algoritmy a jeden algoritmus z oblasti inteligence hejna, který nicméně také spadá do algoritmů inspirovaných přírodou.

Cílem je nalézt optimální nastavení parametrů u každého evolučního algoritmu, a tak dosažení nejlepších možných výsledků, kterých je daná metoda schopna. Tento cíl máme z toho důvodu, abychom mohli co nejlépe porovnat maximální možnosti uvedených variant mezi sebou, a tak utvořili co nejobjektivnější shrnutí jejich účinnosti při učení neuronových sítí.

5.3.1 Společné parametry

Všechny algoritmy se kterými budou probíhat experimenty mají určité společné parametry, zejména co se týče určení kvality výsledně optimalizované sítě nebo reprezentace jedince v rámci evolučního algoritmu. Tím docílíme nejobjektivnějších výsledků při jejich testování, a také nám tento přístup usnadní práci při samotné implementaci algoritmů.

Souhrnně budeme při experimentech tento základní přístup nazývat *Varianta 1.*

Jedinec

Jako jedinec bude vždy považována samostatná neuronová síť o stejné topologii a se stejnými aktivačními a ztrátovou funkcí. Rozdíl bude pouze v hodnotách jejich vah a biasů, které budou vždy náhodné, avšak tyto náhodné hodnoty budou vždy ze stejného rozsahu $\langle -1, 1 \rangle$. To platí samozřejmě pouze při inicializaci algoritmu. Zároveň bude jedinec obsahovat hodnotu svého fitness a také údaj o jeho přesnosti (tzn. procentuální vyjádření, kolik vzorků z trénovacího datasetu klasifikoval správě). Například u algoritmu PSO (Optimalizace hejnem částic) bude navíc takový jedinec obsahovat i své dosud nejlepší (co se týče hodnoty jeho fitness funkce) hodnoty vah a také si bude pamatovat jedince s nejlepším fitness v rámci celé populace. Zakódování jedince bude tedy přímé a budou jej představovat tři matice vah a biasů v reálné podobě. Tato reprezentace jedince samozřejmě platí pro evoluční operátory, hodnoty fitness a přesnosti jsou spíše ukazatele, které nás informují blíže o dané síti a do samotného procesu evoluce (křížení, mutace) se nezapojují.

Fitness

Funkce fitness, která slouží pro hodnocení kvality jedinců v rámci populace bude u všech algoritmů představovat hodnota chybové funkce Cross-entropy. Přesněji její suma v rámci celého trénovacího setu, vydělená velikostí tohoto setu. Tedy vlastně aritmetický průměr hodnot chyby z jedné iterace *Feed-forward* algoritmu jedné konkrétní sítě. Vzorec pro hodnotu fitness je vyjádřen následovně:

$$\text{chyba} = \frac{\sum(\text{chyba každého klasifikovaného vzorku})}{\text{počet všech klasifikovaných vzorků}}$$

Chybová funkce

Jako chybová funkce byla zvolena pro všechny typy optimalizace chybová funkce Cross-entropy. Jelikož u každého datasetu máme 2 a více klasifikačních tříd, byla zvolena varianta pro $C > 2$, kde C označuje počet klasifikačních tříd. Cross-entropy je tedy definována pro určení chyby jednoho vzorku z datasetu takto:

$$CE = -\sum_{i}^{C} d_i \log(y_i)$$

Přesnost

Další ukazatel kvality natrénování sítě je bezpochyby přesnost. Ta nám ukazuje kolik vzorků z datasetu bylo klasifikováno do správné třídy ve vztahu s celkovým počtem klasifikovaných vzorků. Tím nám ukazuje s jakou pravděpodobností síť správně klasifikuje libovolný náhodný vzorek. Přesnost se tedy pohybuje v intervalu $\langle 0 - 1 \rangle$. Kdy přesnost $= 1$ označuje 100% pravděpodobnost správné klasifikace vzorku neuronovou sítí. Ukazatel přesnosti se řídí následujícím vzorcem:

$$\text{přesnost} = \frac{\text{počet správně klasifikovaných vzorků}}{\text{počet všech klasifikovaných vzorků}}$$

Ukončující kritérium

Jako ukončující kritérium pro optimalizační algoritmus budeme brát buď dosažení 100% přesnosti u validačního setu nebo dosažení předem zvoleného počtu iterací algoritmu. Bude záležet u každého experimentu, jak budou zvoleny startovací podmínky, například u datasetu MNIST bude jistě hranice nulové chyby nebo 100% přesnosti nedosažitelná, a tak bude nejspíše zvoleno jiné ukončující kritérium.

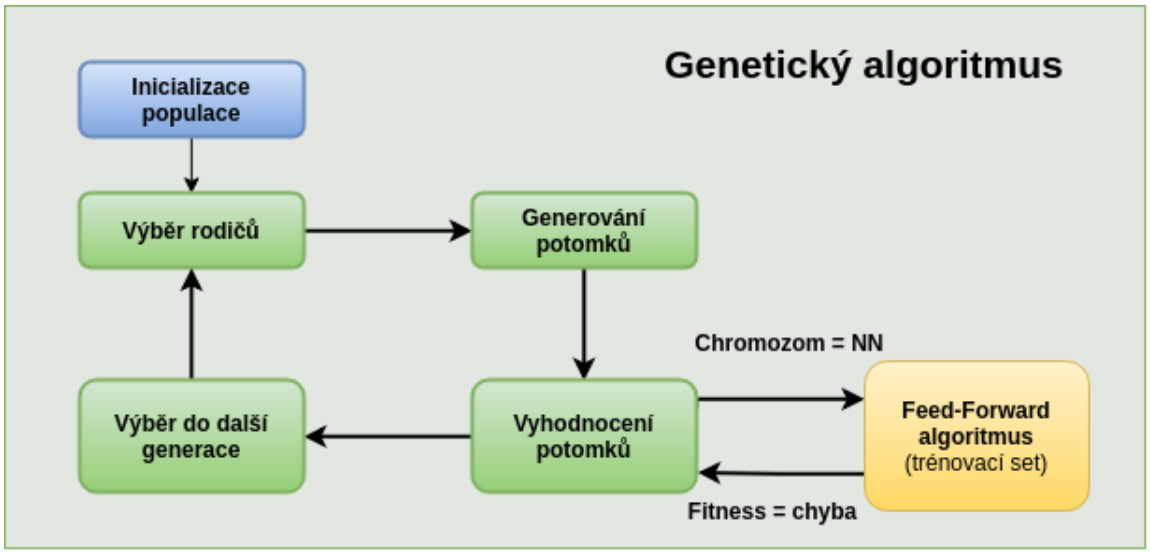
Jelikož se každý optimalizační algoritmus již od základu chová vcelku rozdílně, není vhodné jako ukončovací kritérium volit například rozdíl chyby mezi dvěma iteracemi. U Back-propagation by toto kritérium dávalo smysl, stejně jako i u genetického algoritmu. Naopak u diferenciální evoluce, která je přímo založená na postupném vymaňovaní se z lokálních extrémů nebo u optimalizace hejnem částic, bychom tak ukončili algoritmus příliš brzy a nedosáhli těch nejlepších možných výsledků.

5.3.2 Genetický algoritmus

Jako první evoluční přístup pro optimalizaci vah byl vybrán genetický algoritmus. Tento algoritmus existuje ve všech možných variantách implementace, kdy můžeme měnit princip výběru rodičů, měnit hodnoty určující náhodnost jevů jako mutace a jiné. Dává nám tak volnou ruku v různorodosti jeho variant při dalších experimentech.

Princip je ale takový, že jedinec bude vyhodnocen, tzn. bude mu udělena hodnota fitness na základě Feed-forward algoritmu, kdy se na trénovací sadě určí jeho průměrná chyba. Poté nejlepšímu jedinci v rámci populace bude umožněno provést druhou iteraci algoritmu

Feed-forward na validační sadě, a tak bude zjištěna kvalita natrénování aktuální populace. Nejlepší jedinec pak představuje kvalitu celé generace.



Obrázek 5.9: V tomto obrázku můžeme blíže vidět celkový princip optimalizace pomocí genetického algoritmu.

Nyní si popíšeme v jakých konfiguracích se bude algoritmus účastnit experimentů.

Selekce

Operátor selekce se řídí na základě turnajového výběru, který se prokázal být nejvýhodnějším řešením. Navíc se jednoduše, pouze pomocí úpravy velikosti takového turnaje, tedy počtu jedinců z populace, kteří se zúčastní jednoho turnaje, koriguje selekční tlak. Tím můžeme jednoduchým způsobem měnit strategii výběru. Základní varianty budou počítat s nízkou velikostí turnaje (kolem 5 jedinců) a celkem bude probíhat vždy sudý počet takových turnajů, abychom dosáhli sudého počtu takto vybraných jedinců, kvůli pozdějšímu operátoru křížení. Stejní jedinci se budou moci turnaje účastnit opakovaně, jelikož budou vybírání vždy náhodně. Nicméně nikdy nebudou dva stejní jedinci účastníci stejného turnaje.

Křížení

Křížení probíhá vždy mezi dvojicí rodičů, kteří byli vybráni na základě zvolené selekční heuristiky, typicky Turnajovým výběrem. Pro operátor křížení bylo na výběr zvolit ze dvou různých přístupů. Těmito přístupy jsou aritmetické a heuristické křížení. Bylo vybráno aritmetické křížení, kdy pro každou vrstvu vah sítě byla náhodně zvolena hodnota q, která určuje podíl jednotlivých složek vah z obou jedinců. Řídí se tímto vzorcem:

$$weights_i^{ch1}[x][y] = weights_i^{p1}[x][y] * q + weights_i^{p2}[x][y] * (1 - q)$$

$$weights_i^{ch2}[x][y] = weights_i^{p1}[x][y] * (1 - q) + weights_i^{p2}[x][y] * q$$

Kde i je vrstva vah sítě, ch je potomek, $p1$ a $p2$ rodiči účastnící se křížení a x, y označující indexy vah.

