**ТЕХНОЛОГИЧНО УЧИЛИЩЕ “ЕЛЕКТРОННИ СИСТЕМИ”**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: Реализация на алгоритъм, обучаващ се да играе аркадна игра.

Невронната мрежа взима решения на база пиксели от играта.

Дипломант: Научен ръководител:

*Мартин Йорданов Росен Недялков*

СОФИЯ

2020

Дипломанта Мартин Йорданов е изпълнил успешно поставените задачи в заданието и е покрил критериите за изготвяне на дипломна работа.

Моля да бъде допуснат до защита пред държавната квалификационна комисия.

За рецензент предлагам инж. Любомир Стоянов.

# Увод

Въпреки че видео игрите, в днешно време, се намират основно в домовете, те всъщност водят своето начало от научните лаборатории.

През 1952, британски професор създава игра на име OXO, или както е по-известна днес - ‘Морски шах’. А през 1958, Уилям Хигинботъм създава ‘Tennis for Two’, за чиято демонстрация било нужно да се свърже осцилоскоп, който играе роля на графичен монитор.

Още от зората на аркадните игри, играчите се конкурират кой ще постигне по-висок точков резултат. Това желание на хората да видят името си на първо място, довежда до множество открити закономерности, както и някой непредвидени, дори и от създателите, вратички в правилата на различните игри. За жалост, както при много други проблеми, човешките възможности са ограничени, затова се възползваме от бързо-развиващите се технологии на изкуствения интелект, като ги интегрираме във всевъзможни уреди и програми. Това което дава голямо предимство на алгоритмите за самообучение пред човешките решения, е способността на компютъра да взима хиляди решения в секунда, дори и те да не са оптимални, възможността за бърза корекция, компенсира този недостатък.

Целта на този проект е да постигна, невронна мрежа с универсална топология, тренирана да играе аркадни игри, по-точно 'SeaQuest' от Atari. Входните параметри на системата представляват данните от всеки пиксел, а изходът е число в рамките на възможните действия. Във целта се включва и реализацията на 'Attention Mechanism' и 'Графично изобразяване на loss функцията в реално време'. С цел по-лесна разработка ще бъде използван продуктът gym от OpenAI.

# Проучване

След избора на тема, алгоритъмът бе планиран да се изгради чрез технологията **NEAT (Neuroevolution through Augmenting Topologies).** След кратко проучване в тази сфера, се обърнах към контакт с богат опит в областта на машинното самообучение, който изказа мнението си, предлагайки употребата на технологията **Deep Q-Learning.** Тя предлага подход, пасващ максимално на първоначалната визия за разработката на проекта. След по обстоен преглед се изясни и фактът, че миграцията от NEAT към DQL, не предполагаше големи промени по технологичният stack, което представляваше предимство предвид отделеното време за запознанство с конкретните технологии.[[1]](#_m52bpmqk3uzi)

## 

## **Neuroevolution through Augmenting Topologies**

Важен въпрос в neuroevolution е как да спечелим предимство от развиващите се топологии и тегла на невронната мрежа. Представен е методът NeuroEvolution of Augmenting Topologies (NEAT), който превъзхожда, по бързодействие, най-добрия метод с фиксирана топология при решаването на сложна задача. Твърди се, че повишената ефективност се дължи на прилагането на принципен метод за ‘кръстосване’ на различни топологии, започвайки от минимална структура, сложността нараства с всяка следваща итерация и кръстосване на два (или повече) модела. Това твърдение се тества, чрез поредица от изследвания за аблация, които показват, че всички компоненти са полезни за системата и релацията помежду им. NEAT също има важен принос към генетичните алгоритми, тъй като показва как е възможно чрез еволюция едновременно да се оптимизира и да се усложнява дадено решения, предлагайки възможност за еволюиране на все по-сложно решавани проблеми във всяко следващо поколение и засилване на аналогията с биологичната еволюция.

При традиционните подходи в neuroevolution се избира топологията на развиващата се мрежи преди началото на експеримента. Обикновено мрежовата топология е единичен скрит слой от неврони, като всеки скрит неврон е свързан към всеки мрежов вход и всеки мрежов изход. Evolution търси в пространството на теглата на връзките на тази напълно свързана топология, като позволява да се възпроизвеждането на най-високоефективни мрежи (фиг 1.1). Пространството на теглата се изследва чрез ‘кръстосването’ на векторите с тежестите и чрез мутацията на теглата на една мрежа. По този начин целта на neuroevolution е да оптимизира теглата на връзките, които определят функционалността на една мрежа. Въпреки това теглата на връзките не са единственият аспект на невронните мрежи, който допринася за тяхното поведение. Топологията или структурата на невронните мрежи също влияе върху тяхната функционалност. Промяната на мрежовата структура се показа като ефективна като част от обучението. През последното десетилетие също има голям интерес към развиващите се мрежови топологии, както и тежести. На пръв поглед изглежда че развиващите се топологии заедно с теглата не осигурят предимство пред развиващите се тегла върху фиксирана топология, тъй като напълно свързана мрежа може да приближава всяка непрекъсната функция. Така че изглежда че няма смисъл да се губят усилия, пермутирайки различни топологии. Всъщност се оказва че най-голямата пречка при невронните мрежи е това колко време и компютърна мощ се изисква за тяхното трениране и тук е предимството на развиващите се топологии, понеже те имат по-малко връзки, следователно има по-малко числа за смятане след всяка итерация.[[13]](#_m52bpmqk3uzi)

### фиг. 1.1 Процес на кръстосване

*на две топологии*

## **PyTorch framework**

PyTorch е библиотека за машинно обучение с отворен код, базирана на библиотеката Torch, намираща приложение при разработката на

**Computer Vision** и обработка на **Natural Language** . Основнaта разработка е работа на AI Research Laboratory (FAIR), принадлежаща на Facebook. Това е безплатен продукт, с отворен код, пуснат под лиценза на **Modified BSD**. Въпреки че интерфейсът за Python е по-изчистен и е в основният фокус на развитието, PyTorch има и интерфейс C++.

