Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

УПОТРЕБА НА ГЕНЕТСКИ АЛГОРИТМИ ЗА ОПТИМИЗАЦИЈА НА АРХИТЕКТУРАТА НА НЕВРОНСКА МРЕЖА

Семинарска работа по предметот Вештачка интелигенција

Изработено од:
Андреј Стерјев (211262)
Соња Петровска (211026)
Филип Самарџиски (211097)
Тамара Стојанова (211079)

Ментор:
Проф. д-р Андреа Кулаков

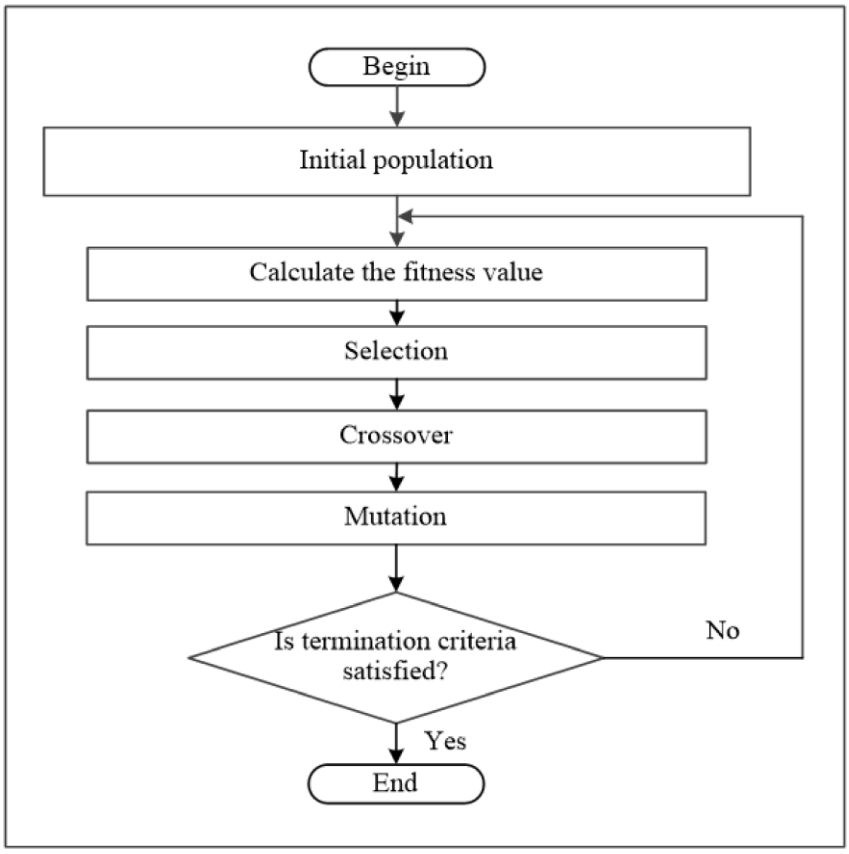
Содржина

Апстракт

Генетските алгоритми се една од најчесто употребуваните фамилии на оптимизациски алгоритми во областа на вештачката интелигенција. Вештачките невронски мрежи спаѓаат меѓу најмоќните модели кои може да се користат за секаков вид на машинско учење. Сепак, еден од главните предизвици при користењето на невронски мрежи за решавање на проблеми од машинско учење претставува несигурноста за тоа која е оптималната архитектура за невронската мрежа да враќа најдобри можни резултати, со најголема прецизност, без пренагодување. Ова е нешто што не може да се знае однапред. Во оваа семинарска работа, го истражуваме потенцијалот на комбинирањето на невронските мрежи со генетските алгоритми, преку користење на соодветна репрезентација на невронските мрежи како примероци од популација чии карактеристики генетскиот алгоритам ќе може да ги оптимизира.

Генетски алгоритми

Генетските алгоритми се класа на алгоритми која влече инспирација од природните еволутивни процеси кои се јавуваат во живиот свет. Тоа се алгоритми кои можат да решат разни видови на оптимизациски проблеми со висока веројатност за конвергенција до оптимално решение. Постојат многу паралели помеѓу генетски алгоритам и процесот на еволуција, кој генетскиот алгоритам го имитира. Чекорите на генетскиот алгоритам, како и различните начини на кој истиот може да се дизајнира ќе бидат објаснети накратко подолу.



Слика 1 - Дијаграм на типичен генетски алгоритам

Прв чекор кај генетски алгоритам е избирање на соодветна репрезентација на она што сакаме да го оптимизираме, а тоа типично е еднодимензионална податочна структура како листа. Потоа, треба да се одреди големината на првобитната генерација, односно бројот на единки со кои генетскиот алгоритам почнува да работи. Овие единки можат да бидат конкретно декларирани или случајно генерирани од вкупниот број на пермутации кој избраната репрезентација го дозволува. Во врска со големината на популацијата, преголеми вредности носат ризик алгоритмот да се извршува предолго, а премали

вредности ја намалуваат шансата за конвергенција. Следно, треба да се одреди колку најподобни единки од моменталната генерација ќе бидат издвојувани. Друг параметар кој исто така треба да се одреди е тоа колкава ќе биде големината на последователните генерации, која може, на пример, да остане иста како во првобитната популација, или постепено да се намалува за построго да се исфрлаат најнеподобните единки, итн.

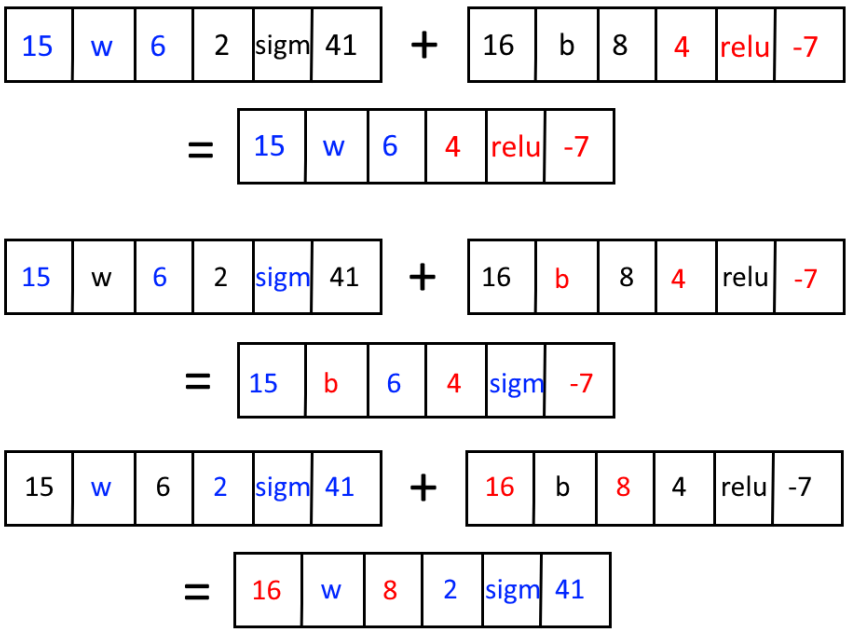
Следниот чекор е одредување како ќе се оценува кои единки се "подобни", односно воведување на таканаречената функција на подобност. Ова треба да биде некоја функција која преку нумеричка вредност ќе претставува колку добро решение таа единка претставува за проблемот кој се оптимизира. Функцијата на подобност во голема мера зависи од типот на проблем кој се разгледува, на пример:

- → за оптимизација на решавање на шаховска партија, проста функција на подобност може да биде разликата меѓу бројот на фигури на играчот кои се нападнати, и бројот на фигури на противникот кој играчот ги напаѓа со потенцијален потег од просторот на состојби;
- → за оптимизирање на параметрите на линеарна или полиномна регресија подобноста може да се изрази како реципрочна вредност од средноквадратната грешка;
- → за пронаоѓање на силна лозинка, функција на подобност може да биде бројот на последователни букви кои не се наоѓаат во речник како дел од познат збор итн.