Samotné křížení pak probíhá u dvou korespondujících vah stejné vrstvy dvou rodičů a takto proběhne postupně u všech vah a biasů. Takto vzniklí jedinci jsou poté zmutováni.

Mutace

Operátor mutace probíhá na principu, že s pravděpodobností p je u každého jedince zmutována jedna váha každé vrstvy a to o náhodnou hodnotu řídící se tímto vzorcem:

$$mutace = \frac{max(weights_i[x][y])}{10}$$

Nalezne se nejvyšší hodnota váhy dané vrstvy a z této hodnoty se její 10% velikost přičte nebo odečte k náhodně vybrané váze.

Výběr do další generace

Výběr do další generace se řídí dvěma přístupy. Prvním je poloviční generační obměna, tedy nově vzniklí jedinci operátorem křížení představují polovinu nové populace a druhou polovinu představují jejich rodiče, tedy ti jedinci vybráni v předchozí selekci turnajovým výběrem.

Druhý přístup je celková generační obměna, tedy je vybrán turnajovým výběrem počet tvořící celou novou generaci, ta je posléze zkřížena a zmutována a tento celek nových jedinců bude představovat novou generaci.

Postupně budou experimenty prováděny s oběma variantami a následně upřednostněna výkonnější z nich.

5.3.3 Diferenciální evoluce

Optimalizace vah diferenciální evolucí je druhou evoluční metodou se kterou budeme provádět experimenty. Zakódování chromozomu bude opět stejné jako u genetického algoritmu a princip zisku hodnoty fitness také. Budeme tedy sledovat fitness na základě algoritmu Feed-forward při trénovacím setu a nejlepšího jedince v rámci každé generace také otestujeme na validačním setu. Hodnoty vah a biasů budou opět inicializovány náhodně v rozsahu $\langle -1, 1 \rangle$.

Křížení

Jelikož je diferenciální evoluce přímo určená pro genotypy složené z reálných hodnot, nemusíme tak využívat žádného speciálního typu křížení a využijeme standardní varianty křížení pro tento algoritmus. Konkrétně se bude jednat o 4 varianty, podle kterých se i tyto varianty jmenují. Opět spolu budeme křížit navzájem korespondující váhy jednotlivých vrstev, lišit se tak budou tyto varianty pouze v počtu jedinců zapojujících se do operátoru křížení a výběr kandidátního jedince.

Varianta rand/2/bin

U této varianty je kandidátní jedinec, ke kterému se budou dané diference vah přičítat, zvolen náhodně v rámci celé populace. Následně jsou také zvoleni 4 další jedinci. Pro tyto jedince platí, že musejí být rozdílní jak mezi sebou, tak i s kandidátním jedincem. Poté se vypočítají jejich diference mezi korespondujícími vahami a následně vynásobené mutační konstantou se přičtou k tomuto kandidátnímu jedinci. Jelikož je užito v této variantě binomické rozložení pravděpodobnosti při křížení, je každá váha z každé vrstvy nezávisle na ostatních vahách s pravděpodobností CR (*crossing-point*) vyměněna na korespondující index stejné vrstvy do aktuálního jedince. Poté je nově vzniklý jedinec ohodnocen funkcí fitness opět pomocí jedné iterace algoritmu Feed-forward nad trénovacím setem a jeho fitness porovnáno s aktuálním jedincem. Pokud je lepší, nahrazuje tento nově vzniklý jedinec v populaci toho aktuálního. Takto proběhne křížení pro celou populaci v rámci jedné iterace algoritmu.

$$w_i^{mut}[x][y] = w_i^a[x][y] + MR * (w_i^b[x][y] - w_i^c[x][y] + w_i^d[x][y] - w_i^e[x][y])$$

Kde w_i je vrstva vah i, *mut* je mutační jedinec, MR je *mutation rate* - mutační faktor, *a, b, c, d, e* jsou jedinci náhodně vybráni pro křížení a jedinec a je kandidátní jedinec a x, y jsou indexy jednotlivých vah.

Varianta best/2/bin

U varianty best/2/bin je kandidátní jedinec, ke kterému se budou diference vah přičítat, zvolen jako nejlepší jedinec v dané generaci. Nejlepšího jedince vybíráme na základě jeho hodnoty fitness. Jinak je tato varianta prakticky totožná s variantou *rand/2/bin*, nicméně při běhu se obě varianty chovají rozdílněji, než by se na první pohled mohlo zdát.

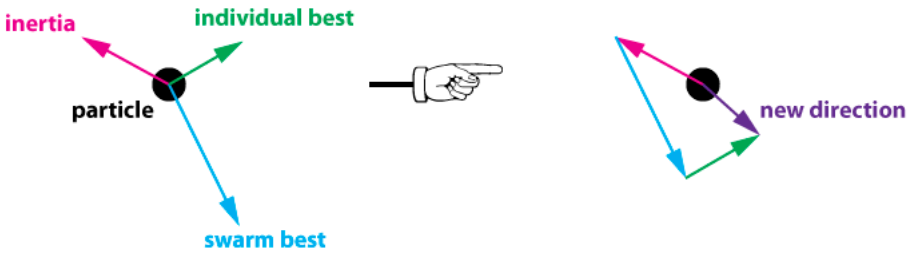
5.3.4 Optimalizace hejnem částic

Všechny podmínky ohledně neuronové sítě budou i u optimalizace hejnem částic stejné jako u ostatních dvou algoritmů. Tedy fitness funkce bude určovat průměrnou hodnotu chyby na základě algoritmu Feed-forward nad trénovacím setem, opět nejlepší řešení v rámci populace bude otestováno na validačním setu. Váhy budou inicializovány opět náhodně ve stejném rozsahu $\langle -1, 1 \rangle$. Křížení ani mutace při tomto algoritmu nejsou potřeba, bude se pouze vypočítávat směrový vektor vel_i, na základě kterého jedinec mění svou pozici ve stavovém prostoru řešení.

$$vel_{i+1} = (w * vel_i) + (c_1 * rand_1) * (best_i - indiv_i) + (c_2 * rand_2) * (global_i - indiv_i)$$

Kde parametr w určující sílu vlivu směrového vektoru vel_i k budoucí nové pozici bude při experimentech v rozsahu $(0, 1)$, parametry c_1, c_2 určující sílu vlivu historicky nejlepší pozice samotné částice ($best_i$) a druhý sílu vlivu pozice nejlepší částice v rámci celého hejna ($global_i$) budou v rozsahu (0,2). $indiv_i$ je aktuální pozice částice a čísla $rand_1, rand_2$ jsou dvě náhodné reálné hodnoty v rozsahu $(0, 1)$. Proměnná i označuje index aktuální částice a nabývá hodnoty až do N, tedy velikosti populace.

Obrázek níže naznačuje graficky výpočet nové pozice částice při každé iteraci:



Obrázek 5.10: Výpočet nové pozice v n-dimenzionálním stavovém prostoru řešení u Optimalizace hejnem částic. Swarm best - značí nejlepší pozici jedince (nejlepší řešení) v rámci celého hejna, Particle - aktuální částici pro kterou se provádí výpočet nové pozice, *Inertia* - směrový vektor (velocity = inertia) vel_i, Individual best - pozice historicky nejlepšího řešení aktuální částice.

5.4 Modifikovaná metoda - (Varianta 2)

Modifikovaný přístup k optimalizaci vah neuronové sítě bude spočívat v průběhu evaluace a zjišťování hodnoty fitness jednotlivých sítí v rámci populace. Dosud jsme nově vzniklého jedince otestovali funkcí fitness a zjistili jak dobře se umístil v rámci své aktuální populace. Dělali jsme tak tím způsobem, že jsme na optimalizovanou síť pustili algoritmus Feed-forward s trénovací sadou a poté spočítali její průměrnou chybu na jeden vzorek. To tvořilo hodnotu fitness této sítě, která se dále zapojila do průběhu evolučního algoritmu. Nyní tento přístup poněkud změníme a budeme sledovat, zda-li se modifikace promítne do kvality a rychlosti učení a hodnoty fitness budou konvergovat rychleji ke svému minimu, tzn. jednotlivé hodnoty vah a biasů sítě ke svému optimu při zachování maximální přesnosti.