Редица приложения разчитащи на **Deep Learning** са изградени върху платформата на PyTorch, включително Pyro на Uber, Transformers на HuggingFace и Catalyst.

PyTorch предоставя две функции на високо ниво, които печелят голяма част от разработчиците на софтуер:

* Tensor изчисления (като NumPy) със силно ускорение чрез графични процесорни единици (GPU)
* Възможност за изграждане на дълбоки невронни мрежи, изградени на основана на автоматично диференциране
* **Dynamic computation graphs** - Вместо предварително дефинирани граф със специфични функционалности, PyTorch предоставя възможността да изграждат изчислителни графи, по време на изпълнението **(runtime)**.

Създателите на PyTorch казват, че тяхната философия - всичко да бъде императивно. Това означава, че извършва изчисленията си веднага. Това пасва перфектно на методологията на програмиране в python, тъй като не е необходимо да се чака да бъде написан целият код, преди да се разбере дали работи или не. Лесно може да бъде стартирана част от кода и да бъде инспектирана в реално време. Това се указва много потребно, когато се стигне до момента в който трябва да се отстрани някоя грешки в невронната мрежа.

От излизането ѝ в началото на януари 2016 г. много изследователи я възприемат, поради лесното изграждане на изключително сложни графи. Библиотеката все още не е широко разпространена, поради простата причина е, че тя все още е в процес на разработка, което носи със себе си своите опасности и проблеми.

## **SciKit-learn framework**

Scikit-learn, още известна като sklearn е безплатна софтуерна библиотека предназначена за машинно самообучение, намисана на езика за програмиране Python. Тя разполага с различни алгоритми за **класификация**, **регресия** и **clustering**, включително поддържащи **векторни машини**, **random forests**, **gradient boosting**, **К-means** и **DBSCAN** (Density-based spatial clustering of applications with noise),

и е проектирана да работи съвместно с научните библиотеки на Python NumPy и SciPy.

Проектът за разработка на scikit стартира като scikits.learn, проект на Google Summer of Code начело с David Cournapeau. Името ѝ произтича от идеята, за съчетаване на думите „SciKit“ (SciPy Toolkit), но все пак подчертаващо, че е отделно разработено и разпространявано, third-party разширение на към SciPy. Оригиналната база по-късно е пренаписана от други разработчици. През 2010 г. Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort и Vincent Michel, всички от Френския институт за изследвания в областта на компютърните науки и автоматиката в Rocquencourt, Франция, поели ръководството на проекта и направиха първата публична версия на 1 февруари 2010 г. различни scikits, scikit-learn и scikit-image бяха описани като „добре поддържани и популярни“ през ноември 2012 г. Scikit-learn е една от най-популярните библиотеки за машинно обучение в GitHub, благодарение на своите възможности:

* Cross-validation: Има различни методи за проверка на точността на контролираните модели върху невиждани данни
* Unsupervised learning: Отново има голямо разпространение на алгоритми в предлагането - като се започне от clustering, factor analysis, principal component analysis до unsupervised neural networks.
* Разнообразен набор от данни за експерименти: Това е полезно, докато при разучаване на scikit-learn. Например разучаването на SAS с помощта на различни набори от данни, като IRIS

и Boston House. Това, че са удобни за употреба, докато бъдат разучени функционалностите на библиотека, помогна много, при начинаещи.

* Feature extraction: Полезно за извличане на функции от изображения и текст, например при Computer Vision.

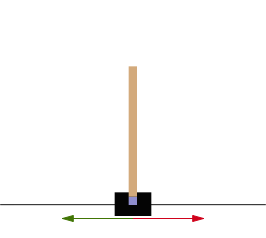
## Pandas library

Появата на Pandas предоставя на потребителите още едно ниво на абстракция, когато става въпрос за анализиране на данни с Python и е един от най-предпочитаните и широко използвани инструменти при обработка на данни, ако не и най-използваният. Pandas е с отворен код, свободен за използване, под лиценз BSD и първоначално е написан от Уес Маккини.

Най-голямото предимство на Pandas е, че той приема данните, като CSV и TSV файл или дори SQL база данни, и създава Python обект с редове и колони, наречени **data frame**, който изглежда много подобен на таблицата в статистическите софтуерни продукти като Excel или SPSS, а хората запознати с R, също биха видели сходства с него. Така е много по-лесно да се използват данните в сравнение с работата със списъци или асоциативни масиви, обхождани чрез цикли и list comprehension-и. Валидирането на данни е много важна част от анализа на данните. Винаги се проверява за липсващи стойности в данните, като в случаят това става по следният начин: ***pd.isnull()***, този метод проверява за нулеви стойности, и връща булев масив (масив със true за липсваща стойности и false за не-липсващи стойности). А методът ***pd.notnull()*** представлява обратното на ***pd.isnull()***. Pandas предлага и няколко различни подхода за справяне с липсващите стойности, най-често използваният е запълването на липсващите стойности с други стойности, като се използва ***df.fillna(x)***, която запълва липсващите стойности с **x**, а ако на мястото на **х** поставим - ***s.mean()***, то тогава всички нулеви стойности ще бъдат заменени със средна стойност.[[7]](#_m52bpmqk3uzi)