За многу проблеми постојат повеќе различни функции на подобност што може да се изберат. Бидејќи генетските алгоритми се своевиден вид на алгоритми за пребарување на простор на состојби, а функцијата на подобност има улога на хевристичка функција, критериумот за избор на најдобра функција на подобност е ист како критериумот за најдобра хевристика – онаа која најпрецизно може да ја изрази вистинската подобност. За покомплицирани проблеми, покомплицирани функции на подобност работат подобро.

Следен чекор е одбирање на единки за вкрстување со нивно сортирање според излезот од функцијата на подобност за секој од нив. Може првите N најподобни единки да се одбираат директно, а може и, за да се внесе елемент на случајност, секоја единка да добие веројатност да биде избрана која соодветствува на нејзината подобност, така што најподобните ќе добијат најголеми веројатности. Ако сакаме да имаме минимална улога на случајност во процесот на избирање, но сепак случајноста да е присутна, можеме да ги квадрираме вредностите на подобност, па потоа да скалираме според квадратот на максималната подобност, итн.

Потоа, одбраните единки ги вкрстуваме по парови и од нив добиваме нови единки. Операторот за вкрстување (анг. *crossover*) исто така може да работи на различни начини. Во суштина, идејата е дека ако единките претставуваат секвенци на карактеристики (пандан на ДНК секвенци), во новата единка ќе се вкрстат дел од првата секвенца и дел од втората. Во најпрост случај, тоа може да бидат последователни членови од секој родител, но и не мора. Може да се земаат наизменично парните членови од едниот родител и непарните од другиот, или може случајно да се одбираат индексите на членовите, итн. Во однос на бројот на членови земени од секој родител, тоа повторно е работа на избор, може да е еднаков, може да не биде, може да биде фиксиран однапред или да се одредува пропорционално според разликата во подобност на двата родители. Најдобрите параметри за работа најчесто се откриваат експериментално.



Слика 2 - Различни можности за вкрстување на единки

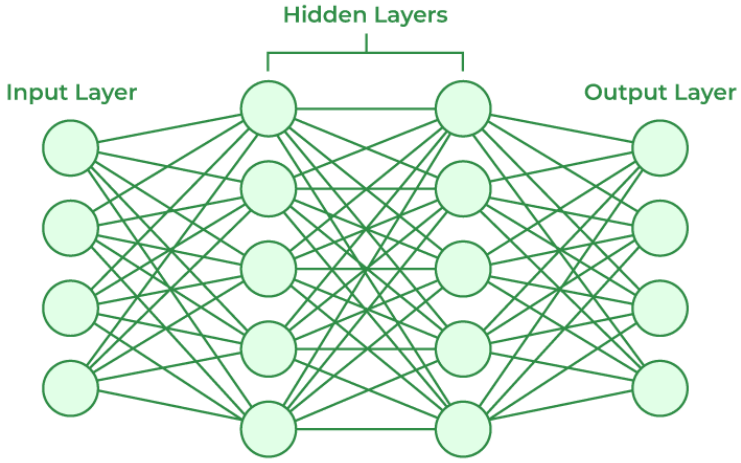
Наредниот чекор е мутација на децата. Во листата на карактеристики на секое дете, на секој индекс поединечно може вредноста да "мутира" односно да се промени со некоја веројатност. Новата мутирана вредност може да биде случајна, да биде зголемена или намалена од првобитната вредност, да биде наредната/претходната од листа предетерминирани дозволени вредности, итн. Суштината на мутациите е дека тие се важен механизам за генетскиот алгоритам подобро да генерализира, бидејќи го намалуваат ризикот алгоритмот да заглави во локално оптимално решение, со воведувањето на случајните промени во единките. Од друга страна, како компромис мутациите малку го успоруваат конвергирањето кон глобалниот оптимум.

Откако ќе се добијат децата единки после вкрстувањето, потребно е да се пресмета нивната подобност, па потоа да се сортираат повторни сите единки, вклучувајќи ги и децата. Од оваа нова популација од родители и деца, ги избираме првите K најподобни, и останатите ги отфрламе, ако K е потребната големина на наредната генерација.

Генетските алгоритми се итеративни, па така овој циклус се повторува додека или се пронајде решение кое ги задоволува претходно зададените критериуми, или се достигне некој максимален број на итерации. На овој начин се оптимизираат проблеми од секаков карактер.

Невронски мрежи

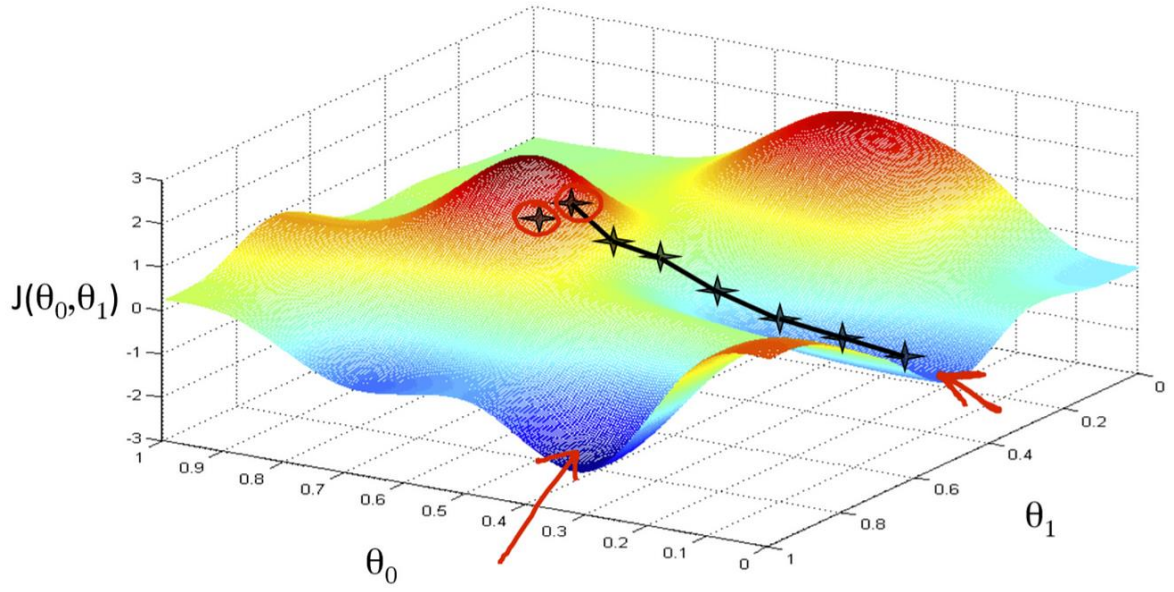
Вештачките невронски мрежи се срцето на модерното машинско учење. Базирани се на структурата на човечкиот мозок, во кој информации се споделуваат преку многу милиони неврони со висок степен на меѓусебна поврзаност. Дизајнирани се да работат на сличен принцип како вистинскиот мозок, но на упростен начин. “Невроните” во вештачка невронска мрежа се јазли во кои влегуваат информации од претходниот слој и од кои излегуваат информации и патуваат кон наредниот слој. Во вистински мозок, информациите патуваат преку дендритите и синапсите како електрохемиски сигнал овозможен од невротрансмитери како допамин, серотонин, адреналин итн. Во вештачките невронски мрежи, “сообраќајот” на информации во архитектурата на мрежата се контролира преку тежините на врските меѓу невроните и активациските функции. Имено, секоја врска од некој неврон до друг неврон има соодветна тежина, која симболизира колку е “важен” излезот од претходниот неврон како влез во наредниот. Сите влезови кои ги прима еден неврон од претходниот слој, заедно со нивните соодветни тежини, се комбинираат на некој начин од активациската функција, која може да биде тежинска сума, сигмоидална функција, итн., но кај различни невронски мрежи каде распоредот на влезовите е важен може да биде пермутациски варијантна функција како конволуција. Невронската мрежа е составена од почетен слој, кој го претставува влезот на моделот, скриени слоеви низ кои влезот се трансформира, и излезен слој, кој треба да даде одговор на задачата за којашто се тренира мрежата, како класификација, регресија, кластерирање итн.