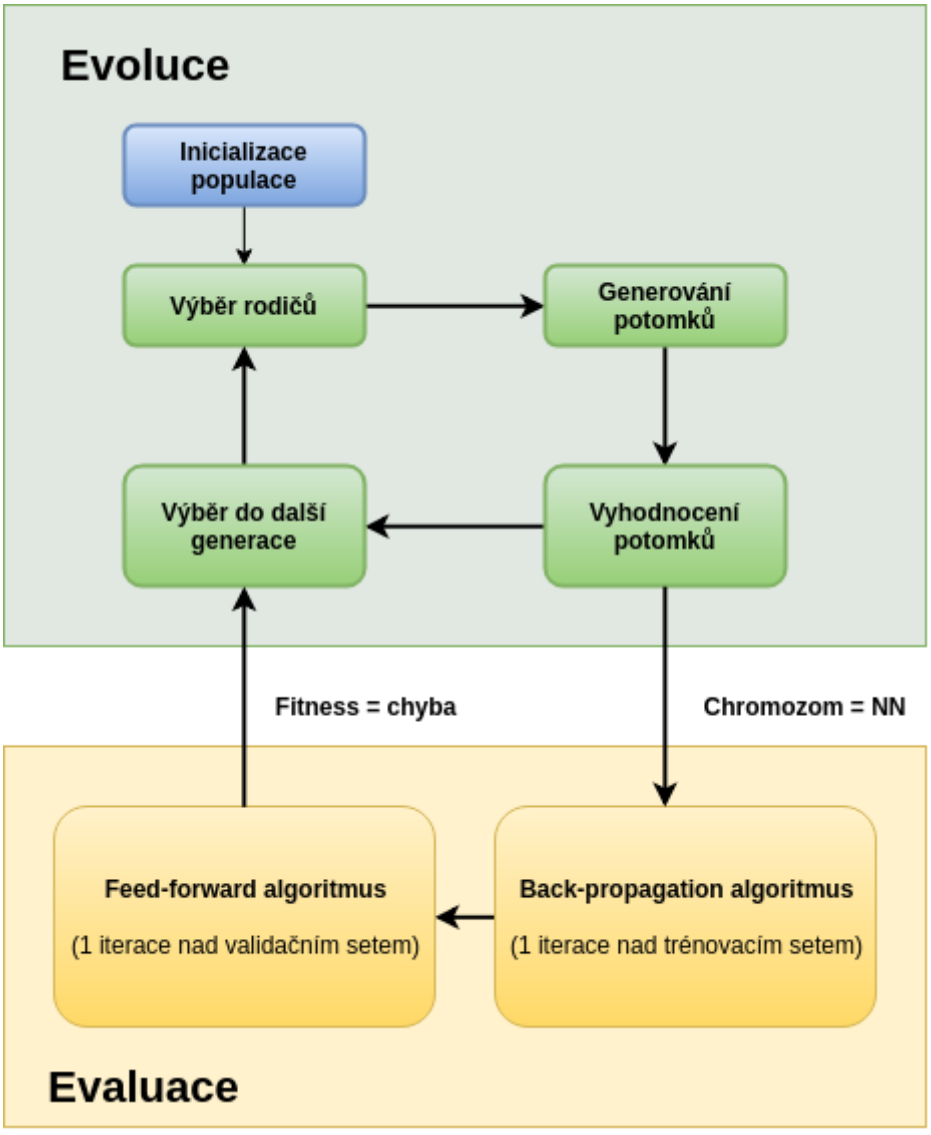
Tento přístup spočívá v tom, že nově předtím, než budeme jedince (síť) evaluovat, tedy pouštět na ní algoritmus Feed-forward, síť jednou iterací naučíme pomocí Back-propagation. Tedy pustíme na ni algoritmus back-propagation s trénovacím setem. Tím změníme jednotlivé hodnoty vah, které nám budou konvergovat správným směrem ke své optimální hodnotě a teprve poté na takto modifikovanou síť pustíme Feed-forward algoritmus, ale tentokrát již s validačním setem. Takto získaná průměrná hodnota chyby z validačního setu bude nově představovat naši hodnotu z funkce fitness a na jejímž základě bude síť poměřována s ostatními jedinci v rámci své aktuální populace. Abychom dosáhli výraznější změny vah a tedy, aby i algoritmus back-propagation mělo smysl vůbec zahrnovat, bude zvýšena hodnota míry učení (*learning-rate*), oproti hodnotě využívané při experimentech se samotnou back-propagation.

Váhy a biasy sítě se při přechodu algoritmu back-propagation budou aktualizovat po každém vzorku z trénovacího setu. Stejně tak pracuje i samotná implementace back-propagation se kterou budeme srovnávat naše evoluční přístupy.

Předpoklad je, že i jedna iterace algoritmu back-propagation může výrazně napomoci konvergenci hodnot vah blíže svým optimálním hodnotám a zároveň nezabere příliš mnoho

výpočetního času. Také pro samotné optimalizační evoluční algoritmy může tento přístup pomoci podobným způsobem, tak jako v jiných evolučních přístupech spojených s učením neuronových sítí, kdy jsou sítě po inicializaci nejprve lehce natrénovány, aby jejich váhy neměly čistě náhodné hodnoty, a tak byly již od začátku optimalizace mnohem blíže ke svým optimálním hodnotám. My toto předtrénování nebudeme nyní využívat při inicializaci sítí, ale vždy před jejich evaluací. Takový je tedy předpoklad a bude zajímavé sledovat, jestli i v realitě tato "dopomoc"bude opravdu užitečná a povede k celkovému zrychlení nebo zpřesnění našich evolučních optimalizací. Budeme se opět soustředit na přesnost, hodnotu chyby tj. hodnotu fitness, celkový výpočetní čas i počet iterací potřebných k dosažení daných výsledků. Při experimentech bude tato metoda pojmenována jako *Varianta 2*.

Nový přístup blíže vysvětluje následující schéma:

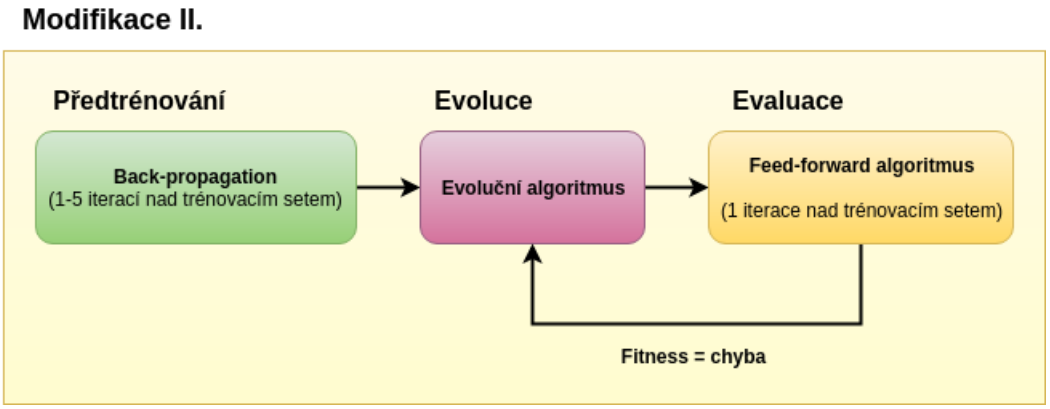


Obrázek 5.11: Modifikovaný přístup demonstrovaný při použití genetického algoritmu

5.5 Druhá modifikace - (Varianta 3)

Pro srovnání na závěr také vyzkoušíme metodu krátkého předtrénování sítě pomocí algoritmu Back-propagation. Jednat se bude o malý počet iterací (1-5) nad trénovací sadou před zahájením samotného evolučního optimalizačního algoritmu. Budeme se pokoušet síť natrénovat jen do takové míry, aby vlastní optimalizace a zdokonalování výsledků probíhalo hlavně díky evoluční optimalizaci. Všechny tyto přístupy budou spolu opět stejným způsobem porovnány.

Tato metoda bude při experimentech označována jako *Varianta 3*.



Obrázek 5.12: Druhá modifikovace - předtrénování sítě pro rychlejší konvergenci hodnot vah sítě na jejich optimální hodnoty

5.6 Implementace

Pro implementaci programové části této práce byl zvolen programovací jazyk *Python* ve verzi 3.6. Jako jediná knihovna třetích stran byla použita knihovna *Numpy*. Tato knihovna slouží k usnadnění práce s vektory a maticemi a operacemi nad nimi. Je implementována v programovacím jazyce C, tudíž výpočty prováděné s touto knihovnou jsou velice rychlé a poměrně dobře optimalizované. Program byl napsán v objektově orientovaném paradigmatu.

Implementační balík se sestává z několika částí, které si ve zkratce popíšeme. Celý balík obsahuje vše potřebné pro kompletní běh programu a spuštění následných experimentů. Hlavní části programu jsou:

- `NN - neural_network` - obsahující implementaci neuronové sítě
- `GA - genetic_algorithm` - obsahující implementaci genetického algoritmu
- `DE - differential_evolution` - obsahující implementaci diferenciální evoluce
- `PSO - particle_swarm_optimization` - obsahující implementaci optimalizace hejnem částic
- `Dataset` - obsahující kód na přípravu datasetů pro použití v programu
- `Graph` - obsahující kód pro tvorbu grafů, samostatně spustitelný

Pro implementaci neuronové sítě, algoritmu back-propagation ani žádné z variant optimalizačních algoritmů nebyly použité žádné specializované knihovny určené pro strojové učení. Všechny zmíněné celky byly implementovány pouze za použití čistého jazyka Python spolu s matematickou knihovnou Numpy. Bylo tak učiněno proto, abychom pochopili celkový princip fungování jednotlivých složek algoritmů a zejména proto, aby se vymazaly případné rozdíly implementací. Jelikož žádná knihovna třetích stran neobsahuje všechny zmíněné algoritmy zároveň dohromady, mohlo by tak vznikat v důsledku různých přístupů k implementaci například k rozdílné rychlosti evaluace jednotlivých částí a tudíž k neobjektivnímu srovnání jednotlivých přístupů při učení neuronových sítí.

5.6.1 Použité funkce a struktury

Pro implementaci byly hojně využity možnosti matematické knihovny Numpy. Jelikož bylo naším cílem optimalizovat hodnoty vah a biasů neuronové sítě, proto byly pro urychlení těchto kroků váhy a biasy implementovány jako datový typ `numpy.array`. Jedná se o matici těchto vah, kdy její řádky odpovídají počtu neuronů vrstvy i a počty sloupců vrstvě neuronů vrstvy nadcházející, tedy vrstvy $i + 1$. Každá váha je reálné číslo datového typu `numpy.float64`. Inicializované na náhodnou hodnotu v rozmezí $\langle -1, 1 \rangle$.

Pro všechny náhodné hodnoty byl použit generátor náhodných čísel při rovnoměrném rozložení pravděpodobnosti `numpy.random.uniform`. Tento generátor využívá algoritmu pro generování náhodných hodnot zvaný *Mersenne Twister*.

Neuronová síť

Samotná neuronová síť je implementována ve třídě `neural_network`, kdy každá jednotlivá síť představuje instanci této třídy. Jako vstupní argumenty konstruktoru této třídy jsou počty neuronů jednotlivých vrstev sítě. Samotný objekt pak obsahuje pět proměnných, tři z nich jsou implementovány jako `numpy.array` obsahující matici vah a biasů jednotlivých vrstev sítě. Zbylé dvě proměnné `fitness, accuracy` ukládají hodnotu fitness funkce a přesnost konkrétní sítě. Proměnná `fitness` je inicializována na hodnotu *infinity*, která je součástí Python knihovny `math`. Proměnná accuracy je inicializována na hodnotu -1.

