EVO - Lineární genetické programování

xglosm01

April 2023

1 Zadání

Pro vypracování projektu jsem si zvolil téma lineárního genetického programování (LGP) pro klasifikaci objektů. Hlavním cílem je implementace *lineárního genetického algoritmu* a statistické vyhodnocení algoritmů LGP s různými nastaveními parametrů, včetně počtu registrů a jejich inicializačních hodnot. Dále se práce zaměřuje na vyhodnocení chování algoritmu při klasifikaci vizuálních datových sad, konkrétně na datové sady MNIST, FashionMNIST, CIFAR10 a CIFAR100 a v případě úspěchu další.

2 Implementace

Jako součást projektu vznikl modul `lga` - implementující variantu *lineárního genetického algoritmu*, pracující nad datasety určenými ke klasifikaci z modulu *torchvision*. Samotný modul se pak skládá ze dvou submodulů (`LGA` a `utils`) a skriptu (`lga_cli.py`), poskytujícího rozhraní algoritmu pro příkazovou řádku.

Zajímavostí modulu je vyšší úroveň abstrakce programu a instrukcí, které nejsou zakódovány přímo do binarního řetězce – instrukce i program je zapouzdřen do vlastních tříd, které implementují veškeré funkcionality nutné pro optimalizace genetickým algoritmem, které jsou blíže popsány níže. Tento přístup byl zvolen z důvodu použití vektorizovaných funkcí modulu *Pytorch* pracujících nad GPU (pokud GPU není k dispozici, je použito CPU). Dalším důvodem bylo použití instrukcí pracujících s tensory namísto skaláry. Binární zakódování by tak bylo příliš složité a vzhledem k vektorizovaným instrukcím je režie zpracování nezakódovaných programů (mutace, tvorba nových programů, atd...) vůči času stráveném výpočtem zanedbatelná.

Možnosti spuštění jsou popsány v souboru `README.md` (včetně argumentů pro skript `lga_cli.py`), případně na GitHubu, kde najdete vždy nejnovější verzi.

2.1 Modul `utils`

Soubory modulu:

- `__init__.py`
- `args.py` - Definuje argumenty pro skript `lga_cli.py` pro konfiguraci hyperparametrů algoritmu, datasetu a ostatní konfigurace skriptu. Implementuje také funkci pro jejich validaci.
- `datasets.py` - Definuje třídu `Dataset` a funkci pro změnu velikosti klasifikovaných objektů.
- `losses.py` - Definuje 2 fitness funkce: poměr správně klasifikovaných objektů a upraven8 křížová entropie (vynásobena -1, aby mohla být použita jako fitness funkce).
- `other.py` - Definuje vlastní výjímky a ostatní, nezařazené funkce.

2.1.1 Třída `Dataset`

Tato třída pracuje ve dvou konfiguracích:

- Testovací - data jsou vygenerována uměle a objekty jsou vektory nul s jedinou jedničkou na indexu rovném čísle třídy (one-hot kódování třídy objektu). Takový dataset slouží pouze k ověření, zda-li se ve zdrojových kódech nenacházejí fatální chyby a řešení je správně nalezeno.
- Obrázkový - je zvolen jeden z datasetů, které jsou poskytovány modulem `torchvision` a jsou určeny pro klasifikaci. Tyto datasety již slouží k ověření výkonnosti algoritmu.

Součástí třídy je metoda pro normalizaci dat do libovolného intervalu a také metoda pro její konverzi na řetězec, která vypíše hlavní atributy třídy ve strukturovaném formátu. Třída po inicializaci sama vytvoří nebo stáhne data, která jsou pak k dispozici přes atributy pro: trénovací sadu objektů, trénovací sadu tříd (labelů) a jejich testovací (validační) ekvivalenty.

Součástí modulu jsou také 2 funkce, které jsou používány jako fitness funkce pro vyhodnocení populace na datech. Jednou je `accuracy_score` a druhou `cross_entropy_loss`.

- `accuracy_score` - Udává poměr správně klasifikovaných objektů.
- `cross_entropy_loss` - Udává hodnotu křížové entropie mezi správnýma třídama a výstupními registry, vynásobenou -1, aby směr optimalizace modelu odpovídal předchozí funkci.