Слика 3 - Состав на едноставна невронска мрежа

Откако ќе се добие некоја вредност во излезниот слој, треба да се евалуира дали е добиен посакуваниот излез (на пример за класификација на некој примерок), или колку е блиску излезот до вистинската вредност (кај регресиони проблеми). Откако ќе се добие

излезот за секој од примероците кои се на располагање, може да се пресмета колку е добар моделот преку функцијата на грешка. Постојат различни функции на грешка зависно од тоа што треба невронската мрежа да прави. Вредностите на овие грешки се пропагираат наназад (анг. backpropagation) низ слоевите на мрежата преку пресметување на парцијални изводи. Вака се пресметува градиентот на моменталната грешка што ја има мрежата и преку оптимизациски алгоритам наречен спуштање по градиент (анг. gradient descent) се ажурираат внатрешните тежини помеѓу невроните, што влијае да се промени излезот во наредната итерација. На овој начин, мрежата добива ограничена адаптабилност на проблемот, и може да се менува самата себеси за да го реши тој проблем.



Слика 4 - Спуштање по градиент. Координатите (параметрите) каде што се наоѓа глобалниот минимум на функцијата на грешка се оптимално решение.

Сепак, иако вештачките невронски мрежи ја поседуваат оваа способност да си ги ажурираат тежините, тоа е многу ограничена адаптабилност во споредба со онаа што ја има вистинскиот мозок, кој има својство на невропластичност (анг. neuroplasticity). Биолошкиот мозок, кога е соочен со нови и непознати стимули, може да ја менува својата структура со преповрзување на своите неврони, а дури може и одредени региони од мозокот да се пренаменуваат од една во друга функција, посебно во случаи на повреди. Вештачките невронски мрежи сами по себе ја немаат оваа можност. Еднаш иницијализирана невронска мрежа, со однапред зададен број на слоеви, број на неврони во секој слој, начин на поврзаност, активациски функции, итн. не може самата да се оптимизира себеси преку додавање/вадење на слоеви, или додавање/вадење на неврони во слоевите, менување на активациските функции, итн. Иако постојат механизми за секоја невронска мрежа да може евентуално да конвергира до глобалниот оптимум, доколку би се дознала оптималната архитектура за таа мрежа, конвергенцијата би се случувала многу побрзо и поефикасно. Оттука доаѓа идејата за вкрстување на концептот на генетски

алгоритам и невронска мрежа, односно за користење на генетски алгоритам за оптимизација на самата архитектура на невронската мрежа преку соодветна репрезентација на нејзините параметри како генетски примерок.

Комбиниран пристап

Како што беше спомнато погоре, идејата е освен оптимизацијата на тежините внатре во невронската мрежа, што веќе може да се постигне со алгоритмот на спуштање по градиент, исто така да може да се оптимизираат разни други параметри на невронската мрежа. За да се користат генетски алгоритми за оваа намена, главниот предизвик е да се пронајде соодветна репрезентација на невронските мрежи како генетски примероци. Оваа репрезентација треба да дава слика за карактеристиките на мрежата и треба да се состои од сè што ќе сакаме да се менува во текот на генетскиот алгоритам. Каква ќе биде формата на оваа репрезентација зависи од тоа врз кои параметри сакаме да имаме контрола, како и кој е типот на невронските мрежи и како се креираат. Различни библиотеки во Python ни даваат различно ниво и начин на контрола на конфигурирањето на мрежите. Меѓутоа, во секој случај, најлогично е репрезентацијата на невронската мрежа да биде листа со сите нејзини параметри. Овие параметри може да бидат:

- → Број на скриени слоеви во мрежата
- → Типови на скриени слоеви во мрежата
- → Број на неврони по слој
- → Активациски функции за секој слој
- → Степен на поврзаност на невроните во слоевите
- → Тежини на врските, со различен елемент од листата за секоја тежина
- → Тип на секој скриен слој индивидуално, ако сакаме поголема контрола
- → Тип на оптимизатор за gradient descent алгоритмот
- → Големина на batch
- → Активациска функција на излезен слој

и други.

На пример, ако сакаме со генетски примерок да претставиме мрежа изградена со библиотеката Keras, можеме по договор да речеме дека слоевите се претставуваат во тројки од [тип на слој, број на неврони, активациска функција], каде може активациската функција да се испушти ако се користи дифолтна вредност, освен последниот член, кој го означува типот на оптимизатор кој се користи за тренирање на тежините. Пример таква мрежа би била претставена со листата подолу:

Dense	64	relu	Dense	32	relu	Dense	32	sigmoid	Dense	1	adagrad

Слика 5 - Еден можен примерок од невронска мрежа за користење во генетски алгоритам

Репрезентациите на мрежите можат да бидат едноставни или комплицирани до степен што го одредуваме ние. На пример, ако сметаме дека не ни е потребно за секој слој експлицитно да го наведуваме бројот на неврони и активациските функции, можеме да ја упростиме репрезентацијата во форма [број на скриени слоеви, тип на скриени слоеви, број на неврони во скриени слоеви, активациска функција во скриени слоеви, број на неврони во излезен слој, тип на излезен слој, активациска функција на излезен слој, оптимизатор] и така би добиле пократки репрезентации за мрежите, особено ако се работи за длабоки мрежи со над 5 слоја, но ќе изгубиме дел од модуларноста на популацијата. Пример за таква репрезентација е даден подолу:

6	Dense	16	relu	1	Dense	softmax	adam

Слика 6 - Друг можен примерок од невронска мрежа за користење во генетски алгоритам