Další součástí této třídy je funkce `feed_forward()` implementující stejnojmenný algoritmus, přijímající jako vstupní argument objekt neuronové sítě, dataset a pravdivostní hodnotu, zda-li má být využit algoritmus Back-propagation.

Třída také obsahuje samotný algoritmus Back-propagation a také obsahuje všechny implementace aktivačních funkcí a derivací těchto funkcí potřebné pro správný běh Back-propagation, včetně *cross-entropy* a funkce *soft-max*.

Také obsahuje funkci pro zjištění hodnoty fitness `fit_function()` a funkci pro inicializaci populace neuronových sítí `init_population()`.

Dataset

Tento soubor obsahuje funkce potřebné pro přípravu datasetů pro jejich následné použití při prováděných experimentech. Načítají přiložené datasety ve formátu `csv` a hodnoty klasifikační třídy převádí na binární podobu. Jedinou výjimkou je dataset `MNIST`, který je nahrán prostřednictvím knihovny *TensorFlow* [2].

Genetický algoritmus

Hlavní funkcí obsluhující celý algoritmus je funkce `genetic_algorithm()`, přijímající vstupní argumenty jako zvolený dataset, počet iterací algoritmu, velikost populace, parametry neuronové sítě - tedy parametry jedince a zvolenou variantu optimalizace. Z této funkce se následně volají ostatní funkce nezbytné pro běh algoritmu.

Jsou to funkce `selection_tournament()` zajišťující zvolení rodičů v rámci populace na základě turnaje. Následně jsou tito jedinci zkříženi ve funkci `crossover_selected()` a zmutováni ve funkci `mutate()`. Inicializace populace je prostřednictvím již dříve zmíněné funkce `init_population()` nacházející se v souboru `nn.py`.

Diferenciální evoluce

Hlavní funkcí tohoto algoritmu je funkce `differential_equation()`. Přijímající opět stejné parametry jako hlavní funkce u genetického algoritmu, ale navíc doplněná parametrem, který označuje specifickou variantu diferenciální evoluce (*rand1bin, best2bin, atd*). Opět je na počátku vygenerována populace neuronových sítí funkcí `init_population()`. Následně je změřeno fitness opět funkcí `fit_function()` nacházejícím se v souboru `nn.py`. Následně je funkcí `execute_method()` zvolena daná varianta algoritmu a je v ní provedeno křížení. Tyto varianty jsou implementovány ve funkcích: `best1(), best2(), rand1(), rand2()`. Kandidátní jedinec je uložen v proměnné `candidate`. Pro nalezení nejlepšího jedince v rámci populace je užita funkce `get_best()` nebo `get_best_idx()`, podle toho, zda-li chceme ukazatel na jedince nebo pouze jeho index v rámci datové struktury `list` všech jedinců.

Pro tvorbu jedinců s identickými vlastnostmi využíváme vestavěné funkce `deepcopy()` v knihovně `copy`.

Optimalizace hejnem částic

Hlavní funkcí souboru `pso.py` je funkce `particle_sworn_optimization()`. Obsahující celý algoritmus bez potřeby užití jiných funkcí specifických pro tuto metodu.

Při každé iteraci algoritmu je vypočítána nová hodnota směrového vektoru pro každou váhu, uložená v proměnné `new_velXY`. O kterou je následně inkrementována hodnota vah tohoto jedince. Opět se jedná o proměnné datového typu `numpy.array`.

V tomto algoritmu jsou využity i další proměnné nacházející se v objektu neuronové sítě z třídy `neural_network` a to datového typu `numpy.arrays` obsahující dosud nejlepší hodnoty vah `best_weightsXY` dané instance a také hodnoty jejich směrových vektorů `vel_weightsXY`. Dále také korespondující hodnotu fitness v proměnné `best_fitness`. Tyto proměnné jsou inicializovány právě pouze při využití této optimalizační metody. Je to z toho důvodu, aby nezabíraly zbytečně mnoho paměťového prostoru při běhu ostatních variant optimalizačních algoritmů, které jej ke své činnosti nepotřebují.

Opět jsou využity funkce ze souboru `nn.py` pro zjištění fitness a inicializaci počáteční populace neuronových sítí.

Výstup

Každý ze zmíněných souborů obsahujících optimalizační algoritmus obsahují také kód pro zapisování průběžných výsledků do souborů nacházejících se ve složce `output`. Mimo jiné je i tento výstup v přehlednější formě vypisován na standardní terminálový výstup. Jedná se o číslo generace, hodnotu nejlepšího jedince fitness a jeho přesnost. Do souboru je také zazna-

menáno finální fitness na validačním setu nejlepšího jedince na konci celého algoritmu. Soubory také obsahují kód pro měření celkového času trvání jednotlivých optimalizací. Všechny tyto údaje nám poslouží pro porovnání jednotlivých algoritmů a variant optimalizací jak mezi sebou, tak k jejich porovnání s algoritmem back-propagation.

Soubor `graph.py` obsahuje implementaci grafického zobrazení výsledků z jednotlivých experimentů. Jedná se o samostatně běžící program, který není nijak závislý na hlavním programu. Jako vstupní argument bere název souboru, který obsahuje informace o průběhu jednotlivých optimalizací (nacházející se ve složce `output`). Tento argument čte z příkazové řádky při spuštění. Na výstup nám dává dvojici grafů znázorňujících měnící se hodnotu fitness a přesnosti sítě v závislosti na počtu iterací (generací) algoritmu. Při spuštění lze zadat také více cest k souborům pro vytvoření kombinovaného srovnávacího grafu více variant optimalizace.

Vykreslování grafů je implementováno pomocí knihovny `matplotlib`, která je určená pro vykreslování grafů za pomocí matematické knihovny Numpy v programovacím jazyce Python3.

Konfigurační soubor

K programu je v kořenové složce přiložen i konfigurační soubor `config`, ve kterém lze nastavit dodatečné parametry jednotlivých optimalizačních algoritmů. Tyto parametry nehrají zásadní roli při našich experimentech, a tak byly jednorázově nastaveny a po celou dobu experimentů neměněny. Nicméně zůstává zde možnost jejich případné změny v této podobě. Pohodlně a bez přímého zásahu do kódu.

Kapitola 6

Experimenty a srovnání s Back-propagation

V této kapitole budou provedeny a rozebrány jednotlivé experimenty s evolučními algoritmy a jejich následné porovnání s algoritmem back-propagation. Výsledky experimentů budou rozděleny do několika částí, a to podle jednotlivých optimalizačních algoritmů, které budou optimalizovat váhy neuronových sítí. Dále také podle jednotlivých navržených variant spolupráce těchto algoritmů s back-propagation a v neposlední řadě také podle jednotlivých datových sad použitých pro tyto experimenty. Budeme sledovat nastavení jednotlivých parametrů a například i velikost populace a zjišťovat, které z nich jsou klíčové a s největším úspěchem v poměru rychlost/kvalita optimalizují jednotlivé váhy a zda-li jsou vůbec tyto metody schopné řádného naučení neuronové sítě pro její reálné použití.

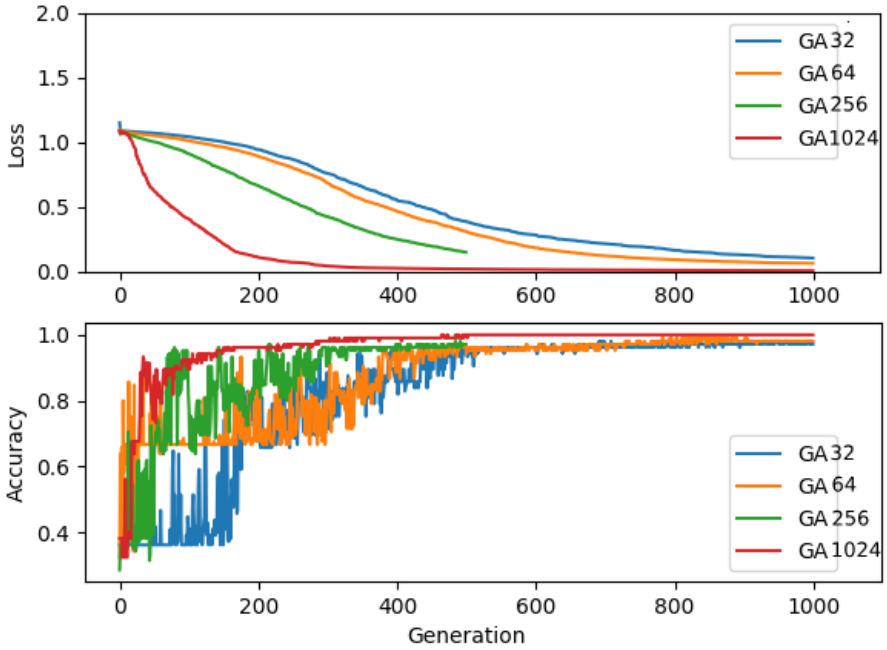
Pro srozumitelnější popis u experimentů budeme využívat názvosloví, kdy *verzí* budeme označovat evoluční algoritmus (dále jen EA), který se liší pouze v nějakém svém parametru nebo např. velikosti populace, avšak stále se jedná o stejný EA. To platí také pro verze DE (*rand/1, rand/2, best/1, best/2*). Pro typy námi navrhnutých optimalizátorů viz. 5.3, 5.4 a 5.5 budeme užívat název *varianta* s číslicí *1,2,3*. Pokud budeme mluvit o určitém EA ve srovnání s ostatními EA, budeme užívat výraz *metoda*.

6.1 Genetický algoritmus

V prvním experimentu Obrázek 6.1 sledujeme vliv velikosti populace na vývoj učení pomocí genetického algoritmu, dále jen GA ve *variantě 1*. Tabulka 6.1 popisuje počáteční parametry GA.