2.2 Modul `LGA`

Soubory modulu:

- `__init__.py`
- `lga.py` - Definuje třídu `LGA` - implementuje *lineární genetický algoritmus*.
- `program.py` - Definuje třídu `Program` - individuální člen populace *lga*.
- `instruction.py` - Definuje třídu `Instruction` - jedna instrukce programu.
- `operations.py` - Definuje operace pro instrukce, případně jejich bezpečné verze (NaN, nekonečno a minus nekonečno je nahrazeno 0) a jejich seskupení podle parity do seznamů.
- `mutations.py` - Definuje statickou třídu `Mutation`, která implementuje mutační procesy pro třídu `Program` a `Instruction`.

2.2.1 Třída `LGA`

Tato třída je nejvyšší abstrakcí *lineárního genetického algoritmu*. Při inicializaci je možné specifikovat hyperparametry algoritmu, včetně možnosti načtení prvotního programu do populace. Při inicializaci třídy jsou také inicializovány adresáře pro ukládání logů a nejlepšího modelu.

Dále třída implementuje metodu `LGA.fill_population`, která aktuální populaci doplní na maximální nastavenou hodnotu metodou `LGA.new_individual`. Ta v případě, kdy není u populace vyhodnocena fitness, vytvoří náhodného individuála pomocí metody `Program.create_random_program`. V případě, kdy je hodnota fitness v populaci vyhodnocena, je pro každého nového individuála s nastavenou pravděpodobností rozhodnuto, jestli dojde ke křížení, mutaci, růstu počtu instrukcí v programu nebo snížení počtu instrukcí. Tyto 4 události jsou na sobě nezávislé a mohou nastat všechny najednou, nebo také žádná (libovolná kombinace). V případě, že nemá nastat ani jedna z událostí, je opět vygenerován náhodný program. Pokud má dojít ke křížení, jsou náhodně (podle fitness či unifromně) vybráni 2 rozdílní individuálové, u kterých dojde ke křížení. V opačném případě je vybrán jediný individuál a je zkopírován.

Proces trénování je implementován v metodě `LGA.train`. Celý proces se opakuje (dle konfigurace) v bězích nezávisle na sobě v těchto krocích:

1. Je doplněna populace až do specifikovaného limitu
2. Je vyhodnocena funkce *fitness* pro celou populaci
3. V populaci je ponechána pouze *elitní* část populace (konfigurovatelná konstanta)
4. Je proveden záznam o trénování pro účely analýzy

Poté, co jsou všechny běhy dokončeny, je zavolána funkce `LGA.final_evaluation`, která naplní pole pro populaci posledními elitními individuály ze všech běhů a hromadně vyhodnotí jejich úspěšnost predikce na evaluační části datasetu. Nejlepší individuál a záznamy o trénování jsou uloženy pomocí modulu *pickle* do předem specifikovaných adresářů. Trénování skončí výpisem výsledků a nejlepšího programu.

2.2.2 Třída `Program`

Tato třída reprezentuje lineární program, který je řešením lineárního genetického algoritmu. Implementuje všechny potřebné genetické operace, jako jsou křížení, mutace a transkripce, a také vyhodnocení funkce fitness programu. Implementuje také funkce pro uložení ze / načtení do souboru. Při inicializaci je inicializován prázdný seznam, do kterého jsou později vygenerovány či kopírovány instrukce. Dalšími atributy programu jsou vstupní, pracovní a výstupní registry, se kterými instance třídy `Program` a `Instruction` provádí výpočty a inicializační hodnoty pracovních a výstupních registrů (hodnoty, na které jsou nastaveny příslušné registry před spuštěním lineárního programu).

Třída Program má několik důležitých atributů:

- `Program.lga` - statický atribut, který umožňuje přistoupit k objektu `LGA` ze kterékoli instance třídy `Program`. Čtením hyperparametrů algoritmu z jednoho místa redukuje režii při tvorbě nových jedinců.
- `hidden_register_initial_values` - Tensor s inicializačními hodnotami pracovního registru pro jeden objekt.
- `result_register_initial_values` - Tensor s inicializačními hodnotami výstupního registru pro jeden objekt.
- `input_registers` - Tensor obsahující datovou sadu, nad kterou je program právě spuštěn.
- `result_registers` - Tensor obsahující výsledné registry pro každý klasifikovaný objekt.
- `hidden_registers` - Tensor obsahující pracovní registry pro každý klasifikovaný objekt.
- `instructions` - Seznam instancí třídy `Instruction` - instrukce programu.