## Cartpole Deep Q-Learning

Cartpole - известно още като ‘Обърнатото махало’ е игра представляваща - махало с център на тежестта над неговата точка на въртене (фиг 1.2). То е нестабилно, но може да се контролира чрез преместване на точката на въртене под центъра на масата. Целта е да се поддържа балансирано махалото, като прилагате подходящи сили към точка на въртене.[[3]](#_m52bpmqk3uzi)



### фиг 1.2 Схематичен чертеж на Cartpole

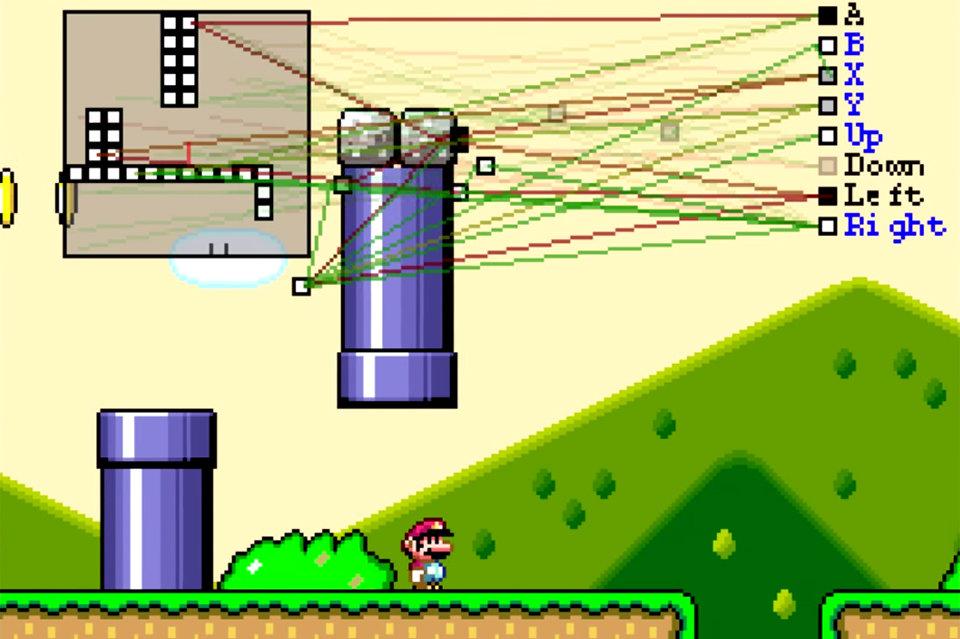
* Виолетовият квадрат обозначава точка на въртене.
* Червените и зелени стрелки показват възможни хоризонтални сили, които могат да бъдат приложени към точка на въртене.
* Чрез прилагането на всички налични техники алгоритъмът се научава да играе играта средно за ~191 опита.

Разликата между този проект (Cartpole) и разглежданията в документацията (SeaQuest) се корени, в това че досега алгоритъмът

Deep Q-Learning се прилага основно за по-прости игри тъй като не е толкова широко използван с такова приложение, Cartpole е един пример за това. По-голямата простота при Cartpole идва от, това че пространството на възможните действия (action space) е много по-малко (2D вектор), отколкото при SeaQuest, където то представлява 18-измерен вектор. Разбира се има няколко проучвания в които се доказва че този вид алгоритми са способни да научат много на брой и то сложни игри от ранга на SeaQuest, но точните параметри на алгоритъма използван при проучването не са публични.

## NeuroEvolution with MarI/O

Може би това, че всички нива имат прости цели, отляво - надясно, но по някаква причина по-старите игри на Mario отдавна са цел на много ентусиасти, които се интересуват от машинно самообучение. Най-новото постижение се казва MarI/O. И то научава цяло ниво на Super Mario World само за 34 опита.

За разлика от други алгоритми и програми, MarI/O (фиг. 1.3) не е трениран на нищо преди да бъде пуснат в играта - дори не знае, че краят на нивото е вдясно - вместо това, му биват зададени някои прости параметри. Алгоритъмът има ниво на стойност на качеството (fitness), която се увеличава, колкото по-напред в нивото достигне, а ако героят се движи наляво, стойността намалява. Единственото нещо което алгоритъмът знае, е че стойността показва колко добре се справя и така, след като разбере, че движението в дясно увеличава тази статистика, той бива стимулиран да продължи.**

### фиг. 1.3 Графично представяне на невронната мрежа

### и потребителски изглед

# Изисквания

След проучвателната част на тази дипломна работа, бе ясно че изборът на аркадна игра се свежда до производителят **Atari**,тъй като тази компания има много богат каталог от игри. От тези игри почти всички са разучавани (статистически), освен това тяхната сложност варира в много широк спектър, възможните действия (action space) в различните игри се простират от 1-2, до 20. По-късно открих библиотеката **gym**, която предоставя възможността да бъдат изпробвани различни алгоритми, срещу елементарни игри. Разглеждайки каталогът с потенциялни игри, сведох изборът до следните три игри: **Pong, Breakout, SeaQuest**. Но тъй като първите две изброени имат прекалено малко поле на действие (action space) - 2 (ляво - дясно), а **SeaQuest** притежава 18 потенциални действия от всяка позиция. Така бе избрана тази игра за тази конкретна дипломна работа.

## Основа на проектът

Както вече бе споменато алгоритъмът, представляващ този проект, е трениран на база играта на име **“SeaQuest”** (фиг. 2.1). 

### фиг. 2.1 Стандартен екран от играта

В нея играчът използва подводница, за да стреля по врагове и спасителни водолази (фиг. 2.2). 