Се разбира, дури и ако се ограничиме на само една библиотека и еден тип на невронски мрежи, бројот на различни начини на кои можеме да се договориме да ги претставуваме мрежите е огромен, и начинот го бираме ние зависно од нашите потреби. Освен тоа, самиот избор на библиотека одредува кое ниво на пристап имаме до архитектурата на мрежата. Ако во Keras имаме можност лесно да ја манипулираме поврзаноста на слоевите, активациските функции итн., во мрежа испрограмирана рачно со помош само на NumPy би имале пристап и до тежините на секоја врска посебно, кои би се претворале од матрици во вектори за потребата на генетскиот алгоритам, и така би имале оптимизација директно врз тежините на врските без да зависиме од надворешна библиотека, односно би користеле генетски алгоритам за она за кое вообичаено би користеле спуштање по градиент. Пример за таква репрезентација на мрежа би бил:

3	12	7	5	8	122	0.3	6	0.9	1	0	77	9.4	5	1	21	2.2	0	0	15	1.5	4

Слика 7 - Примерок кој содржи информации само за тежините на врските

Изобилството на избор за каква репрезентација ќе одбереме станува уште поголемо кога ќе разгледуваме различни типови на невронски мрежи кои ни дозволуваат да ги манипулираме на различен начин, на пример кај конволуциски невронски мрежи можеме да ги додадеме во репрезентацијата големините на филтрите, и слично можеме да правиме кај било кои специфични видови на мрежи (LSTM, рекурентни, граф невронски мрежи, итн.).

Откако ќе ги добиеме репрезентациите, најголемиот дел од генетскиот алгоритам се одвива прилично стандардно. Треба да се пресмета подобноста на нашата моментална генерација (може да биде нешто едноставно како прецизноста на мрежата, среднoквадратна грешка за регресија итн. или некоја пософистицирана функција која

вклучува и фактори како време до конвергенција, број на внатрешни параметри во мрежата, во смисла да се преферираат мрежи кои се поедноставни, итн.)

Потоа треба да се одберат единките за вкрстување. Овде треба да се внимава на дизајнот на функцијата за вкрстување ако нашите листи со карактеристики на мрежите имаат некоја покомплексна структура. На пример, ако имаме листи кои претставуваат само листи од тежини на врските, можеме слободно да ги вкрстуваме како сакаме. Меѓутоа, ако нашите листи се како во примерот со подредени тројки, треба да внимаваме да не ги "сечеме" тројките при вкрстувањето и по грешка да добиеме примерок со невалиден формат.

Кај мутациите исто така има мало ограничување бидејќи во повеќето вакви листи ќе имаме членови кои претставуваат категориски променливи, како на пример функциите за активација или оптимизаторите, и нив не можеме да ги мутираме со генерирање на случаен број, туку мора да им доделиме друга вредност од предефинираните дозволени вредности.

Меѓутоа, освен овие мали детали, генетскиот алгоритам се одвива многу слично како и во општ случај. Бидејќи промената на генерациите се изведува на тој начин што цело време се креираат нови мрежи со вкрстени карактеристики (не може да се вадат, додаваат и менуваат работи од постоечки модел), кои треба сите индивидуално да се компајлираат и тренираат, другиот предизвик во овој процес е потенцијалното долго време на извршување. На крај, откако ќе ги добиеме новите најдобри N модели, старите се бришат од листата што ја чуваме.

Конкретна имплементација во код

Во оваа секција подетално ќе ги истражиме и објасниме техниките коишто ги користиме во нашата Python апликација за оптимизација на параметрите на невронската мрежа преку примена на генетските алгоритми.

Првата стапка во користењето на апликацијата започнува со отворање на главниот дел на програмата, односно фајлот main.py.

Овде корисникот прво треба да внесе број на генерации, односно број на пати на еволуција на популацијата. Потоа тој треба да ја внесе и посакуваната големина на секоја од популациите, односно бројот на невронски мрежи кои ќе бидат тренирани со податоците. Истите подоцна ќе треба да класифицираат нови непознати податоци од тестирачко множество, со што на крај ќе се пресмета нивната точност. Следниот чекор е избирање на податочното множество, а за тоа има две опции. Првата опција е податочното множество MNIST коешто се состои од 60.000 слики за тренирање и 10.000 слики за тестирање од десетте арапски цифри напишани рачно. Ова популарно податочно множество е користено за различни системи за дигитално процесирање на слика и машинско учење. Втората опција е податочното множество CIFAR10, кое е исто така големо податочно множество што се користи за машинско учење и компјутерска визија. Се состои од 50.000 слики за тренирање и 10.000 слики за тестирање, кои се распоредени во 10 различни класи како

автомобили, мачиња, кучиња, птици и слично. Соодветно, може да се изберат и број на слики со коишто ќе се тренираат и тестираат моделите.

```
if __name__ == '__main__':
    generations = int(input("Enter the number of generations, i.e. the number
of times to evolve the population\n"))
    population = int(input("Enter the size of the population, i.e. the number
of networks in each generation\n"))
    dataset = input("Enter the dataset [Options: 'mnist', 'cifar10']\n")
    size_of_train = int(input("Enter the amount of pictures to train on
[Defaults: MNIST-60000, CIFAR10-50000]\n"))
    size_of_test = int(input("Enter the amount of pictures to test on
[Defaults: MNIST-10000, CIFAR10-10000]\n"))
```

Потоа, во самиот код на нашата програма ние ги дефинираме можните опции, т.е. предефинирани вредности за избор на четирите параметри на невронската мрежа кои ги избравме за оптимизација со помош на генетскиот алгоритам, а тие се:

- број на неврони во даден скриен слој
- број на скриени слоеви во мрежата
- тип на активациска функција во секој од скриените слоеви
- тип на оптимизатор.

Овие параметри и нивните можни вредности ги дефинираме со користење на речник.

```
nn_param_choices = {
    'nb_neurons': [64, 128, 256, 512, 1024],
    'nb_layers': [1, 2, 3, 4],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'optimizer': ['lbfgs', 'adam', 'sgd'],
}
```

Потоа следува повик кон функцијата generate(), каде што како нејзини параметри се предаваат досега споменатите: број на генерации, големина на популација, податочно множество, речник од параметри со листа од можни вредности и големина на тренирачко и тестирачко множество.

```
print("***Evolving %d generations with population %d***" %
      (generations, population))

generate(generations, population, nn_param_choices, dataset, size_of_train,
size_of_test)
```

Во оваа функција се случуваат најголем дел од работите во програмата со повикување на разни функции од други класи. Најпрво се креира објект од класата Optimizer во кој се чува речникот од параметри и нивни можни вредности. Оваа класа всушност ја претставува имплементацијата на генетскиот алгоритам. Освен овој речник, класата содржи уште неколку атрибути, а тие се: retain – процент на популацијата што ќе преживее во следната генерација, random_select – веројатност за одредена незадоволителна невронска мрежа да остане во популацијата, и mutate_chance – веројатност дека одредена невронска мрежа ќе мутира во идната генерација. Потоа со функцијата create_population() од истата класа, со