Statická metoda `Program.create_random_program` vytvoří náhodný program. Náhodně vytvoří minimální počet instrukcí a inicializační hodnoty registrů inicializuje ze standardního normálního rozložení. Statická metoda `Program.crossover` implementuje křížení dvou instancí třídy `Program`. U výsledného programu je seznam instrukcí vytvořen jednoduchým jednobodovým křížením a inicializační registry jsou sestaveny náhodnou kombinací inicializačních registrů rodičovských programů. Poslední statická metoda - `Program.transcription` pouze zkopíruje inicializační hodnoty registrů a instrukce, které vloží do nově vytvořené instance třídy *Program*. Tato instrukce je volána pouze v kombinaci s jakýmkoli typem mutace, čímž se zabrání existenci shodných programů v jedné generaci populace.

Evaluce programu je implementována v metodě `Program.evaluate`, která program vyhodnotí nad poskytnutou datovou sadou. Nejprve jsou inicializační hodnoty registrů nakopírovány tolikrát, kolik se v datové sadě nachází objektů. Pole s objekty je "nahráno"do vstupních registrů a postupně

na všech instrukcích programu je volána metoda `Instruction.execute`. Ta počítá přímo s registry rodičovského programu. Nad výstupními registry a seznamem korektních tříd je pak vypočítána fitness funkce pro daný program.

Další metodou je `Program.delete_instruction`, která vymaže buďto náhodnou nebo argumentem specifikovanou instrukci. Metody `Program.grow` a `Program.shrink` pak přidají nebo odeberou náhodnou instrukci. Vždy je zachován maximální a minimální stanovený počet instrukcí - má-li být jeden s limitů překročen, jsou aplikovány obě funkce pro opětovné splnění těchto podmínek.

Téměř všechny parametry jsou nastavitelné, jejich seznam najdete v souboru **README.md**.

2.2.3 Třída `Instruction`

Tato třída reprezentuje jeden příkaz v lineárním programu. Každá instance si uchovává odkaz na instanci třídy `Program`, do které patří, pro efektivní přístup k jeho parametrům a registrům. Tato třída je jedinečná nejen tím, že se najednou vykoná na celé datové sadě (na všech registrech nadřazeného programu), ale také kvůli možnosti tensoru jako vstupu do instrukce - operandu, který je pak jednou z agregujících funkcí (např: suma, produkt, minimum) zredukován do skalární hodnoty.

Třída Instruction má následující atributy:

- `parent`: odkaz na rodičovský program instrukce.
- `operation_parity`: parita operace instrukce.
- `operation_function`: funkce implementující operaci instrukce.
- `input_registers`: jména vstupních registrů, seznam délky dle parity.
- `output_register`: jmého výstupního registru.
- `input_register_indices`: indexy do vstupních registrů.
- `output_register_indices`: index do výstupního registru.
- `is_area_operation`: zda instance třídy Instruction pracuje s tensory.
- `area_operation_function`: funkce pro reduci tensorů na hodnoty.

Instrukce vznikají buďto zavoláním metody `Instruction.copy`, která udělá hloubkobou kopii instrukce. Další možností je metoda `Instruction.random`, která vygeneruje náhodnou instrukci tímto procesem:

1. Rozhodne, jestli instrukce bude zpracovávat tensory (s pravděpodobností `p-area`).
2. Pokud se jedná o instrukci zpracující tensory, nastav paritu na 1. Jinak nastav paritu náhodně podle počtu binárních a unárních operací (každá operace má stejnou pravděpodobnost zvolení).
3. Zvol konkrétní operaci (uniformní vzorkování z operací vybrané parity).
4. Náhodně zvol vstupní registrové pole (dle parity), pravděpodobnost výběru registrového pole je přímo úměrná celkovému počtu registrů v daném poli.
5. Náhodně zvol výstupní registrové pole, pravděpodobnost výběru registrového pole je přímo úměrná celkovému počtu registrů v daném poli.

6. Pokud je instrukce pracující s tensory:
    - `True` náhodně zvol pro každý rozměr vstupního registru seznam náhodných indexů až o velikosti poloviny daného rozměru.
    - `False` náhodně zvol index do vstupních registrů pro každý jejich rozměr.
7. Náhodně zvol index do každého rozměru výstupního registru.

Pro výpočet instrukce je volána metoda `Instruction.execute`. Ta pomocí metody `Instruction.obtain_operands` získá vstupní operandy instrukce, aplikuje operaci pomocí metody `Instruction.compute` a metodou `Instruction.save_result` uloží výsledek výpočtu do výstupního registru. Metoda `Instruction.obtain_operands` získá ze vstupního tensoru subtensor-operand indexováním seznamem v každém rozměru. Nad každou dimenzí tenzoru se pak zavolá funkce `torch.index_select`, viz oficiální dokumentace.