### фиг. 2.2 Водолази (долу под кислорода)

Враговете включват акули и подводници, които изстрелват ракети по подводницата на играча. Играчът трябва да избягва враговете (фиг. 2.3), като изстрелва неограничен брой ракети, докато се опитва да спасява водолазите, плуващи през водата.

### фиг 2.3 Противници (врагове) в играта

подводница (сиво) и акула (лилаво)

Точките, присъдени на играча за стрелба по враг, започват от 10 точки всяка и се увеличават с напредването на играта. Подводницата може да побира до шест водолаза наведнъж (фиг. 2.2). Всеки път, когато играчът изплува отново, преди да има пълен товар от шест водолаза, един от водолазите се отстранява.

### фиг. 2.4 Точков резултат (отгоре); брой животи (отдолу)

Подводницата има ограничено количество кислород. Играчът трябва да се появява често на повърхността, за да попълни кислорода (фиг. 2,5), но ако играчът се появи отново без спасени водолази, той ще загуби живот. Ако играчът се появи отново с максималното количество водолази, той ще получи бонус точки за останалия кислород на подс. Всеки път, когато играчът се появи на повърхността, трудността на играта се увеличава; враговете се увеличават по брой и скорост.

### фиг. 2.5 Кислороден запас в подводницата

В крайна сметка вражеска подводница започва да патрулира по повърхността, оставяйки играча без сигурно убежище.

Играчът започва играта с 3 допълнителни живота (фиг. 2.4) и му се предоставя допълнителен допълнителен живот всеки път, когато играчът спечели 10 000 точки (фиг. 2.4).[[10]](#_m52bpmqk3uzi)

## Изисквания към приложението

* Проектиране и трениране на невронна мрежа
* Универсалност на топологията на невронната мрежа
* Имплементация на “Attention mechanism”
* Графика в реално време на “loss function”

Трябва да бъде изградена основна структура на приложението. Да се направят основни класове, представляващи различни видове алгоритми за самообучение, чрез които добавянето на нови функционалности да става бързо и лесно. Да се изгради лесна и логически-издържана комуникация между различните части на приложението, като графиката в реално време и играта да бъдат винаги синхронизирани.

Алгоритъмът трябва успешно да разпознава и тълкува заобикалящата го среда. Всеки противник в играта трябва да бъде ясно разграничаван. Случващите се изчисл не трябва да забавя изпълнението на играта или графиката, като кадрите в секунда е желателно да не падат под 30. Графиката демонстрираща качеството на работа на алгоритъмът трябва да е с неограничена големина без това да се отразява на производителността на целия продукт.

Приложението трябва да изисква възможно най-малко от устройството. Данните, получени от помощната библиотека **gym**, трябва да бъдат обработвани достатъчно бързо, подсигурявайки взетото решение от невронната мрежа да бъде своевременно с цел трениране във възможно най-близки условия до тези в реална обстановка когато не би могло да се забавя действието на играта. Цялостното закъснение от страна на приложението трябва да е минимално през цялото време на работа.

## Съображения за избор на програмни средства и развойната среда

За реализацията на проекта е избрано да се използват библиотеки поради многото си предимства при разработка, като например:

* Множество имплементирани математически операции и функции, които спестяват време на разработчиците.
* Изключително високо ниво на абстракция, което дава възможност за концентрация върху по съществени промени.
* Сигурността, която библиотеките осигуряват, поради факта че са разработени от големи фирми/групи програмисти, това намалява шанса от грешки при разработката.

Избраната библиотека в този проект е **keras**, тя предоставя възможност за композиране на много сложни невронни мрежи, благодарение на вече имплементираните си класове и методи. Друго голямо предимство на този продукт пред другите е, че той вече е използван и по този начин се прескача времето нужно за запознанство с възможностите му.

## GitHub система за контрол и управление на версиите

GitHub представлява уеб-базирана услуга за качване на софтуерни проекти върху отдалечен сървър (т.нар. хранилище) и осъществяването на съвместни разработки върху тях. Базиран е на Git системите за контрол и е избран заради вече съществуващия опит с него.

## 

## Visual Studio Code текстов редактор

Цялостната процедура по писане на кода беше осъществена в Visual Studio Code. То представлява текстов редактор, разпространяващ се като софтуер с отворен код. Главното му предимство е множеството плъгини и модули, които са налични за сваляне веднага след инсталация, той разполага и със сравнително опростен интерфейс и оцветяване на синтаксиса за над 50 програмни езика.

## Keras библиотека за невронни мрежи

Тази библиотека позволява създаването на всевъзможни невронни мрежи, чрез помощта на клас “контейнер” в който биват добавяни слоевете на мрежата, накрая чрез извикването на един метод върху контейнера, той си създава всички връзки между невроните и слоевете, без нужда от сложни операции от страна на разработчика.[[4]](#_m52bpmqk3uzi)

### 

## Gym библиотека за симулация на аркадни игри

OpenAI (фиг. 2.6) е независима изследователска организация, създадена от корпорацията OpenAI Inc. и нейната дъщерна организация, с нестопанска цел OpenAI LP. Корпорацията провежда изследвания в областта на изкуствения интелект със заявената цел да насърчава и развива тази сфера по такъв начин, че да бъде от полза за човечеството като цяло. Организацията е основана в края на 2015 г., базирана в Сан Франциско тя има за цел постигането на свободно сътрудничество с други институции и изследователи, като прави патентите и изследванията си свободни за употреба от общността.