праќање на претходно зададената големина, ќе биде креирана популација од толкав број на невронски мрежи кои имаат случајно избрани параметри од речникот на параметри и вредности, а исто така ќе биде вратен и речник кои ќе ги содржи досега креираните невронски мрежи односно нивните параметри, а вредноста ќе биде точноста (на почетокот нула).

```
def generate(generations, population, nn_param_choices, dataset,
size_of_train, size_of_test):
    optimizer = Optimizer(nn_param_choices)
    networks, networks_dict = optimizer.create_population(population)

class Optimizer:

    def __init__(self, nn_param_choices, retain=0.4,
                 random_select=0.1, mutate_chance=0.2):
        self.mutate_chance = mutate_chance
        self.random_select = random_select
        self.retain = retain
        self.nn_param_choices = nn_param_choices

    def create_population(self, count):
        pop = []
        pop_dict = {}
        for _ in range(0, count):
                # Create a random network.
                network = Network(self.nn_param_choices)
                network.create_random()
                # Add the network to our population.
                pop.append(network)
                pop_dict[str(network.network)] = network.accuracy
        return pop, pop_dict
```

Овие невронски мрежи се всушност објекти од класата Network, која се состои од атрибутите точност, речник од можни опции за параметри, како и речник за конкретните параметри за оваа невронска мрежа кои ќе бидат избрани случајно со функцијата create_random().

```
class Network:
    def __init__(self, nn_param_choices=None):
        self.accuracy = 0.
        self.nn_param_choices = nn_param_choices
        self.network = {}  # (dic): represents MLP network parameters

    def create_random(self):
        for key in self.nn_param_choices:
            self.network[key] = random.choice(self.nn_param_choices[key])
```

Следно што прави функцијата generate() е итерирање низ бројот на генерации којшто го зададовме, каде што всушност се случува самата еволуција на популациите низ различните генерации.

```
# Evolve the generation.
for i in range(generations):
    print("***Doing generation %d of %d***" %
          (i + 1, generations))

    # Train and get accuracy for networks.
    train_networks(networks, networks_dict, dataset, size_of_train,
size_of_test)
```

Во суштина, тука се случува тренирањето на невронските мрежи по популација со функцијата train_networks(). Се итерира низ сите невронски мрежи од таа популација и со функцијата train(), која пак ја повикува функцијата train_and_find_accuracy(), на крај се случува самото тренирање над податочното множество пратено во функцијата. Важно е дека исти мрежите се тренираат само еднаш, а ова го регулираме со помош на речникот networks_dict кој го праќаме во функцијата. Во овој речник, точноста на соодветната мрежа ја поставуваме како нејзина вредност, и кон тренирање на мрежата пристапуваме само доколку вредноста за клучот од параметрите на таа мрежа е еднаков на нула..

```
def train_networks(networks, networks_dict, dataset, size_of_train,
size_of_test):
    pbar = tqdm(total=len(networks))
    for network in networks:
        if networks_dict[str(network.network)] == 0:
            network.train(dataset, size_of_train, size_of_test)
            networks_dict[str(network.network)] = network.accuracy
        else:
            network.accuracy = networks_dict[str(network.network)]
        pbar.update(1)
    pbar.close()
```

```
def train(self, dataset, size_of_train, size_of_test):
    if self.accuracy == 0.:
        self.accuracy = train_and_find_accuracy(self.network, dataset,
size_of_train, size_of_test)
```

Во функцијата train_and_find_accuracy() прво се проверува за кое податочно множество станува збор, затоа што двете податочни множества кои ги имаме на располагање имаат различен метод на пристап до податоците, како и различна големина и форма.

```
def train_and_find_accuracy(network, dataset, size_of_train, size_of_test):
    if dataset == 'cifar10':
        batch_size, x_train, x_test, y_train, y_test =
get_cifar10(size_of_train, size_of_test)
    elif dataset == 'mnist':
        batch_size, x_train, x_test, y_train, y_test =
get_mnist(size_of_train, size_of_test)
```

Двата методи get_mnist() и get_cifar10() се прилично слични, со мали разлики. Се креира променлива за бројот на класите што ги има во двете податочни множества, односно 10, и

променлива за големина на batch. Batch всушност претставува еден вид на парче што определува колку податоци(примероци) од тренинг множеството се користат при една епоха од тренирањето на таа мрежа. Потоа, и за двете податочни множества се користи библиотеката Keras за нивно вчитување, со помош на cifar10.load_data() и mnist.load_data(). Откако податоците се вчитуваат, тие се делат на тренирачко и тестирачко множество, односно x_train и x_test се сликите од податочните множества, додека пак y_train и y_test се соодветните класи. Следно, потребно е податоците да се подготват и обработат со цел нивно полесно процесирање од страна на компјутерот. Обликот на тренирачките и тестирачките слики од CIFAR10 се преобразува од (50000, 32, 32, 3) во (50000, 3072) и од (10000, 32, 32, 3) во (10000, 3072) соодветно, каде што бројот 3072 се добива со множење на 32, 32 и 3, кои претставуваат двете димензии на секоја слика и пикселите за секоја димензија (32x32), како и третата димензија која го претставува бројот на канали за секоја слика (3-Red, Green, Blue). Во методот за MNIST се случуваат истите работи само со различна големина, односно бидејќи во MNIST секоја слика се состои од само 2 димензии и секоја слика е со 28x28 пиксели, па со нивно множење се добива бројката 784, за да се претвори обликот од (60000, 28, 28) во (60000, 784) и од (10000, 28, 28) во (10000, 784) за тренирачкото и тестирачкото множество соодветно. Со примена на reshape() од тридимензионални и дводимензионални слики добиваме еднодимензионални слики кои се погодни за работа во невронски мрежи. Потоа, transform_to_float_and_divide(x_train, x_test) враќа нормализирани вредности на пикселите во опсег [0,1] во типот float. Последниот чекор во оваа фукнција е претворањето на класите (целите броеви од MNIST или категориите од CIFAR10) во бинарни матрици со користење на функцијата to_categorical() од библиотеката Keras. На пример, класата 3 од MNIST ќе се конвертира во [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] затоа што има 10 класи. Функцијата ги враќа сите споменати параметри batch_size, x_train, x_test, y_train, y_test.

```
def transform_to_float_and_divide(x_train, x_test):
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255
    return x_train, x_test
```

```
def get_cifar10(size_of_train, size_of_test):
    # Set defaults.
    nb_classes = 10
    batch_size = 64
    # Get the data.
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()
    x_train, y_train, x_test, y_test = get_desired_amount_of_data(x_train,
y_train, x_test, y_test, size_of_train,

size_of_test)
    x_train = x_train.reshape(size_of_train, 3072)
    x_test = x_test.reshape(size_of_test, 3072)
    x_train, x_test = transform_to_float_and_divide(x_train, x_test)
    # convert class vectors to binary class matrices
    y_train = to_categorical(y_train, nb_classes)
    y_test = to_categorical(y_test, nb_classes)
    return batch_size, x_train, x_test, y_train, y_test
```

```
def get_mnist(size_of_train, size_of_test):
    # Set defaults.
    nb_classes = 10
    batch_size = 128
    # Get the data.
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, y_train, x_test, y_test = get_desired_amount_of_data(x_train,
y_train, x_test, y_test, size_of_train,

size_of_test)
    x_train = x_train.reshape(size_of_train, 784)
    x_test = x_test.reshape(size_of_test, 784)
    x_train, x_test = transform_to_float_and_divide(x_train, x_test)
    # convert class vectors to binary class matrices
    y_train = to_categorical(y_train, nb_classes)
    y_test = to_categorical(y_test, nb_classes)
    return batch_size, x_train, x_test, y_train, y_test
```