Třída také implementuje metodu pro srozumitelnou konverzi na řetězec, jejíž výstup může vypadat například takto:

```
result_registers[2] = add(input_registers[11,12], hidden_registers[1,0])
     hidden_registers[0] = identity(hidden_registers[(3,2)).mean()
```

2.2.4 Třída `Mutations`

Statická třída `Mutation` implementuje všechny operace mutace pro instance tříd `Program` a `Instruction`. Hlavní metodou je `Mutations.mutate_program`:

1. Nejprve náhodně určí počet registrů a instrukcí určených k mutaci, vždy je vybráno náhodné číslo mezi 1 a parametrem specifikovanou maximální hodnotou.
2. Náhodně se zvolí registry (počet určen v předchozím kroku), ke kterým jsou připočteny hodnoty vzorkované ze standartního normálního rozložení.
3. Náhodně se vyberou instrukce (počet určen v kroku 1.), nad kterými je zavolána funkce `Mutations.mutate_instruction`.

Funkce `Mutations.instruction` s pravděpodobností stanovenou hyperparametrem `p-area` promění funkci zpracující tensory na funkci zpracující skalární hodnoty - `Mutations.mutate_area`. Pokud není zvolena tato metoda, je pak náhodně (se stejnou pravděpodobností) zvolena jedna ze tří dalších funkcí.

Funkce pro mutaci instrukce:

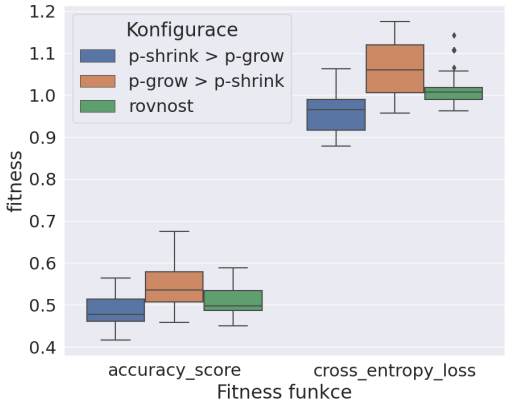
- `Mutations.mutate_area` - Pokud operace pracuje s tensory: odstraň operaci redukující tensor na skalár, ponech základní unární operaci, vyber index do vstupního registru z oblasti původního vstupního tensoru. Jinak: náhodně zvol operaci pro redukci tensorů, nádodně na vstupním registru zvol subtensor.
- `Mutations.mutate_input_registers` - Náhodně zvol jeden ze vstupních registrů a náhodně do něj vyber nový index.
- `Mutations.mutate_output_register` - Zvol nový index do výstupního registru.
- `Mutations.mutate_operation` - Náhodně zvol novou operaci, pokud dojde k navýšení parity, náhodně zvol druhý operand.

3 Experimenty

V rámci projektu jsem provedl několik experimentů, jejichž cílem byla konfigurace hyperparametrů algoritmu tak, aby dosahoval co nejvyšší přesnosti u klasifikace objektů. Algoritmus byl vyhodnocován na grafické kartě **NVIDIA RTX 2060 (6GB)**, a to v několika nezávislých bězích po 10000 (případně i 20000 a více) generacích pro každou testovanou konfiguraci. K experimentování byla použita datová sada **MNIST**.

3.1 Růst a zkracování programu

První experimenty byly provedeny s hyperparametry algoritmu *p-grow* a *p-shrink*, které udávají pravděpodobnost, že nově vytvořenému programu bude náhodná instrukce přidána nebo odebrána. Celkem byly konfigarece testovány ve 3 variantách - oba parametry měly stejnou hodnotu (vždy vyšší z páru hodnot), parametr *p-grow* měl větší hodnotu, a naopak. Tyto hyperparametry byly testovány s těmito páry hodnot: $0.95, 0.05$ a $0.25, 0.05$. Každá konfigurace byla spuštěna v celkem 30 bězích, každý z nich po 10000 generacích.



Obr. 1: Fitness funkce na vyhodnocovacích datech po dokončení procesu trénování. Výsledky jsou rozdělené do sloupců podle fitness funkce použité během trénování. Hodnota křížové entropie je spočítána ještě předchozí implementací - hodnota je odečtena od konstanty 3, v dalších experimentech je již spočítána postupem uvedeným výše.