Gym е инструментариум за разработване и сравняване на reinforcement learning алгоритми помежду им. Той притежава агенти за обучение на почти всички познати игри, а и не само. Gym притежава среда за ходене и пазене на баланс, както и на игри като Pong или Pinball. Но в случаят на този проект, от най-голяма полза са средите в категория Atari.[[16]](#_m52bpmqk3uzi)

# Програмна реализация на “DeepQuest”

Универсалността на топологията на невронната мрежа, позволява тя да се използва върху множество игри, но с цел демонстрация на проекта, а и поради ограничената компютърна мощ, алгоритъмът бе обучен единствено върху играта “SeaQuest” на Atari.

## Дизайн на алгоритъма

Невронната мрежа се състои от 3 конволюционни слоя, следвани от 3 напълно-свързани неврони.

* Входът на мрежата е с размери - **(210, 160, 1)**
* **Конволюция, чието ядро е с размери (5, 5) и 8 на брой филтри, активация - ReLU.**
* Пуулинг с рамка (2, 2).
* Конволюция, чието ядро е с размери (3, 3) и 16 на брой филтри, активация - ReLU.
* Пуулинг с рамка (2, 2).
* Конволюция, чието ядро е с размери (3, 3) и 8 на брой филтри, активация - ReLU.
* Пуулинг с рамка (2, 2).
* Зануляване на 20% от стойностите.
* **Линеализиране на стойностите от матрица (24, 18, 8),**

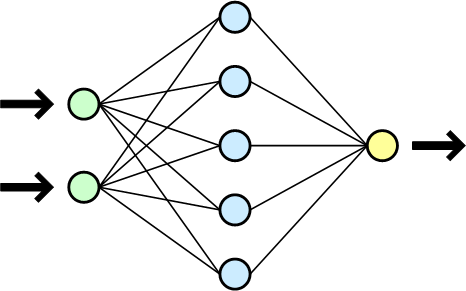
**към вектор 3456.**

* **256** невронен слой, активация - ReLU.
* **64** невронен слой, активация - ReLU.
* изходът е с **18** на брой стойности, всяка представляваща вероятност определено действие да бъде най-подходящо, активация - линейна.

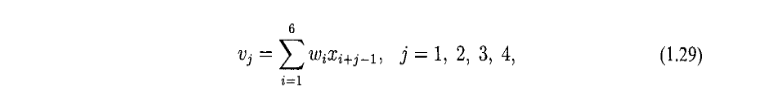
Допълнително при добавянето на **skip connections,** са добавени и връзки между двата слоя отбелязани с удебелен шрифт.

## Невронни мрежи

Математическият аналог на биологичната невронна мрежа представлява множество от взаимосвързани прости изчислителни елементи (неврони) (фиг. 3.1). Всеки неврон приема сигнали от предходните (под формата на числа), сумира ги, като сумата минава през активационна функция, и така се нормализира стойността, която се предава по изходящите връзки към другите неврони.



### фиг 3.1 Опростен изглед на изкуствена невронна мрежа

Всяка връзка има тегло, което, се умножава със сигнала, като така определя неговата значимост (фиг. 3.2). Теглата на връзките са аналогични на силата на синаптичните импулси, предавани между биологичните неврони. Отрицателна стойност на теглото съответства на потискащ импулс, а положителна – на възбуждащ.

### фиг. 3.2 Математическо представяне на формула за

### пресмятане на резултат в един неврон

В невронната мрежа винаги съществува входен и изходен слой от неврони, във входния се въвежда информацията към мрежата, след това сигналите от входните неврони преминават през един или няколко слоя от междинни (скрити) неврони, според топологията на невронната мрежа, като сигналите накрая стигат до изходния слой, откъдето се чете получената информация.

Математически е доказано, че всяка невронна мрежа с поне един скрит слой от достатъчно на брой неврони между входния и изходния слой, може да симулира поведението на всяка съществуваща функция.

Теглата на връзките между невроните определят функционалността и поведението на невронната мрежа. За да бъде една невронна мрежа използваема и приложима към даден проблем, тя трябва да бъде предварително обучена.

Обучаването на една невронна мрежа се извършва чрез промяна на теглата на връзките между невроните и се осъществява чрез правила, които определят как да се променят тези тегла. Най-разпространеното сред тях е метода на обратното разпространение на сигнал за грешка (back-propagation), където за всеки изходен неврон се изчислява разликата от желаното му поведение, като се формира сигнал за грешка, който се движи назад към входния слой и по пътя си променя теглата на връзките така, че при следващата активация на мрежата грешката да бъде по-малка от сегашната. Този начин на „обучение“ на мрежата обаче води до „забравяне“ – ако мрежата бъде обучена да разпознава един елемент и впоследствие той не се повтори във входните данни, мрежата „забравя“ този елемент.

В настоящият момент на невронните мрежи се гледа като на инструмент, който помага за решаването на проблеми, без ясна първоначална представа за това какви са стъпките за решаване на задачата. Невронните мрежи работят на принципа, ключовите елементи, след като една мрежа бъде тренирана могат да се изследват тежестите ѝ, като по-този начин става ясно кои части от информацията с най-важни.

За разлика от класическия, символен подход към изкуствения интелект, невронните мрежи много повече се доближават до мозъка на човешкия индивид по начина си на функциониране. Те се състоят от множество свързани прости изчислителни единици (неврони). Изчисленията не се извършват на едно централно място, а са разпределени по цялата мрежа. Това прави мрежата изключително гъвкава, като тя продължава да работи дори когато от нея умишлено са премахнати изчислителни елементи. За сравнение, фон Ноймановата архитектура престава да функционира ако от нея липсва дори един елемент.