Понатаму во функцијата train_and_find_accuracy() следи повикот на функцијата compile_model().

```
model = compile_model(network, batch_size)
```

Тука се извлекуваат параметрите на соодветната невронската мрежа од речникот што погоре го спомнавме. Се креира торка каде се излистува бројот на неврони во секој од слоевите, што ни дава можност за уште поголема контрола на архитектурата на мрежата. На крајот, со сите овие параметри се креира Multi-Layer Perceptron (MLP) модел, којшто функцијата го враќа.

```
def compile_model(network, batch_size):
    # Get our network parameters.
    nb_layers = network['nb_layers']
    nb_neurons = network['nb_neurons']
    activation = network['activation']
    optimizer = network['optimizer']
    list_layers = []
    for i in range(nb_layers):
        list_layers.append(nb_neurons)
    layers_with_neurons = tuple(list_layers)
    model = MLPClassifier(hidden_layer_sizes=layers_with_neurons,
activation=activation, solver=optimizer,
                          max_iter=10000, verbose=0, batch_size=batch_size,
random_state=0)
    return model
```

Составениот модел се тренира со користење на податоците за обука (x_train и y_train), добиени од чекорот за вчитување на податоците. Методот fit се користи за обука на моделот. Истрениранот модел ги прави своите предвидувања над тестирачкото множество, и врз основа на стекнатите податоци се пресметува неговата точност, која самата функција train_and_find_accuracy() ја враќа како резултат. Оваа вредност понатаму се зачувува во атрибутот accuracy на соодветниот објект од класата Network.

```
model = compile_model(network, batch_size)
model.fit(x_train, y_train)
predictions = model.predict(x_test)
correctly_predicted = 0
for true, pred in zip(y_test, predictions):
    pred = np.array(pred, dtype=float)
    element_wise_comparison = pred == true
    if np.all(element_wise_comparison):
        correctly_predicted += 1
score = correctly_predicted / len(y_test)
return score
```

Во остатокот од функцијата generate(), се пресметува и печати средната вредност за точноста на невронските мрежи од една генерација. Сè додека не се работи за последната генерација, популацијата понатамошно еволуира. Откако ќе се обработат сите генерации, конечната популација се подредува врз основа на точноста по опаѓачки редослед и топ петте мрежи со најголема точност се печатат со помош на функцијата print_networks. Бидејќи во алгоритмот допуштаме мрежи со исти параметри да бидат генерирани повеќе пати, овие топ пет мрежи ги извлекуваме од речникот networks_dict, каде што ист клуч не може да се појави повеќе од еднаш, со што спречуваме мрежа со иста точност повеќе пати да биде испечатена од страна на функцијата.

```
    average_accuracy = get_average_accuracy(networks)
    # Print out the average accuracy each generation.
    print("Generation average: %.2f%%" % (average_accuracy * 100))
    print('-' * 80)
    # Evolve, except on the last iteration.
    if i != generations - 1:
        # Do the evolution.
        networks, networks_dict = optimizer.evolve(networks, networks_dict)
# Sort our final population.
sorted_dicts = sorted(networks_dict.items(), key=lambda item: item[1],
reverse=True)
# Print out the top 5 networks.
print_networks(sorted_dicts[:5])
```

Што се однесува до еволуцијата на популацијата, во овој процес, најпрво се пресметува резултат од функцијата на подобност за секоја мрежа во популацијата – тоа е всушност самата нејзина точност, и овој резултат се става во пар со самата мрежа.

```
def fitness(network):
    return network.accuracy
```

```
def evolve(self, pop, pop_dict):
    # Get scores for each network.
    graded = [(self.fitness(network), network) for network in pop]
```

Потоа, се извршува сортирање во опаѓачки редослед на овие резултати, со цел да се одреди кои мрежи се најдобри. Се пресметува бројот на мрежи што треба да се задржат во следната генерација, кој е дефиниран од атрибутот retain. Сите мрежи коишто ќе бидат

задржани при оваа елиминација се всушност родителите, и тие се задржуваат за следната генерација. Освен ова, по случаен избор може да бидат задржани уште неколку мрежи, и тие да бидат додадени во избраните родители. Останатите празни места во популацијата се полнат со деца, односно нови единки создадени преку процесот на вкрстување.

```
# Sort on the scores.
graded = [x[1] for x in sorted(graded, key=lambda x: x[0], reverse=True)]
# Get the number we want to keep for the next gen.
retain_length = int(len(graded) * self.retain)
# The parents are every network we want to keep.
parents = graded[:retain_length]
# For those we aren't keeping, randomly keep some anyway.
for individual in graded[retain_length:]:
    if self.random_select > random.random():
        parents.append(individual)
# Now find out how many spots we have left to fill.
parents_length = len(parents)
desired_length = len(pop) - parents_length
children = []
# Add children, which are bred from two remaining networks.
while len(children) < desired_length:
    # Get a random mom and dad.
    male = random.randint(0, parents_length - 1)
    female = random.randint(0, parents_length - 1)
    # Assuming they aren't the same network...
    if male != female:
        male = parents[male]
        female = parents[female]
        # Breed them.
        babies = self.breed(male, female)
        # Add the children one at a time.
        for baby in babies:
            # Don't grow larger than desired length.
            if len(children) < desired_length:
                children.append(baby)
                pop_dict[str(baby.network)] = baby.accuracy
```

Во нашиот алгоритам, се бираат двајца различни родители по случаен избор, кои формираат деца со помош на функцијата breed(). Тука, всушност секое дете претставува нова невронска мрежа, а нивната архитектура се одредува на случаен начин од вредностите на родителите за секој параметар соодветно.

```
def breed(self, mother, father):
    children = []
    for _ in range(2):
        child = {}
        # Loop through the parameters and pick params for the kid.
        for param in self.nn_param_choices:
            child[param] = random.choice(
                [mother.network[param], father.network[param]]
            )
        # Now create a network object.
        network = Network(self.nn_param_choices)
        network.create_set(child)
        # Randomly mutate some of the children.
```

```
        if self.mutate_chance > random.random():
            network = self.mutate(network)

        children.append(network)
    return children
```

Секако, по случаен избор може да се случи некоја од единките да мутира. Мутацијата се врши така што избираме еден од параметрите на мрежата, и неговата вредност ја поставуваме на еден од соодветните дозволени избори за истата.

```
def mutate(self, network):
    # Choose a random key.
    mutation = random.choice(list(self.nn_param_choices.keys()))
    # Mutate one of the params.
    network.network[mutation] =
random.choice(self.nn_param_choices[mutation])
    return network
```

На овој начин се формираат единките деца, и тие се враќаат како резултат од breed() функцијата. На родителите потоа ги надодаваме создадените деца, и така популацијата еволуира.

```
    parents.extend(children)
    return parents
```

Резултати

Резултати од програмата со генетски алгоритам

Извршувањето на програмата којашто детално ја објаснивме траеше околу 14,715 часа. Се добива излез, кој дава податоци за секоја генерација посебно, како просечна точност, време на извршување итн., а на крајот се печатат и топ петте невронски мрежи, сортирани според точноста, до кои генетскиот алгоритам дошол со генерирање на единки по случаен пат и нивно вкрстување. Како што напоменавме, параметрите се внесуваат на почетокот, односно тоа се бројот на генерации и големината на популацијата во секоја од нив, изборот за податочното множество, и количината на тренирачки и тестирачки слики кои ќе се користат од наведеното податочно множество. Секако, овие параметри многу влијаат на времето на извршување на програмата. Ние како некоја „оптимална“ вредност за бројот на генерации го избравме бројот десет, а за бројот на единки во секоја генерација избравме дваесет. Ова значи дека ќе бидат креирани десет генерации од по дваесет единки т.е. невронски мрежи, односно 200 мрежи кои ќе бидат тренирани на множество од 60.000 слики, а тестирани на 10.000 слики. Токму на ова се должеше долгото време на извршување на нашата програма.