Zde je velice zajímavé, že nastavení parametru *p-grow* na vyšší hodnotu má pozitivní vliv na přesnost natrénovaného modelu. Je tomu pravděpodobně proto, že takové modely rychleji dosáhly maximálního počtu instrukcí a v tom případě je operátor růstu programu shora omezen, tudíž před přidáním nové instrukce musí být jiná náhodně odebrána, čímž se zajistí rychlejší evoluce. Tuto doměnku podrobíme testu v následujícím experimentu.

Dalším tenstem souboru konfigurací ověříme, zda-li má rychlost růstu programu vliv na jeho schopnost optimalizace. Parametru *p-shrink* je nastaven na 0, maximální počet instrukcí ponecháme z minulé konfigurace (stejně jako všechny ostatní parametry) a minimální počet instrukcí bude společně s parametrem *p-grow* podroben analýze. Testovací hypotézou je, že algoritmus benefituje z dosažení maximálního množství instrukcí co nejdříve, a to buďto kvůli rychlejší mutaci v případě dosažení maximální možné délky programu a nebo možností zaseknutí algoritmu na suboptimálním řešení s menším množstvím instrukcí, ze kterého se křížením ani mutací nedokáže vymanit k optimálnějšímu řešení, které vyžaduje více instrukcí. Experimenty proběhly s konfigurací parametru *p-grow* na 100% a 2% a parametru minimálního množství instrukcí - *min-i* na 48 a 16 (maximum instrukcí je vždy nastaveno na 48).

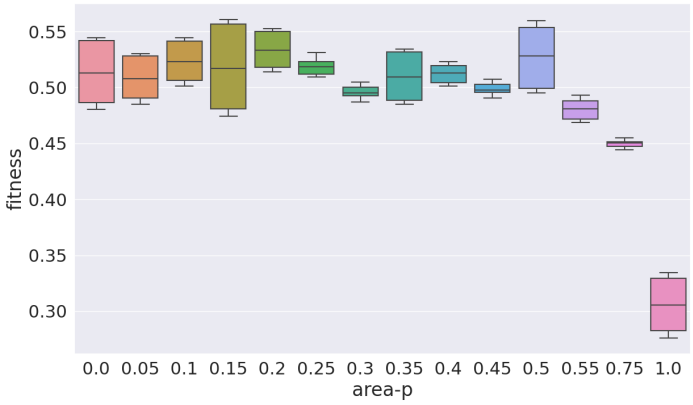
Průměrná vyhodnocená přesnost	Standardní odchylka	*min-i*	*p-grow*
51.82 %	3.52 %	16	2 %
59.17 %	3.04 %	16	100 %
54.68 %	3.78 %	48	2 %
60.04 %	2.97 %	48	100 %

Obr. 2: Vyhodnocení 2. experimentu ukazuje výrazný rozdíl v konfiguraci obou hyperparametrů.

Z výsledků v tabulce 2 je zřejmé, že parametr *p-grow* má relativně velký vliv na přesnost výsledného modelu. Postupný růst programu pak přesnost ovlivňuje spíše negativně, avšak nikoli v takovém měřítku jako parametr *p-grow*.

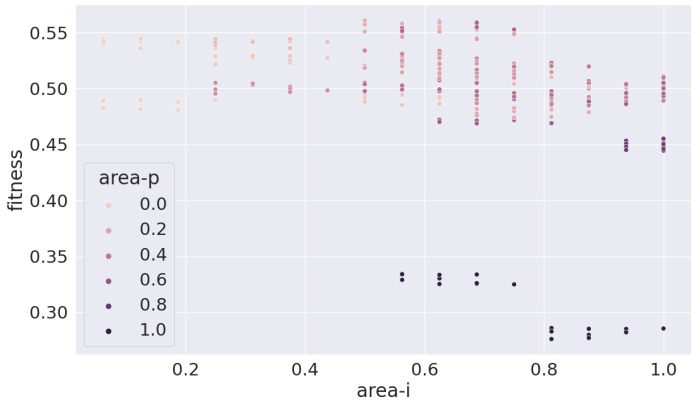
3.2 Instrukce zpracovávající tensory

Předmětem tohoto experimentu bylo zjistit vliv parametru *p-area*, který specifikuje pravděpodobnost, že náhodná instrukce bude vygenerována jako instrukce pracující s tensory, a v případě mutace určuje pravděpodobnost prohození z instrukce pracující s tensory na instrukce pracující pouze se skalárními hodnotami.
Z důvodu velkého množství experimentů a spojité povaze atributu *p-area* jsem zvolil trénování po dobu 10000 generací, avšak pouze ve 20 bězích pro konfiguraci. Výslekdy experimentů přesto poskytují zajímavé informace.