Parametry algoritmu GA					
Dataset	Iris	Wine	Mnist	**Iterace**	500 - 1000
Síť	(10,10)	(10,10)	(20,20)	**Turnaj**	5
Mutace	10%			**Mutation rate**	0.5
Gen. obměna	0.5 - 1			**Learn rate**	0.15

Tabulka 6.1: Síť - obsahuje počty neuronů ve skrytých vrstvách - hodnoty byly zvoleny experimentálně s cílem snížení výpočetní náročnosti, avšak při zachování schopností sítě



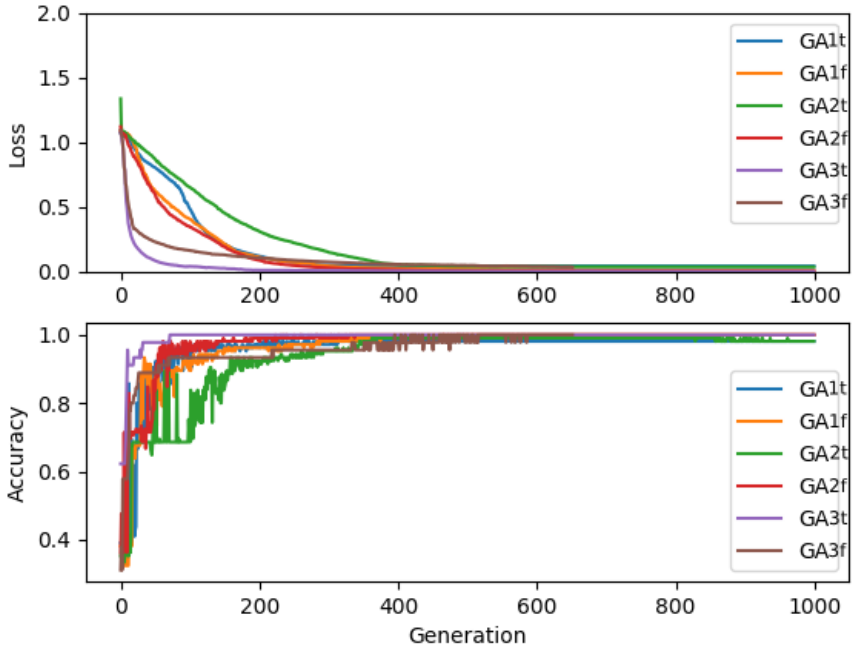
Obrázek 6.1: GA varianta 1: M-32, O-64, Z-256, Č-1024 jedinců (Iris)

Tabulka výsledků 6.2 pro experiment srovnávající účinnost GA na základě velikosti populace při datasetu Iris.)

Výsledky GA Varianta návrhu 1			
Populace	**Chyba**	**Přesnost**	**Čas**
32	0.112	0.95	246s
64	0.021	1.0	520s
256	0.141	0.97	1047s (500 iter)
1024	0.045 0.255	1.0 0.97	13540s G.O. 8484s

Tabulka 6.2: (G.O. - generační obměna - úplná, Iter - počet iterací

Z tohoto experimentu lze jasně vidět, že z hlediska úspěšnosti nalezení řešení se GA nejlépe chová při větších velikostech populace. Mnohem rychleji konverguje ke globálnímu minimu chybové funkce a to již po cca 200 iteracích. Tato skutečnost může být odůvodněna tím, že při větší populaci je mezi jedinci větší diverzita, a tak se zde nachází i více jedinců s dobrými vlastnostmi. Navíc při zachování malé velikosti turnaje (5) je vysoká pravděpodobnost výběru takovýchto různorodých, ač třeba i slabších jedinců. Tato rychlá konvergence z pohledu počtu generací je však vyvážena velice dlouhou dobou evaluace. Pro populaci o velikosti 1024 trvala evaluace 1000 generací průměrně 11000s (3h) u nejjednoduššího datasetu (Iris) 6.2. Při porovnání verze s 32 jedinci je to téměř 100 násobný rozdíl v čase potřebném na evaluaci. Tyto verze s méně jedinci také dosáhly prakticky stejné přesnosti po 1000 iteracích (průměrně 98%), avšak potřebovaly k tomu prakticky celou délku běhu algoritmu.



Obrázek 6.2: GA: 1024 jedinců (Iris), srovnání 3 variant, kde t značí úplnou generační obměnu a f poloviční generační obměnu

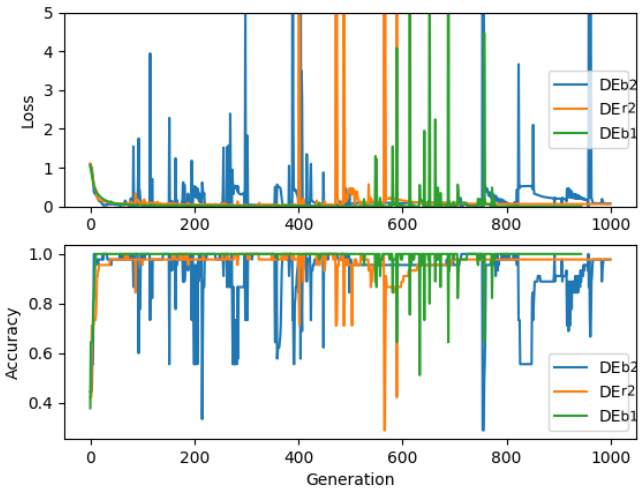
Při porovnání *variant 1, 2 a 3* u populace čítající 1024 jedinců lze vidět nejrychlejší konvergenci *varianty 3*, tedy 2. modifikace, kdy byla každá síť 5 iteracemi natrénována a až poté se začala účastnit GA. Tato prudká konvergence začíná již kolem 20. iterace algoritmu a kolem 100 již zůstává hodnota chyby prakticky neměnná. Tedy dosáhla globálního minima. Můžeme si také všimnout, že tato verze užívá poloviční generační obměnu, která mimo jiné razantně šetří výpočetní čas, řádově třetina výpočetního času byla ušetřena oproti verzi s úplnou generační obměnou. To může být odůvodněno tím, že tento čas je jinak potřebný na vytvoření nových jedinců v procesu křížení a také na turnajový výběr celého počtu populace.

Průměrným výsledkem skončily *varianty 2*, tedy kdy jsou sítě natrénovány v každé iteraci algoritmu, ale jejich fitness je vypočteno po průchodu validačního setu. To je poměrně zajímavé, protože ostatní evoluční algoritmy měly s touto *variantou* problémy a nedosáhly nějakých zásadnějších výsledků. Zejména to byla diferenciální evoluce u které nemůžeme mluvit o užitečném učení. Skóre jak chybové funkce tak přesnosti se neustále mění opačným směrem, bez tendencí konvergovat k nějaké stálé hodnotě viz Obrázek 6.3.

Je to z toho důvodu, že se neustále mění hodnoty vah díky průchodu BP, a tak se i celkové fitness každého jedince s každou generací velmi mění. Nicméně i zde jsou vidět tendence zůstávat delší dobu u globálního minima, ale také neschopnost algoritmu se v takovém minimu delší dobu udržet. Zde se nabízí možnost řešení v podobě ukončení algoritmu po dosažení určité hranice přesnosti, pak by tento problém byl již irelevantní. V experimentech jsme ale tuto možnost záměrně vynechávali, abychom mohli důkladněji sledovat činnost jednotlivých algoritmů, a to i po nalezení optimálního řešení.

Varianta 2 u GA při datasetu MNIST se nechová, tak jako u DE 6.3, ale naopak se zde velmi projevil vliv BP, který prakticky naučil síť hned při první iteraci, blíže v sekci 6.4.

Celkově můžeme říci, že GA se prakticky vždy dostane na nebo velice blízko globálnímu minimu, ovšem za cenu vysoké časové náročnosti, která daleko převyšuje přednosti této metody. Je bezkonkurenčně nejpomalejší metodou ze všech optimalizačních přístupů, a tak její využití v surové formě je spíše sporadické. Můžeme jí pomoci například použitím nějaké heuristiky v podobě krátkého natrénování sítě za pomocí BP nebo jiného přístupu přiblížení k optimálním hodnotám řešení. Pak by tato metoda dostávala větší význam, hlavně díky své vlastnosti, kdy nemá problém jako ostatní metody - uváznutí v lokálním extrému.



Obrázek 6.3: Srovnání 3 variant DE při (*Variantě 2*) na datasetu Iris

Očekávanou nejméně úspěšnou variantou je základní *varianta 1*, které jako jediné nebylo dopomoženo ke správným hodnotám vah, a tak musel GA začínat s absolutně náhodnými hodnotami. Ačkoliv i ony dokázaly dosáhnout úspěšnosti kolem (98%), jejich průběh je však v porovnání s ostatními velmi pozvolný a až průměrně kolem 300. iterace se začínají blížit k relevantnějším výsledkům nad (95%) přesnosti u datasetu Iris.

Souhrnná tabulka výsledků 6.3 jednotlivých verzí a variant GA při populaci 32 jedinců:

Srovnání variant GA					
Varianta	**Chyba**	**Přesnost**	**Doba běhu**	**Dataset**	**Iterace**
1	0.55	0.74	296s	Wine	1000
2	0.54	0.74	807s	Wine	1000
3	0.48	0.76	299s	Wine	1000
1	0.05	0.99	235s	Iris	1000
2	0.04	0.95	699s	Iris	1000
3	0.02	1.0	243s	Iris	1000
1	0.44	0.83	4485s	MNIST	500
1	0.26	0.89	8257s	MNIST	1000
2	0.05	0.98	20s	MNIST	3*
3	0.03	0.99	5116s	MNIST	500

Tabulka 6.3: 3* - BP prakticky naučí celou síť, a tak je pro MNIST tato varianta nevhodná

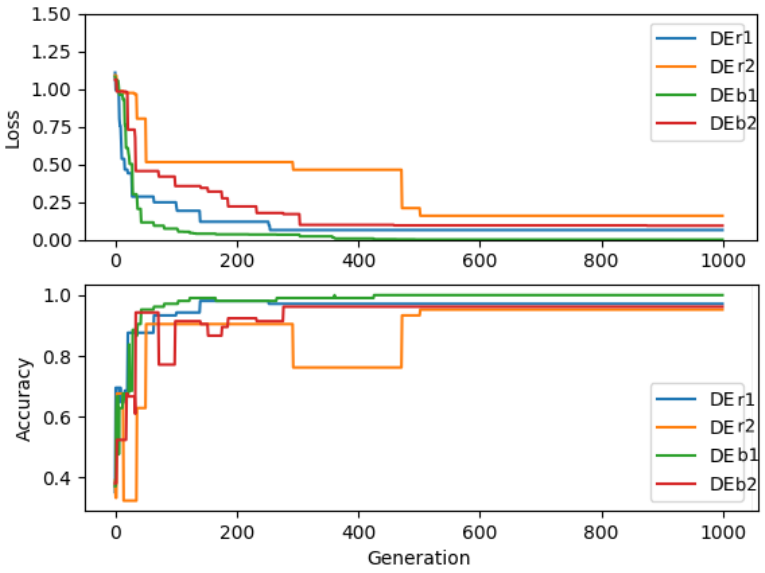
6.2 Diferenciální evoluce

V této sadě experimentů srovnáváme jednotlivé verze diferenciální evoluce, dále jen DE mezi sebou a také stejné verze při svých *variantách návrhu*.