Пробавме да ја извршиме програмата и со намалени вредности за секој од параметрите, но тоа не секогаш резултираше со поголема брзина на извршување. На пример, забележавме дека едно од ваквите извршувања беше многу споро, а потоа сфативме дека

ова се должело на малиот број на единки во една генерација. Односно, во еден сегмент од кодот, кога сакаме да вкрстуваме единки, по случаен избор избираме две единки од генерацијата, кои треба да бидат различни. Но, при популација со мал број на единки, многу е поверојатно да се избере истата единка по случаен избор. Исто така, и просечните точности на генерациите и петте најголемите вредности за точноста при тие извршувања беа значително помали. Поради ова, избравме поголеми бројки за генерации и големина на популации, но се обидовме кодот да го рефакторираме и оптимизираме на различни начини така што, на пример, нема да има потреба невронски мрежи со исти параметри да бидат два пати тренирани, итн.

Во последното извршување на програмата со користење на генетски алгоритам, го добивме следниот излез, кој исто така може да се прочита и во фајлот output-genetic.txt од нашиот проект.

Enter the number of generations, i.e. the number of times to evolve the population
10
Enter the size of the population, i.e. the number of networks in each generation
20
Enter the dataset [Options: 'mnist', 'cifar10']
mnist
Enter the amount of pictures to train on [Defaults: MNIST-60000, CIFAR10-50000]
60000
Enter the amount of pictures to test on [Defaults: MNIST-10000, CIFAR10-10000]
10000
Evolving 10 generations with population 20
Doing generation 1 of 10
100%|██████████| 20/20 [5:34:15<00:00, 1002.79s/it]
Generation average: 82.09%

Doing generation 2 of 10
100%|██████████| 20/20 [52:02<00:00, 156.13s/it]
  0%| | 0/20 [00:00<?, ?it/s]
Generation average: 95.87%

Doing generation 3 of 10
100%|██████████| 20/20 [2:16:49<00:00, 410.49s/it]
  0%| | 0/20 [00:00<?, ?it/s]
Generation average: 97.57%

Doing generation 4 of 10
100%|██████████| 20/20 [12:36<00:00, 37.84s/it]
  0%| | 0/20 [00:00<?, ?it/s]
Generation average: 97.93%

Doing generation 5 of 10
100%|██████████| 20/20 [1:19:43<00:00, 239.15s/it]
  0%| | 0/20 [00:00<?, ?it/s]
Generation average: 98.06%

Doing generation 6 of 10
100%|██████████| 20/20 [01:58<00:00, 5.95s/it]

```
  0%|          | 0/20 [00:00<?, ?it/s]
Generation average: 98.20%
--------------------------------------------------------------------------------
**Doing generation 7 of 10**
100%|██████████| 20/20 [33:17<00:00, 99.87s/it]
  0%|          | 0/20 [00:00<?, ?it/s]
Generation average: 98.26%
--------------------------------------------------------------------------------
**Doing generation 8 of 10**
Generation average: 98.36%
--------------------------------------------------------------------------------
**Doing generation 9 of 10**
100%|██████████| 20/20 [59:16<00:00, 177.81s/it]
100%|██████████| 20/20 [59:38<00:00, 178.90s/it]
  0%|          | 0/20 [00:00<?, ?it/s]
Generation average: 98.36%
--------------------------------------------------------------------------------
**Doing generation 10 of 10**
100%|██████████| 20/20 [1:53:10<00:00, 339.54s/it]
Generation average: 98.36%
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.42%
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'logistic', 'optimizer': 'adam'}
Network accuracy: 98.36%
{'nb_neurons': 1024, 'nb_layers': 2, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.21%
{'nb_neurons': 1024, 'nb_layers': 3, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.12%
{'nb_neurons': 512, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.11%

Process finished with exit code 0
```

Според излезот, најголемата точност во овој случај се добила со невронската мрежа со параметри **{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}**, и истата изнесува **98.42%.**

Резултати од програмата со алгоритам на груба сила

Покрај ова, нашиот тим беше заинтересиран и за тоа за колку најголемата вредност за точноста добиена при горенаведеното извршување на програмата на основниот начин, односно со користење на генетскиот алгоритам, ќе се разликува од вистински т.е. теоретски најголемата вредност за точноста којашто може да се добие за некоја невронска мрежа при извршување на програмата. Повторно, важно е да се напомене дека добиените вредности се однесуваат на тренирање и тестирање на програмата на целосното ‘mnist’ податочно множество, и секое менување на овие влезни параметри може да резултира со различни вредности за точноста на мрежите.

Со помош на речникот со дозволени вредности на параметрите на моделот MLPClassifier којшто го користиме, можеме да го пресметаме вкупниот број на различни комбинации кои може да се направат од истите.

```
nn_param_choices = {
    'nb_neurons': [64, 128, 256, 512, 1024],
    'nb_layers': [1, 2, 3, 4],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'optimizer': ['lbfgs', 'adam', 'sgd'],
}
```

Имаме 5 можни вредности за број на неврони во даден скриен слој, 4 можни вредности за број на скриени слоеви, 4 можни вредности за тип на активациска функција и 3 можни оптимизатори. Вкупниот број на комбинации (без повторување) се добива со множење на бројот на предефинирани параметри, односно 5 x 4 x 4 x 3 = 240 вакви комбинации. Со помош на алгоритам со груба сила, ние ги испитавме сите можни различни невронски мрежи.

```
import logging
from network import Network
from tqdm import tqdm

def train_networks(networks, dataset):
    pbar = tqdm(total=len(networks))
    for network in networks:
        network.train(dataset)
        network.print_network()
        pbar.update(1)
    pbar.close()

    # Sort our final population.
    networks = sorted(networks, key=lambda x: x.accuracy, reverse=True)

    # Print out the top 5 networks.
    print_networks(networks[:5])


def print_networks(networks):
    logging.info('-' * 80)
    for network in networks:
        network.print_network()


def generate_network_list(nn_param_choices):
    networks = []

    # This is silly.
    for nbn in nn_param_choices['nb_neurons']:
        for nbl in nn_param_choices['nb_layers']:
            for a in nn_param_choices['activation']:
                for o in nn_param_choices['optimizer']:
                    # Set the parameters.
                    network = {
                        'nb_neurons': nbn,
                        'nb_layers': nbl,
```

```
                        'activation': a,
                        'optimizer': o,
                    }

                    # Instantiate a network object with set parameters.
                    network_obj = Network()
                    network_obj.create_set(network)

                    networks.append(network_obj)

    return networks


if __name__ == '__main__':
    dataset = input("Enter the dataset [Options: 'mnist', 'cifar10']")

    nn_param_choices = {
        'nb_neurons': [64, 128, 256, 512, 768, 1024],
        'nb_layers': [1, 2, 3, 4],
        'activation': ['relu', 'elu', 'tanh', 'sigmoid'],
        'optimizer': ['rmsprop', 'adam', 'sgd', 'adagrad',
                      'adadelta', 'adamax', 'nadam'],
    }

    logging.info("***Brute forcing networks***")

    networks = generate_network_list(nn_param_choices)

    train_networks(networks, dataset)
```