Този модел на човешкия мозък не е перфектен. Повечето съществуващи към момента изкуствени невронни мрежи не могат да моделират ефекта на хормоните върху мозъка. Освен това се оказва, че за да може математически да се опише една изкуствена мрежа, в нея трябва да се спазват определени правила, например сигналите да се движат от входния към изходния слой, но не и обратно. (Съществуват и изкуствени мрежи, които са циклични (рекурентни), но поради повишената им сложност все още не са добре изучени.) Може би най-големият недостатък на невронните мрежи е, че тяхното обучение изисква множество итерации.[[14]](#_m52bpmqk3uzi)

## Конволюционен слой

Конволюционният слой е основният градивен елемент на CNN (Convolutional Neural Network) (фиг. 3.3). Параметрите на слоя са следните:

* + 1. **Набор от ядра (kernels)**

Това представляват малки матрици, обикновено с размери 3х3, 5х5 или 7х7. Тези ядра се движат върху входните данни (представени в матричен вид), единственото изискване е броят на измеренията на двете матрици да е равен. Ядрата притежават малко поле на възприятие, но то се простира в пълната дълбочина на входните данни. По време на движението си напред всеки филтър преминава по ширината и височината на входните данни, изчислявайки точково произведение между стойностите на филтъра и входа, като по този начин генерира двуизмерна карта на активиране (activation map) на този филтър. В резултат на това мрежата научава филтри, които се активират, когато открие някакъв специфичен тип, отличителна черта, във пространството на входните данни.[[8][9]](#_m52bpmqk3uzi)

* + 1. **Брой филтри на слой**

Всеки слой притежава повече от едно ядро, с цел научаване на филтри за повече отличителни черти. Подреждането на активационните картите на всички филтри по измерението на дълбочината формира пълния изходен набор от данни на слоя. Всеки запис на изхода може също така да се интерпретира като изход на единичен неврон, който разглежда малък регион във входа и споделя връзки с неврони в една и съща карта на активиране. Друг важен хипер-параметър на слоевете е стъпката, тя задава през какво разстояние да се налагат филтрите върху входните данни.

### фиг. 3.3 Графично представяне на процесите в

### конволюционните слоеве

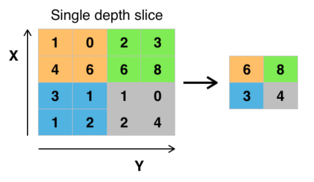
* + 1. **Локална свързаност**

Когато се работи с многоизмерни входове като изображения, е непрактично да се свързват всеки неврон към всички останали в предишния слой (фиг. 3.3), тъй като такава мрежова архитектура не взема предвид пространствената структура на данните. Конволюционните мрежи използват пространствено локална корелация чрез налагане на минималистична локална схема на свързаност между невроните на съседни слоеве: всеки неврон е свързан само с малка област от входните данни. Степента на тази свързаност е хипер-параметър, наречен рецептивно поле на неврона. Връзките са локални в пространството (по ширина и височина), но винаги се простират по цялата дълбочина на входа. Такава архитектура гарантира, че научените филтри произвеждат най-високи стойности в локален пространствен шаблон.

### фиг. 3.3 Неврони от конволюционен слой (синьо), свързани към рецепторното ядро (червено)

## Пуулинг

Друга важна концепция на конволюционните мрежи е пуулингът (pooling), което е форма на нелинейно намаляване на обема на данните. Има няколко нелинейни функции за осъществяване на пуулинг, сред които най-често се среща взимане на най-голямата стойност от групата (max). Тази функция разделя входните данни на набор от неприпокриващи се групи и за всяка такава поднабор извежда максимума.

Логично, точното местоположение на дадена стойност е по-маловажно от грубата ѝ позиция спрямо другите части. Това е идеята, която стои зад използването на пуулинг в конволюционните невронни мрежи. Слоят за пуулинг служи за прогресивно намаляване на пространствения размер на данните, респективно за намаляване на броя на параметрите, необходимата паметта и количеството на изчисленията в мрежата, а оттам и за контролиране на претренирането (overfitting). Обикновено е прието да се вмъква пуулинг слой между последователни конволюционни слоеве при CNN архитектурите. Операцията за обединяване осигурява друга форма на транслационна инвариантност.

### фиг. 3.4 Max pooling с размер на филтъра 2x2 и стъпка = 2

Пуулинг слоят работи независимо от дълбочината на входа и го оразмерява пространствено. Най-често срещаната форма е пуулинг слой с филтри с размер 2 × 2, приложени със стъпка от 2 (фиг. 3.4). На всяко ниво, независимо от дълбочината, входът на слоя се намалява 2 пъти по ширина и височина, съкращавайки със 75% броя на необходимите активации.[[15]](#_m52bpmqk3uzi)

## Визуализиране на стойности

В хода на разработка и интеграция на графиката демонстрираща качеството на алгоритъма, изникнаха няколко затруднения. Първият проблем бе трудната реализация на актуализиращата се, в реално време, графика на loss функцията. От библиотеката **matplot**, бяха избрани няколко на брой, потенциално решаващи проблема функции и методи, след опитите с тях, бе установено, че всеки път се създава нов прозорец, в който се визуализира само последната стойност, без да се създава представа за анимация. В крайна сметка след задълбочено търсене и множество опити, бе открит начин да се реализира функционалността за което се използва методът ***.pause(sec)***. Той намира приложение при последователно изобразяване на множество от точки, като това се случва със забавяне от N секунди между всеки 2 точки. В този случай методът е употребен поради факта, че когато той се използва, вместо да отвори нов прозорец, той използва същият. За реализацията, бе използван клас приемащ име на двете оси, при създаването си. Основният метод в този клас, извикван множество пъти е **plot\_loss**, той приема стойност и я визуализира върху графиката отмествайки **х** с 1-ца.[[17]](#_m52bpmqk3uzi)