Parametry algoritmu DE					
Dataset	Iris	Wine	Mnist	**Iterace**	500 - 1000
Síť	(10,10)	(10,10)	(20,20)	**Cross-point**	0.7
Křížení	binomické			**Mutation rate**	0.6

Tabulka 6.4: Základní parametry DE, které byly zvoleny experimentálně. *Síť* opět popisuje počty neuronů ve skrytých vrstvách

Nejzajímavěji se chová verze *best/1/bin* viz Obrázek 6.4, která ve většině případech neuvázla v lokálních extrémech tak, jako ostatní verze, ale nalezla optimální řešení. Dokonce i u složitějšího datasetu MNIST dokázala jako jediná nalézt řešení s 99% přesností. Z tohoto pohledu je nejméně úspěšnou verzí paradoxně verze *rand/2/bin*, která daleko častěji uvázne v lokálním optimu a v rozumném čase se z něj nedokáže vymanit. Druhou nejúspěšnější verzí je *best/2/bin*. Úspěch obou verzí není až tak nečekaný, už jen z toho důvodu, že jako kandidátního jedince využívá nejlepšího jedince v rámci populace, tudíž algoritmus se vždy snaží vylepšovat již schopného jedince. Na rozdíl od verzí *rand/1* a *rand/2*, kdy takto modifikuje vždy náhodného. Ovšem spíše bychom očekávali větší úspěšnost verze *best/2* nad *best/1*, jelikož skládá svůj šumový vektor z více jedinců z populace, a tak nabízí větší diverzitu z výběru jednotlivých genů. To se ovšem nepotvrdilo, a tak jednodušší verze (*best/1*) ve většině experimentů poráží všechny své konkurenční verze. Verze *rand/1* je na děleném druhém místě, jelikož poměrně nečekaně prokazuje velice dobré výsledky, kdy hned při začátku algoritmu má jednu z nejrychlejších konvergencí k nižším hodnotám chyby ze všech verzí DE.



Obrázek 6.4: DE *Varianta 1*, (populace 32) - (Iris), nejstabilněji se chová verze *best/1*

Celkově u DE velmi záleží na počáteční inicializaci hodnot vah sítě. Občasně se stane, že algoritmus hned několik málo iterací po začátku uvázne na lokálním optimu a ani několik stovek iterací mu nestačí na to, aby se z něj dokázal vymanit. Pokud ale tento problém nenastane, máme vcelku zaručenou poměrně rychlou konvergenci ke globálnímu minimu chybové funkce.

Souhrnná tabulka výsledků 6.5 jednotlivých verzí a variant DE:

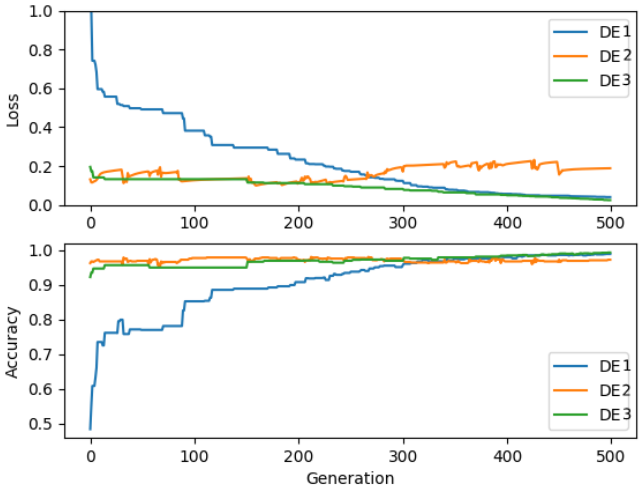
Srovnání variant DE							
Var.	**Metoda**	**Chyba**	**Přes.**	**Běh**	**Minimum**	**Dataset**	**Iterace**
1	best/1	0.47	0.75	150s	58. (0.53,0.77)	Wine	500
1	rand/1	0.51	0.75	152s	375. (0.57,0.76)	Wine	500
2	best/1	0.56	0.72	455s	28. (0.56,0.74)	Wine	500
2	best/2	0.53	0.75	463s	480. (0.51,0.83)	Wine	500
3	best/1	0.53	0.73	153s	82. (0.55, 0.75)	Wine	500
3	rand/1	0.54	0.72	150s	165. (0.54, 0.72)	Wine	500
1	best/1	0.01	0.99	4491s	357. (0.03, 0.99)	MNIST	500
3	best/2	0.12	0.96	4509s	55. (0.12, 0.96)	MNIST	500
3	rand/1	0.11	0.96	4639s	67. (0.11, 0.96)	MNIST	500
1	best/1	0.00	1.00	246s	361. (0.009, 1.0)	Iris	1000
1	rand/1	0.09	0.96	256s	304. (0.09, 0.96)	Iris	1000
1	best/2	0.06	0.97	238s	252. (0.09, 0.98	Iris	1000
2	best/1	0.07	0.97	804s	21. (0.1, 0.97)	Iris	1000
3	best/2	0.0002	1.0	250s	292. (0.002, 1.0)	Iris	1000
3	best/1	0.0007	1.0	239s	302. (0.06, 0.98)	Iris	1000
3	rand/1	0.07	0.97	243s	206. (0.14, 0.97)	Iris	1000

Tabulka 6.5: Srovnávací tabulka jednotlivých *variant* optimalizace a jednotlivých verzí DE při těchto variantách. *Minimum* - sloupec se zajímavými hodnotami z průběhu algoritmu (obvykle nejlepší řešení) viz. text níže

Jelikož je u DE velké množství verzí, byly do shrnující tabulky 6.5 zahrnuty jen vybrané experimenty s nejzajímavějšími výsledky. Mimojiné sloupec s názvem *Minimum* v sobě zahrnuje číslo iterace, při kterém měla DE nejlepší výsledky na základě nejvyšší přesnosti. Údaje jsou v pořadí (číslo iterace, hodnota chyby, hodnota přesnosti). Díky principu činnosti DE neznamená, pozdější iterace = lepší výsledky, tak jako u GA. DE dosáhne minima u velikosti chyby, což je také fitness funkcí pro algoritmus, nicméně u přesnosti již poté může a také dochází k jejím výkyvům a v pozdější fázi algoritmu také k tzv. *overfittingu*, tudíž proto je zahrnut i sloupec s těmito významnými údaji.

Dalším zajímavým aspektem u DE je její schodovitý průběh směrem k minimu, který u ostatních algoritmů nevidíme. Někdy se může zdát, že DE již uvázla v lokálním optimu a nedostane se z něj, ale poté po několika desítkách iterací se algoritmus opět "probudí"a optimalizuje dále.

Příklad verze DE *best/1* jež dosáhla přesnosti 100% a neuvázla v lokálním optimu 6.5.



Obrázek 6.5: *Varianty 1, 2, 3* při DE best/1/bin (Populace 32) na datasetu MNIST

6.3 Optimalizace hejnem částic

Metoda optimalizace hejnem částic, dále jen PSO dokáže překvapivě produkovat jak skvělé, tak zároveň i nejhorší výsledky ze všech zmíněných algoritmů. V ojedinělých případech dokáže doslova po pár iteracích nalézt bezkonkurenčně nejrychleji optimální hodnoty, a tak konvergovat ke globálnímu minimu mnohem rychleji než back-propagation. Tím je myšlena rychlost z pohledu času i z pohledu počtu iterací. Je to jediný algoritmus, který projevil takovéto známky chování viz Obrázek 6.6. Nicméně v častějších případech algoritmus uvázne na lokálním minimu a již se z něj nevymaní. PSO nemá tu schopnost dostat se z lokálního optima, a tak primárně záleží na inicializaci populace. Pokud jsou již od začátku v populaci jedinci s dobrými vlastnostmi, má tento algoritmus tendence konvergovat velice rychle ke globálnímu minimu. Ovšem pokud takové jedince populace neobsahuje, což je bohužel ve většině případů, uvázne v pouhém optimu, které je daleko od toho, aby taková síť mohla být prohlášena za naučenou.

Parametry algoritmu PSO					
Dataset	Iris	Wine	Mnist	**Iterace**	500 - 1000
Síť	(10,10)	(10,10)	(20,20)	**Learn rate**	0.15
C1	1.5			**C2**	1.5
W	0.5				

Tabulka 6.6: Parametry W_1, C_1, C_2 byly zvoleny experimentálně.

Jako jediný z algoritmů však poměrně dobře pracuje ve *variantě č. 2*, tedy při každé iteraci je jedinec jedenkrát naučen back-propagation. Například u datasetu Kvality vína dosáhla tato varianta téměř 90% úspěšnosti, přitom samotná back-propagation stejně jako ostatní algoritmy vždy nalezly nejvýše řešení představující 76% přesnost sítě. V tomto ohledu je tedy PSO velmi účinná, ale jak bylo řečeno, je potřeba velké množství spuštění celého algoritmu, abychom takového výsledku dosáhli a neuvázli v pouhém optimu, což je z hlediska použitelnosti značně limitující.