Формирањето на секоја од овие 240 различни невронски мрежи, нејзино тренирање на целото податочно множество и тестирање на нејзините класифицирачки способности беше голем предизвик за нас. За извршување на кодот и издвојување на резултатите на секоја од наведените мрежи на компјутерска конфигурација со процесор AMD Ryzen 5 3600X, графичка картичка AMD Radeon RX580 и 16GB DDR4 RAM, потребни беа 110 часови, 48 минути и 48 секунди, односно околу 4 денови и 15 часови. Јасно е дека во програмирањето ефикасноста и краткото време на извршување се најважни, па затоа важно е да се спомене дека извршувањето на овој конкретен пример беше исклучиво за истражувачки и експериментални цели.

Во продолжение може да се разгледа излезот кој програмата го генерираше во дел од итерациите (почетни и крајни).

Enter the dataset [Options: 'mnist', 'cifar10']
mnist
Enter the amount of pictures to train on [Defaults: MNIST-60000, CIFAR10-50000]
60000
Enter the amount of pictures to test on [Defaults: MNIST-10000, CIFAR10-10000]
10000
Brute forcing networks
  0%| | 1/240 [30:16<120:37:41, 1816.99s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'identity', 'optimizer': 'lbfgs'}

Network accuracy: 83.43%
  1%| | 2/240 [31:20<51:54:48, 785.25s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'identity', 'optimizer': 'adam'}
Network accuracy: 83.36%
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'identity', 'optimizer': 'sgd'}
Network accuracy: 83.86%
  2%| | 4/240 [36:57<24:23:26, 372.06s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'logistic', 'optimizer': 'lbfgs'}
Network accuracy: 94.05%
  2%| | 5/240 [38:52<18:13:09, 279.10s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'logistic', 'optimizer': 'adam'}
Network accuracy: 94.42%
  2%| | 6/240 [47:45<23:45:19, 365.47s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'logistic', 'optimizer': 'sgd'}
Network accuracy: 95.12%
  3%| | 7/240 [51:34<20:46:29, 320.98s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'tanh', 'optimizer': 'lbfgs'}
Network accuracy: 93.15%
  3%| | 8/240 [52:48<15:36:54, 242.30s/it]
{'nb_neurons': 64, 'nb_layers': 1, 'activation': 'tanh', 'optimizer': 'adam'}
Network accuracy: 94.16%
.
.
.
98%|[redacted]| 235/240 [106:00:38<4:58:27, 3581.50s/it]
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'tanh', 'optimizer': 'lbfgs'}
Network accuracy: 96.95%
 98%|[redacted]| 236/240 [106:23:09<3:14:09, 2912.27s/it]
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'tanh', 'optimizer': 'adam'}
Network accuracy: 97.09%
 99%|[redacted]| 237/240 [108:17:46<3:25:05, 4101.73s/it]
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'tanh', 'optimizer': 'sgd'}
Network accuracy: 97.21%
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'lbfgs'}
Network accuracy: 97.70%
100%|[redacted]| 239/240 [109:47:14<56:59, 3419.13s/it]
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.42%
100%|[redacted]| 240/240 [110:48:48<00:00, 1662.20s/it]
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'sgd'}
Network accuracy: 97.19%

{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.42%
{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'logistic', 'optimizer': 'adam'}
Network accuracy: 98.36%

{'nb_neurons': 1024, 'nb_layers': 2, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.21%
{'nb_neurons': 512, 'nb_layers': 3, 'activation': 'logistic', 'optimizer': 'adam'}
Network accuracy: 98.16%
{'nb_neurons': 1024, 'nb_layers': 3, 'activation': 'relu', 'optimizer': 'adam'}
Network accuracy: 98.12%

Process finished with exit code 0

Како што споменавме и погоре, набројани се дел од невронските мрежи заедно со точноста која ја постигнуваат и времето кое било потребно за извршување. На крајот, издвоени се петте генерирани невронски мрежи со најголеми вредности за точноста, од вкупно 240. Забележуваме дека најголемата точност која може да се добие со оваа програма е **98.42%** и истата се добива од невронската мрежа со следните параметри: **{'nb_neurons': 1024, 'nb_layers': 4, 'activation': 'relu', 'optimizer': 'adam'}.** Целосниот output можете да го пронајдете во фајлот output-brute.txt.

Заклучок

За крај, можеме да заклучиме дека она што го добивме со примена на генетскиот алгоритам, односно невронската мрежа со точност од 98.42%, е навистина одличен резултат. Истиот се совпаѓа со теоретски најдобриот резултат од програмата, кој алгоритмот со груба сила го пронајде.

Сепак, може и да не се добие најдобриот резултат во секој случај кога ја извршуваме програмата со генетскиот алгоритам, односно може да добиеме и полоши резултати, но тие во најголем дел од случаите ќе бидат само за нијанса полоши од вистински најдобриот. Односно, можеби некој ќе се запраша како генетскиот алгоритам се извршува за 14 часови и 43 минути и генерира и тренира 200 мрежи, а алгоритмот со груба сила се извршува за речиси пет дена и генерира и тренира 240 мрежи. Ова се должи на тоа што во генетскиот алгоритам, ние дозволуваме креирање на исти невронски мрежи. Постои шанса да се добијат мрежи со исти параметри низ генерациите поради самата природа на избирање на параметри на случаен начин, односно со random, а и уште повеќе поради малиот избор на можни параметри со кои располагаме. Овој број е мал воглавно поради ограничени ресурси на наша страна во однос на перформансите, но и поради самиот модел MLPClassifier од sklearn, кој нема многу избор за можни параметри. Како што споменавме погоре, оние мрежи коишто се исти не ги тренираме повторно бидејќи е излишно. Тоа значително го скратува времето на извршување. Во алгоритмот со груба сила, креирани и тренирани се 240 *различни* мрежи. Бидејќи во генетскиот алгоритам не ги испитуваме баш сите комбинации на параметри за мрежа, може да се случи да не ја генерираме токму онаа мрежа што би ни дала најдобри резултати. Исто така, мрежите се генерираат и вкрстуваат по случаен избор, за кој е јасно дека не е ист во секое извршување на програмата.

Според некои податоци пронајдени на интернет, човековите перформанси на ‘mnist’ податочното множество водат до просечна точност од 99.77%. Тоа значи дека нашиот

алгоритам сè уште не го „победил“ човекот. Меѓутоа, ние во нашиот код користевме модели коишто ни се изучени и добро познати, а тие за жал немаат многу возможни опции за параметрите. Кога би користеле некои модели со поширок опсег на параметри и нивни вредности, можеби би добиле точност поголема и од човековата.