Obr. 3: Fitness funkce na vyhodnocovacích datech po dokončení procesu trénování.

Lze si povšimnout, že algoritmus trpí na snížení přesnosti nalezeného modelu, když parametr *area-p* překročí hodnotu 0.5. Zajímavou metrikou pro pochopení naměřených hodnot je pak poměr instrukcí pracujících s tensory ku všem instrukcím programu, viz graf 4.
Tato data pocházejí ze stejného experimentu. Lze si povšimnout, že ty nejpřesnější modely měly poměr počtu instrukcí pracujících s tensory k počtu všech instrukcí pragramu nad hodnotu 0.5 a pod 0.75. V tomto intervalu se také vyskytuje největší část nalezených řešení. Lze si podle barvy také všimnout, že přirozený výběr preferuje operace s tensory (pravděpodobně proto, že jedna instrukce obsahuje násobně více výpočetních možností než skalární operace) - poměr instrukcí pracujících s tensory je často větší, než hodnota parametru *area-p*. Výjimkou je však případ, kdy je nastavena hodnota *area-p* na 100%, v takovém případě je tomu spíše naopak, což je možné pouze proto, že během mutace programu může dojít ke změně instrukce na instrukci pracující pouze se skalárními hodnotami.

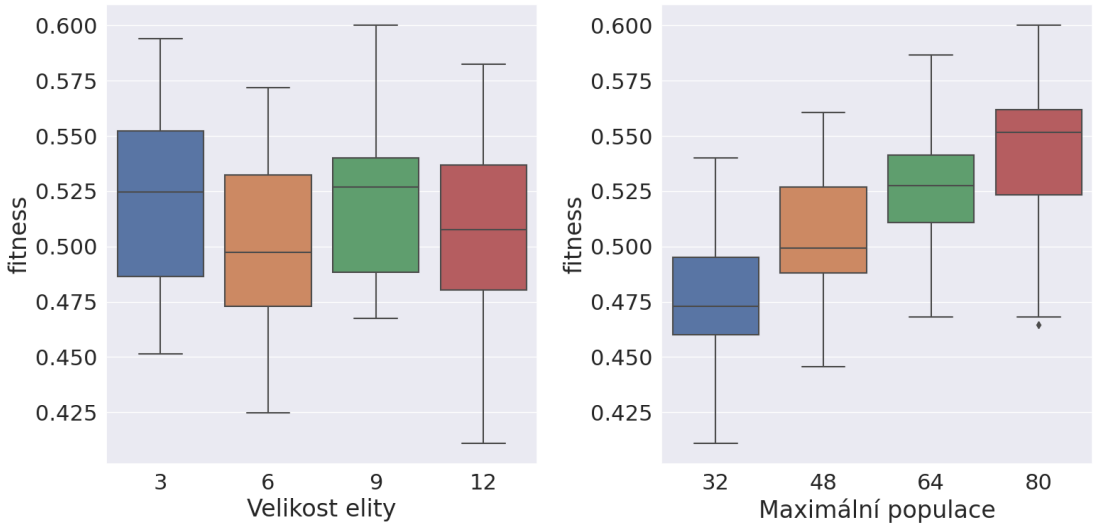


Obr. 4: Graf ukazuje závislosti mezi parametrem *area-p* a metrikami *fitness* a *area-i* (poměr počtu instrukcí pracujících s tensory k počtu všech instrukcí programu).

3.3 Populace a elita

Třetí experiment byl spuštěn pro získání informací, jak ovlivňuje přesnost řešení celková velikost populace a velikost elity (část populace s nejvyšší hodnotou fitness, která přežívá do další generace a z řad elit jsou náhodně vybírání rodiče nových potomků). Každá konfigurace byla spuštěna v celkem 30 bězích po 10000 generací.