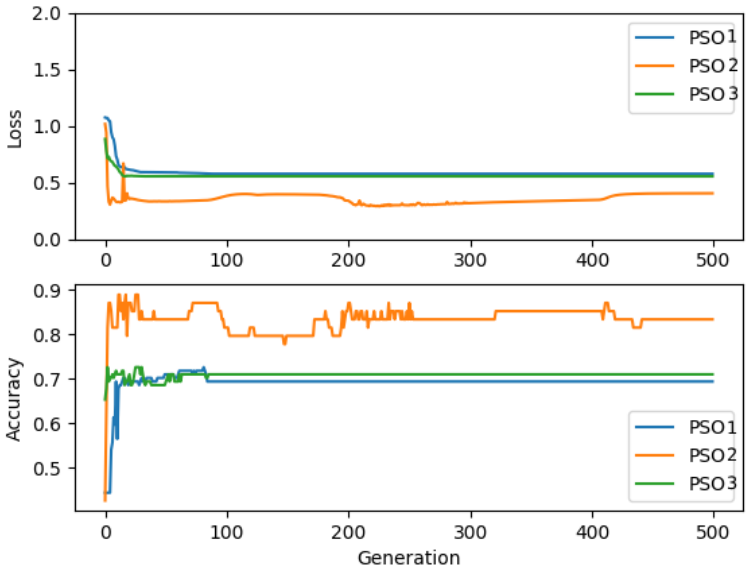
Souhrnná tabulka výsledků 6.7 jednotlivých variant PSO:

Srovnání variant PSO						
Var	**Chyba**	**Přesnost**	**Běh**	**Minimum**	**Dataset**	**Iterace**
1	0.05	0.98	246s	20. (0.08, 0.99)	Iris	1000
1	0.43	0.82	241s	10. (0.45, 0.83)	Iris	1000
2	0.004	1.0	714s	6. (0.12, 1.0)	Iris	1000
3	0.09	0.97	246s	13. (0.11, 0.98)	Iris	1000
3	0.22	0.88	259s	26. (0.27, 0.91)	Iris	1000
1	0.57	0.69	147s	81. (0.57, 0.72)	Wine	500
2	0.4	0.83	395s	11. (0.32, 0.88)	Wine	500
3	0.55	0.7	150s	25. (0.55, 0.72)	Wine	500
1	0.56	0.73	1681s	25. (0.57, 0.73)	MNIST	200
2	0.006	0.99	5021s	11. (0.02, 1.0)	MNIST	200
3	0.07	0.97	1759s	13. (0.08, 0.97)	MNIST	200

Tabulka 6.7: Tabulka výsledků pro PSO, Minimum opět označuje shodně jako u DE 6.5 zajímavé hodnoty z průběhu optimalizace

Také *třetí varianta* optimalizace, tedy krátké naučení před spuštěním algoritmu pracuje s PSO obstojně a ve většině případů algoritmus velice rychle konverguje ke globálnímu optimu. To nám naznačuje, že když zmenšíme stavový prostor řešení a správně algoritmus nasměrujeme, v našem případě použitím back-propagation, PSO velmi rychle a kvalitně nalezne dobré řešení.

Také z experimentů vyplynulo, že PSO se chová lépe a rychleji optimalizuje při použití menších populací (100 a méně jedinců). Při velké populaci se sice může zdát, že se lépe pokryje částicemi stavový prostor, nicméně k nalezení správného řešení to nijak znatelně nepřispívá a dosahujeme podobných, případně horších výsledků, tak jako s malými populacemi a navíc za mnohem delší časový úsek.



Obrázek 6.6: PSO (Populace 32) na datasetu Kvality vína, porovnání všech *3 variant* a jejich typický průběh, kdy v několika iteracích dosáhne lokálního nebo globálního optima a dále se již dramaticky jejich průběh nemění

6.4 Celkové srovnání a porovnání s Back-propagation

Při experimentech s BP jsme používali metodu, při které se upravují váhy po každém vzorku. Kdyby byl použit princip zpracování po dávkách, tak jako je dnes velmi běžné, byla by doba konvergence algoritmu k minimu ještě daleko rychlejší.

U datasetu Iris neměla BP žádný problém v krátkém čase nalézt globální minimum. U datasetu Kvality vína již byla situace poněkud horší a BP dokázala nalézt pouze lokální optimum vah sítě. U datasetu MNIST byla situace opačná, prakticky již po první iteraci BP byla přesnost takto naučené sítě pohybující se kolem 98%. Tato skutečnost proto zavrhuje *varianty 2 a 3* našeho návrhu optimalizace pomocí EA, jelikož by BP provedla většinu optimalizace vah sama. Nicméně při dosažení zmíněných 98% pak zejména GA nemá větší problémy nalézt takové hodnoty vah pro 100% přesnost klasifikace sítě v malém počtu iterací. U ostatních metod záleží na štěstí a náhodnosti hodnot, aby poté takové maximální přesnosti dosáhly.

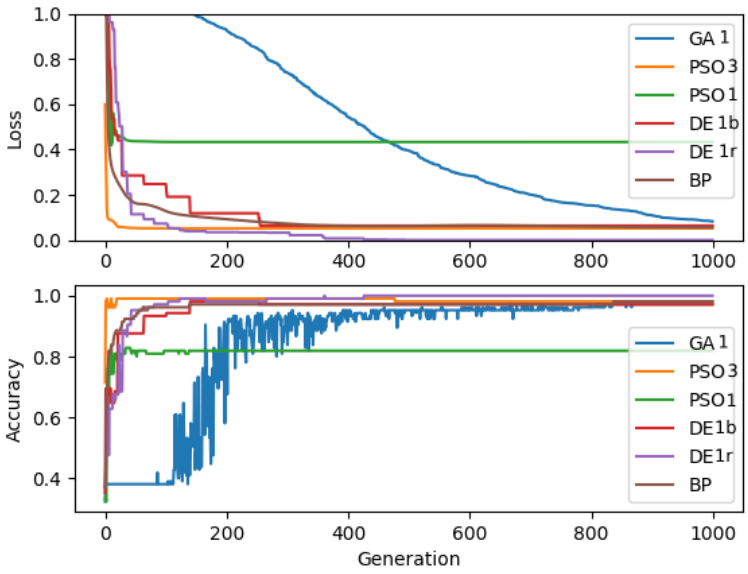
Tabulka 6.8 ukazuje průběrné výsledky samostatně běžící BP. Oproti našim algoritmům probíhá BP vcelku nepřekvapivě mnohonásobně rychleji, avšak v některých případech nedokáže nalézt globální minimum řešení. V tomto je vidina užitečnosti našich optimalizačních metod, protože např. PSO u datasetu kvality vína zhruba ve 20% případů dokáže nalézt lepší hodnoty vah než samotná BP. U ostatních datasetů zase v mnohem menším počtu iterací než BP. U DE je toto procento menší, ovšem také při správné inicializaci vah náhodnými čísly dokáže takové hodnoty nalézt v nižším počtu iterací než právě BP, případně nalezne podobné řešení v také relativně malém počtu iterací. U GA je průběh pozvolnější, za to máme ale jistotu, že dosáhne nebo se velmi přiblíží globálnímu minimu, což se u ostatních metod říci nedá. GA je tedy pozvolnější a pomalejší než ostatní metody, avšak vždy dosáhne optimálních nebo téměř optimálních hodnot.

Výsledky BP					
Chyba	**Přesnost**	**Běh**	**Minimum**	**Dataset**	**Iterace**
0.55	0.68	20s	167. (0.57, 0.75)	Wine	1000
0.0002	1.0	72s	7. (0.002, 1.0)	MNIST	100
0.05	0.98	17s	139. (0.1, 0.97)	Iris	1000

Tabulka 6.8: Tabulka výsledků BP, Minimum opět označuje zajímavé údaje z průběhu algoritmu ve tvaru: (číslo iterace, hodnota chyby, přesnost)

Co se týče průběhu BP, tak je z hlediska počtu iterací její průběh a konvergence podobná s ostatními algoritmy, nicméně za zlomek výpočetního času. V tomto ohledu tedy naše optimalizátory značně pokulhávají za standardní BP.

Celkově se pro budoucí experimenty nejvíce hodí právě DE a PSO. Rozmanitost verzí u DE z ní dělá skvělý případ pro větší zkoumání v této oblasti optimalizace neuronové sítě. U PSO by se mohlo jít více do hloubky v oblasti heuristik, které algoritmus před začátkem optimalizace nasměrují ke globálnímu minimu, poté jej totiž PSO bez větších problémů a v poměrně krátkém čase téměř vždy spolehlivě nalezne.



Obrázek 6.7: Srovnání *variant 1 a 3* optimalizačních algoritmů s Back-propagation na datasetu Iris (U DE, první zmiňovaný (v legendě) je *varianta 1* a druhý *varianta 3.*)

Z důvodu nedostatku prostoru zde v práci a případném větším zájmu o nahlédnutí na další výsledky ostatních experimentů, byly na paměťové médium do adresáře `graphs` přiloženy grafy a textové soubory z dalších provedených experimentů.

Kapitola 7

Závěr

V této práci bylo cílem prozkoumat možnosti spolupráce evolučních algoritmů při učení neuronových sítí a jejich následné porovnání s algoritmem *back-propagation*. Pro tyto experimenty byl také vytvořen prostředek v programovacím jazyce Python bez použití knihoven třetích stran zaměřených na strojové učení. V rešeršní části byly v krátkosti představeny dostupné evoluční algoritmy, rozebrány základy a princip fungování neuronových sítí a následně byl uveden přehled již existujících variant spolupráce těchto dvou oblastí umělé inteligence, která se v moderní literatuře nazývá *neuroevolucí*. Pro optimalizaci byly vybrány tři evoluční algoritmy. Těmito algoritmy byly *genetický algoritmus* (GA), *diferenciální evoluce* (DE) a *optimalizace hejnem částic* (PSO). Tyto algoritmy měly za úkol optimalizovat váhy a biasy uvnitř dopředné, plně propojené hluboké neuronové sítě s fixní topologií. Tato síť sloužila k řešení klasifikačních problémů a klasifikaci do tříd. Pro tyto účely byly zvoleny 3 klasifikační datasety se kterými byly následně provedeny experimenty.