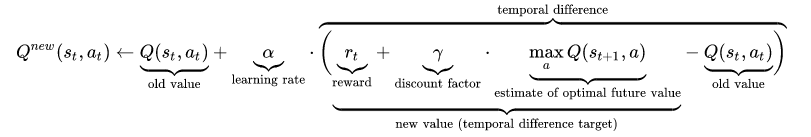
|  |
| --- |
| class LossPlotter:  STEP = 0   def \_\_init\_\_(self, x\_axis='Time', y\_axis='Loss', title='Real-time loss plot'):  self.x\_axis = x\_axis  self.y\_axis = y\_axis  self.title = title  self.setup()   def setup(self):  plt.title(self.title)  plt.ylabel(self.y\_axis)  plt.xlabel(self.x\_axis)   def plot\_loss(self, value):  plt.plot(self.STEP, value, 'r+')  plt.pause(0.001)  self.STEP += 1 |

## Deep Q-Learning

Q-learning е алгоритъм за обучение без модел. Целта на му е да научи политика (policy), който казва на агент какви действия да предприеме при конкретните обстоятелства. Той не изисква модел (оттук и конотацията „model-free“) на заобикалящата го средата и може да се справя с проблеми изградени от стохастични преходи и награди, без да се нуждае от адаптация.[[1]](#_m52bpmqk3uzi)

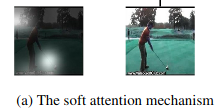
За всеки краен граф на възможните ходове в дадена игра, Q-learning намира политика, която е оптимална по отношение на това, че максимизира очакваната стойност на общата награда и всички последващи стъпки, като се започне от текущото състояние.

Преди да започне обучението, ***Q*** се инициализира с произволна стойност, в случаят тя зависи изцяло от инициализацията на тежестите в мрежата. След всяка стъпка ***t*** агентът избира действие - ***at*** , за което получава награда ***rt*** , което от своя страна довежда до ново състояние ***st+1*** (което може да зависи както от предишното състояние ***st*** , така и от приложеното действие). ***Q*** се актуализира по зададена формула (фиг. 3.5). Ядрото на алгоритъма е уравнението на Белман, представляващ просто итерационно актуализиране на стойност, използвайки средно претеглената стара стойност и получената нова информация:



### фиг. 3.5 Уравнение на Белман

## Attention mechanism

Този похват кара невронните мрежи да научават кои части от входните данни са най-важни. На теория съществуват два вида механизми: hard-attention и soft-attention. Те се реализират посредством рекурентни невронни мрежи (Recurrent Neural Networks) и LSTM (Long Short-Term Memory) модули. Принципът е лесен, с времето мрежата научава шаблони и започва да изгражда карта (heatmap), която определя кои райони са с по-голяма важност. На фиг. 3.6 се вижда в ляво heatmap, който показва най-важните части в изображението, за невронната мрежа (в бяло), а отдясно е представено оригиналното изображение.[[18]](#_m52bpmqk3uzi)

### фиг. 3.6 Attention Mechanism prospective

## Residual neural network

Този вид невронните мрежи, печелят своето предимство ускорявайки процеса на трениране, като за сметка на това, добавя повече тежести към мрежата. Замисълът е прост, при коригирането на тежестите при процесът наречен **backpropagation,** се изчисляват производната на грешката за всеки слой, после този “наклон” се умножава по параметъра **learning-rate,** който е от порядъка на 0.001➗0.09, последната стъпка е изваждането на получената стойност от всички тежести (операция минус върху матрица). Проблемът идва когато невронната мрежа е дълбока (**Deep Neural Network**), защото със всеки следващ слой, гореспоменатата производна намалява, като по този начин колкото по-близо се намира конкретен слой до входа на мрежата, толкова повече итерации са нужни за корекцията на неговите тежести.

Чрез добавянето на “прескачаща връзка”, се създава “пряк път” (фиг. 3.7), прескачащ N слоеве, като по този начин, производната не “затихва” толкова много, стигайки до конкретния слой. Крайният резултат е по-бързо трениране на мрежата за сметка на малко повече памет.[[19]](#_m52bpmqk3uzi)

### 

### фиг. 3.7 Градивен блок на ResNet

## Същинска разработка

Първата стъпка от същинската разработка бе създаването на инстанция на клас (фиг. 3.8), симулиращ играта SeaQuest, това е възможно благодарение на вече споменатата библиотека **gym**. След това благодарение на богатия набор от атрибути в **env**, може да бъде получена информация за размерите на входа, както и за тези на изхода.

|  |
| --- |
| env = gym.make('Seaquest-v0')  state\_size = env.observation\_space.shape  action\_size = env.action\_space.n |

### фиг. 3.8 Създаване на инстанция и извличане на важни данни

Тъй като играта е цветна, всеки пиксел се изразява с 3 стойности, във формата RGB (red-green-blue), като числените стойности варират от 0 до 255. С цел опростяване на невронната мрежа и намаляване на броя на стойностите в нея, се взима решение всеки пиксел да бъде конвертиран в сивия спектър, по този начин той се изразява само с една стойност между 0 и 255 (фиг. 3.9). Този процес е известен под името grayscaling, с помощта на тема във форум, са открити конкретните константи с които трябва да се умножи всяка стойност от RGB.[[5][6]](#_m52bpmqk3uzi)

**aij** = 0.298 \* **rij** + 0.587 \* **gij** + 0.114 \* **bij**

Константите **HEIGHT** и **WIDTH**, са стойности отговарящи на резолюцията на играта, в конкретният случай тя е 260х160.