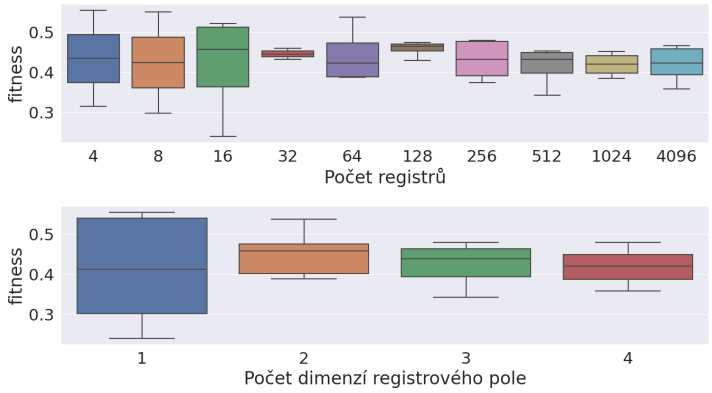
Zlepšení přesnosti modelu s rostoucí populací je zapříčiněno větší mírou prozkoumávání prostoru lineárních algoritmů. Cena výpočtu algoritmu však úměrně rosté také, proto nelze populaci zvětšovat donekonečna. Vliv velikosti elity na přesnost algoritmu je nejednoznačný, dle našeho experimentu je to však hodnota 9, kterou budu nadále používat jako fixní nastavení hyperparametru algoritmu. Výchozí nastavení populace ponechám i pro příští experimenty na 32 z důvodu omezeného výpočetního výkonu.



Obr. 5: Fitness funkce na vyhodnocovacích datech po dokončení procesu trénování.

3.4 Pracovní registry

Velice zajímavým hyperparametru *lineárního genetického algoritmu* je velikost pracovního registrového pole. V mé implementaci však hraje roli také tvar registrového pole, a to sice jen u instrukcí pracujícími s tensory, kde tvar hraje roli v náhodném "řezání"tensoru na subtensor-operand.

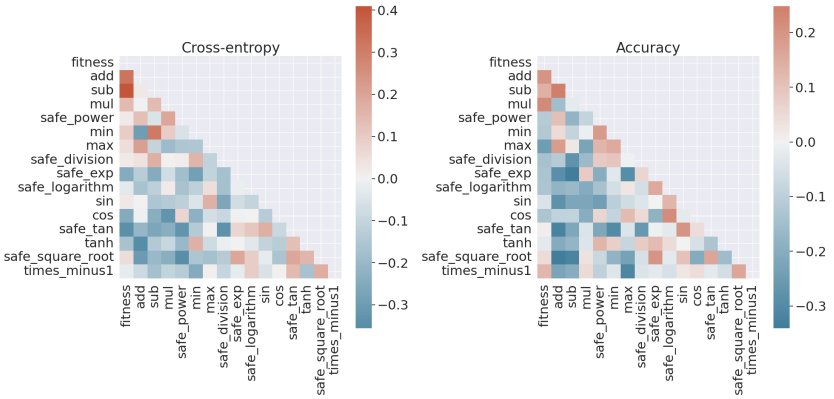


Obr. 6: Vyhodnocení modelů s různými velikostmi a tvary registrových polí.

U těchto výsledků je zřetelné, že registrové pole s jedinou dimenzí vykazuje oba extrémy - a to jak nejnižší, tak i nejvyšší přesnost modelu. To je zapříčiněno různými velikostmi registrových polí a statistika je snižována především díky registru velikosti 4, který neposkytuje příliš mnoho výpočetního prostoru. Nejúspěšnější bylo nastavení velikosti registrového pole na 8 registrů v 1 dimenzi, proto s tímto nastavením registrů budeme pokračovat i v dalších experimentech.

3.5 Instrukce

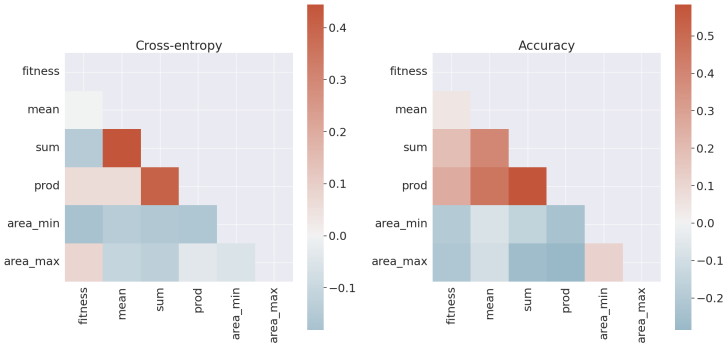
Pro tento experiment jsem agregoval všechny minulé běhy a jejich nejlepší modely. Modely byly následně vyhodnoceny nad validační částí datové sady a následně bylo u každého modelu zaznamenáno, v jakém poměru se v něm nacházejí různé instrukce. Následně byla mezi těmito hodnotama spočítána korelační matice.



Obr. 7: Korelační matice mezi vyhodnocenou přesností modelu a použitých unárních a binárních operací pracujících se skalárními operandy.