Byly navrženy 3 varianty přístupů k optimalizaci vah a biasů neuronové sítě, které nahrazovaly klasické učení pomocí *back-propagation* (BP). Z těchto experimentů vyplynulo, že nejlépe neuronovou síť naučila *varianta 3*, tedy varianta, kdy byla síť několika málo iteracemi BP předtrénována před spuštěním samotného optimalizačního algoritmu. Z těchto algoritmů se nejlépe dařilo diferenciální evoluci, která ve většině případů nalezla optimální řešení v rozumném výpočetním čase. Neméně kvalitně se projevila také varianta optimalizace hejnem částic, která zhruba ve 20% případů nalezla dokonce lepší řešení než algoritmus back-propagation. Jako nejméně úspěšnou metodu musíme prohlásit genetický algoritmus, který sice prakticky vždy dosáhl globálního optima a tedy natrénoval síť kvalitně, nicméně toto natrénování bylo vykoupeno obrovským výpočetním časem, který by byl v praxi hlavní zábranou v rozšíření této metody. Také jako jediná metoda během experimentů neměl problémy s uváznutím v lokálním minimu, se kterým měly zbylé 2 metody občasné potíže.

Co se týče srovnání s back-propagation, nepodařilo se nám porazit tuto standardní metodu učení z hlediska výpočetního času nebo spolehlivosti kvalitního naučení sítě. Algoritmus BP je mnohokrát rychlejší a stabilnější metodou učení než ty zde probrané. Nicméně naše přístupy by se mohly při dalším vývoji jistě lépe optimalizovat, a to například paralelizací svých výpočtů nebo využitím hardware grafické karty, což by mělo za následek rapidní nárůst jejich výpočetní výkonnosti a konkurenceschopnosti s BP.

Z hlediska budoucího vývoje je oblast neuroevoluce velice zajímavou a stále ještě ne zdaleka probádanou oblastí, kterou by se obor umělé inteligence rozhodně měl v budoucnu podrobněji zabývat.

Literatura

[1] Kaggle - *Prediction of quality of Wine*. [Online; navštíveno 10.12.2018].
URL https://www.kaggle.com/vishalyo990/prediction-of-quality-of-wine

[2] *Tensorflow: MNIST Data Set* . [Online; navštíveno 5.3.2019].
URL https://www.tensorflow.org/tutorials

[3] UCI - Machine Learning Repository -*Iris Data Set* . [Online; navštíveno 9.12.2018].
URL https://archive.ics.uci.edu/ml/datasets/iris

[4] UCI - Machine Learning Repository - *Wine Quality Data Set*. [Online; navštíveno 10.12.2018].
URL https://archive.ics.uci.edu/ml/datasets/wine+quality

[5] Wikipedia, The Free Encyclopedia - *Wikipedia Darwinism*. [Online; navštíveno 6.11.2018].
URL https://en.wikipedia.org/wiki/Darwinism

[6] Wikipedia, The Free Encyclopedia - *Wikipedia Neuron*. [Online; navštíveno 3.12.2018].
URL https://simple.wikipedia.org/wiki/Neuron

[7] Ahamad, M. S.; Mohammed, T. K.; Al-Betarb, A.: *A survey on applications and variants of the cuckoo search algorithm*. [Online; navštíveno 9.2.2019].
URL https://doi.org/10.1016/j.asoc.2017.02.034

[8] Brabazon, A.; O'Neill, M.; McGarraghy, S.: *Natural Computing Algorithms*. Springer-Verlag Berlin Heidelberg, 2015, ISBN 978-3-662-43630-1.

[9] Brownlee, J.: *Clever Algorithms - Evolution Strategies* . [Online; navštíveno 5.2.2019].
URL http://www.cleveralgorithms.com/nature-inspired/evolution/evolution_strategies.html

[10] Chiroma, H.; Noor, A. S. M.; Abdulkareem, S.; aj.: *Neural Networks Optimization through Genetic Algorithm Searches: A Review*. 2017, [Online; navštíveno 8.12.2018].
URL http://dx.doi.org/10.18576/amis/110602

[11] Cornelisse, D.: *An intuitive guide to Convolutional Neural Networks*. [Online; navštíveno 28.1.2019].
URL http://bit.ly/2XUnwAG

[12] Ding, S.; Li, H.; Su, C.; aj.: *Evolutionary artificial neural networks: a review*. [Online; navštíveno 1.12.2019].
URL https://doi.org/10.1007/s10462-011-9270-6

[13] García-Pedrajas, N.; Ortiz-Boyer, D.; Hervás-Martínez, C.: *An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization*. 2006, [Online; navštíveno 16.11.2018].
URL http://bit.ly/2XPT6ja

[14] Gunipati, T.: *Must-know Machine Learning Questions – Linear Regression*. [Online; navštíveno 20.2.2019].
URL https://www.upgrad.com/blog/must-know-machine-learning-questions-linear-regression

[15] Hoekstra, V.: *An overview of neuroevolution techniques*. 2011, [Online; navštíveno 27.11.2018].
URL
https://beta.vu.nl/nl/Images/werkstuk-hoekstra_vincent_tcm235-248302.pdf

[16] J. Kennedy and R. Eberhart: Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948. - *Particle Swarm Optimization*. 1995, [Online; navštíveno 9.2.2018].
URL https://doi.org/10.1109/ICNN.1995.488968

[17] KARGER, M.: *Algoritmus Diferenciální Evoluce s prvky deterministického chaosu (ChaosDE) v prostředí Mathematica*. Diplomová práce, Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky, Zlín, 2011 [Online; navštíveno 18.1.2019].
URL https://theses.cz/id/xqgoho/

[18] Kearney, W. T.: *Using Genetic Algorithms to Evolve Artificial Neural Networks*, Honors Theses. Paper 818. 2016, [Online; navštíveno 25.11.2018].
URL http://digitalcommons.colby.edu/honorstheses/818

[19] Khan, M.; Ahmad, A.; Khan, G.; aj.: *Fast learning neural networks using Cartesian genetic programming*. 2013, [Online; navštíveno 10.11.2018].
URL http://bit.ly/2XPT6ja

[20] Koehn, P.: *Combining Genetic Algorithms and Neural Networks: The Encoding Problem*. 1994, [Online; navštíveno 7.1.2019].
URL http://homepages.inf.ed.ac.uk/pkoehn/publications/gann94.pdf

[21] Kvasnička, V.; kol.: *Úvod do teorie neuronových sítí*. Iris, 1997, ISBN 80-88778-30-1.

[22] Kvasnička, V.; Pospíchal, J.; Tiňo, P.: *Evoluční algoritmy*. STU Bratislava, 2000, ISBN 80-227-1377-5.

[23] LeCun, Y.: *MNIST Data Set* . [Online; navštíveno 4.3.2019].
URL http://yann.lecun.com/exdb/mnist/index.html

[24] Mehrotra, K.; Mohan, C. K.; Ranka, S.: *Elements of Artificial Neural Networks*. MIT Press, 1997, ISBN 0-262-13328-8.

[25] Senaratna, N. I.: *Genetic Algorithms: The Crossover-Mutation Debate*. [Online; navštíveno 7.1.2019].
URL https://www-cs.stanford.edu/people/nuwans/docs/GA.pdf

[26] Serengil, S. I.: *A Gentle Introduction to Cross-Entropy Loss Function* . [Online; navštíveno 18.1.2019].
URL http://bit.ly/2GQVaBN

[27] Shibuya, N.: *Demystifying Cross-Entropy* . [Online; navštíveno 17.1.2019].
URL
https://towardsdatascience.com/demystifying-cross-entropy-e80e3ad54a8

[28] Shiffman, D.: *Neuroevolution Flappy Bird* . [Online; navštíveno 5.12.2018].
URL http://bit.ly/2GQRPTa

[29] Stanley, K. O.; Miikkulainen, R.: *Evolving Neural Networks through Augmenting Topologies*. [Online; navštíveno 1.11.2018].
URL https://ieeexplore.ieee.org/document/6790655

[30] Stanley, K. O.; Miikkulainen, R.: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600) (Efficient Evolution of Neural Network Topologies)*. IEEE, 2002, ISBN 0-7803-7282-4.

[31] Streichert, F.: *An overview of neuroevolution techniques*. [Online; navštíveno 19.11.2018].
URL http://www.ra.cs.uni-tuebingen.de/mitarb/streiche/publications/Introduction_to_Evolutionary_Algorithms.pdf

[32] Such, F. P.; Madhavan, V.; Conti, E.; aj.: *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning* . [Online; navštíveno 20.11.2018].
URL https://arxiv.org/abs/1712.06567

[33] Suchomel, O.; Hyhlík, T.: *Optimalizační metody v CFD – diferenciální evoluce*. [Online; navštíveno 20.1.2019].
URL http://www.suchomel.org/files/Optimalizace-v-CFD.pdf

[34] Tata, V.: *An Introduction to Evolutionary Algorithms and Code with Genetic Algorithm in Unity*. [Online; navštíveno 20.2.2018].
URL http://bit.ly/2XUnwAG

[35] Volná, E.: *Neuronové sítě 1*. 2008, [Online; navštíveno 17.11.2018].
URL http://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

[36] Zbořil, F.: *IZU - Základy umělé inteligence slides*. [Online; navštíveno 9.10.2018].
URL https://www.fit.vutbr.cz/study/courses/IZU/private/.cs

[37] Zbořil, F.: *SFS - Soft Computing slides*. [Online; navštíveno 13.2.2019].
URL https://www.fit.vutbr.cz/study/courses/SFC/private/.cs