Z unárních operací jsem ponechal pouze operaci `times_minus1`, která vybranou hodnotu (nebo tensor) vynásobí -1. Z binárních operací se ukázaly efektivní základní operace sčítání, odčítání, násobení a bezpečné dělení. V případě použití upravené křížové entropie je pozitivní korelace i mezi operaci minimum a maximum.

Dalším předmětem analýzy byly instrukce pracující s tensory, jejichž korelace s přesností modelu.



Obr. 8: Korelační matice mezi vyhodnocenou přesností modelu a použitých operací procujících s tensory-operandy.

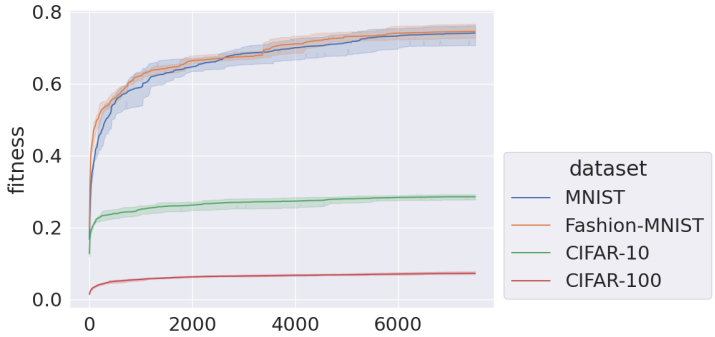
Na korelační matici mají konzistentně pozitivní korelaci pouze produkt a velmi slabou pozitivní korelaci u průměru. Při použití přesnosti jako fitness funkce má pozitivní korelaci také sumace. Tyto operace jsem tedy zvolil pro finální evaluaci modelu na více datasetech.

3.6 Finální vyhodnocení

Finální vyhodnocení konfigurace modelu probíhalo na 4 datových sadách: MNIST, FashionMNIST, CIFAR10 a CIFAR100. Konfigurace hyperparametrů byla následující:

- `area-p` = 10 %
- `grow-p` = 100 %
- `shrink-p` = 0 %
- `elite` = 9
- `max-instructions` = 256
- `register-shape` = (8,) (rozměry pracovního registru)
- `generations` = 7500

Z důvodu náročných výpočtů bylo vyhodnocení provedeno pouze v 5 bězích.



Obr. 9: Průběh trénování během posledního vyhodnocení.

Po vyhodnocení nejlepších modelů natrénovaných na každé datové sadě získáme tuto tabulku přesnosti modelů:

dataset	MNIST	FashinMNIST	CIFAR10	CIFAR100
nejlepší dosažená přesnost	73.07 %	74.89 %	26.43 %	5.1 %

4 Závěr

V rámci projektu byl úspěšně implementován lineární genetický algoritmus (LGA) pro klasifikaci objektů. Byly vytvořeny dva hlavní moduly: modul utils, který obsahuje třídu Dataset a funkce pro konfiguraci hyperparametrů algoritmu, a modul LGA, který obsahuje třídu LGA, třídu Program, třídu Instruction, operace pro instrukce a mutační procesy. Zajímavostí implementace je paralelní výpočet instrukcí na grafické kartě.

Hlavní cíle, které zahrnovaly statistické vyhodnocení algoritmů LGP s různými nastaveními parametrů a vyhodnocení chování algoritmu při klasifikaci vizuálních datových sad, byly dosaženy. Bylo provedeno testování na datových sadách MNIST, FashionMNIST, CIFAR10 a CIFAR100, které přineslo pozitivní výsledky a ukázalo schopnost algoritmu klasifikovat různé typy dat s lišící se úspěšnosti.

I přes urychlení výpočtů na GPU, celý běh algoritmu je vysoce výpočetně náročný, a proto nebylo provedeno tolik experimentů, jako bylo deklarováno v pojednání. Experimenty s konfigurací pravděpodobnosti křížení a mutace jsem bohužel chybně nastavil o 2 řády, proto tyto parametry společně s parametry určujícími proporci mutace nebyly podrobeny analýze.

Po vyhotovení tohoto projektu bych k analýze přistoupil jinak, a to buďto testováním na méně náročném datasetu, případně mnohem zevrubnější testování omezeného množství atributů (což bohužel nešlo bez získání alespoň minimálního přehledu o vlivu parametrů na výkon algoritmu).

Závěrem však hodnotím velice zajímavě zkušenost s implementací *LGA*, především kvůli vyšší abstakci chromozomu jako třídy `Program`.