|  |
| --- |
| def rgb2gray(rgb):  # A proven formula for converting RGB to Grayscale  gray\_frame = np.dot(rgb, [0.2989, 0.5870, 0.1140])  # Put the color in a container, to simulate a color-channel  return gray\_frame.reshape(HEIGHT, WIDTH, 1) |

### 

### фиг. 3.9 Конвертираща функция - RGB -> Grayscale

С цел енкапсулация и по-добро разделение на модули, цялата логика на невронната мрежа е имплементирана в един клас (**DQAgent**). Като той съдържа следните методи и атрибути:

* Конструктор

|  |
| --- |
| def \_\_init\_\_(self, state\_size, action\_size):  self.state\_size = state\_size[:2] + (1,) # no. inputs + 1-channel color  self.action\_size = action\_size # no. outputs  self.memory = deque(maxlen=2000) # decision register  self.gamma = 0.85 # discount rate  self.epsilon\_decay = 0.995  self.epsilon\_min = 0.02  self.learning\_rate = 0.001  self.epsilon = 1 # exploration rate  self.model = self.\_build\_model()  self.plotter = LossPlotter() |

Като Ɣ (gamma) е в ролята на намаляващ коефициент във формулата на Белман (фиг. 3.5). Ε (epsilon) определя колко често изградената политика (тактика) ще взима случайни решения с цел откриване на по-добър избор и обогатяване на възможностите в определена ситуация в бъдеще, тя приема стойност между 0 и 1. epsilon\_decay се умножава с epsilon след всяко случайно решение, тъй като това е по-важно в началото на тренировъчният процес, а при вече тренирана мрежа дори може да окаже лош ефект върху ефективността ѝ. еpsilon\_min задава долна граница на стойността на epsilon, с други думи не позволява на еpsilon да приема стойност по-малка от epsilon\_min.

* Създаване на модела

|  |
| --- |
| def \_build\_model(self):  # Neural Net for Deep-Q learning Model  model = Sequential(name='Questor1.0')  model.add(Conv2D(kernel\_size=5, filters=8, input\_shape=self.state\_size, activation='relu'))  model.add(MaxPooling2D(pool\_size=2))   model.add(Conv2D(kernel\_size=3, filters=16, activation='relu'))  model.add(MaxPooling2D(pool\_size=2))   model.add(Conv2D(kernel\_size=3, filters=8, activation='relu'))  model.add(MaxPooling2D(pool\_size=2))   model.add(Dropout(0.2))  model.add(Flatten())   model.add(Dense(256, input\_dim=self.state\_size, activation='relu'))  model.add(Dense(64, activation='relu'))  model.add(Dense(self.action\_size, activation='linear'))  model.compile(loss='mse',  optimizer=Adam(lr=self.learning\_rate))  return model |

Топологията бе описана по-рано вече в това книжно тяло, тук се вижда как се изгражда тя с помощта на библиотеката keras. Тъй като тук няма прескачащи връзки (skip-connections), създаването става изключително лесно. Посредством опцията предоставена от Keras - Sequential model

* Преиграваща функция (replay)

|  |
| --- |
| def replay(self, batch\_size):  batch = random.sample(self.memory, batch\_size)  states, targets\_f = [], []   for state, action, reward, next\_state, done in batch:  target = reward  if not done:  a = self.model.predict(next\_state)  target = (reward + self.gamma \* np.amax(a[0]))   target\_f = self.model.predict(state)  target\_f[0][action] = target   states.append(state[0])  targets\_f.append(target\_f[0])  history = self.model.fit(np.array(states), np.array(targets\_f), epochs=1, verbose=0)   loss = history.history['loss'][0]   if self.epsilon > self.epsilon\_min:  self.epsilon \*= self.epsilon\_decay   self.plotter.plot\_loss(loss)   return loss |

Ролята на тази функция е да преиграва сцени от миналото и с натрупаните промени (потенциално подобряващи ефективността) да определи дали това е било най-доброто решение в онзи случай, като тук голяма роля играе и фактът че вече се знае каква награда следва **Q(st , at)** -> **rt , st+1** ,

Затова и се запазват точно тези параметри като самостоятелна единица за replay, като допълнителен параметър се съдържа и булевата стойност (**done**) дали това решение **Q(st , at)**, не е предизвикало края на играта.При всяко извикване на функцията се избират batch\_size на брой случайни “моменти” запазени в deque-a, който играе роля на паметта на този алгоритъм.[[11][12]](#_m52bpmqk3uzi)

В процес на разработка бе установено, че Attention mechanism няма как да бъде уместно имплементиран в този проект, поради липсата на времева зависимост между извикванията. Компенсирането на тази липса е осъществено чрез добавянето на “Прескачащи връзки” (skip-connections), тези връзки са основен градивен блок при **Residual Learning** техниката, но желаният ефект не е постигнат. След реализацията на тази функционалност не се забелязва подобрение в представянето на алгоритъма, поради тази причина функционалността е оставена на отделен клон в хранилището.

# Ръководство на потребителя

За употребата на този проект са нужни минимални технически знания от страна на потребителя. Като единственото необходимо знание е употреба на GitHub и евентуално git, в случай че потребителя не желае дa използва GitHub. Този проект е разработен под операционна система Ubuntu 19.10 и се препоръчва да бъде изпълняван под Linux.

## Изтегляне на проекта

В тази част от употребата съществуват два варианта, единият е въвеждайки адресът на проекта в GitHub ([**https://github.com/xxm0703/DeepQuest**](https://github.com/xxm0703/DeepQuest)), след това да се изтегли под формата на архив с разширение .zip. Другият вариант е да се вземе адресът на самото хранилище и с помощта на git то да бъде клонирано.

## Сваляне през браузъра

Потребителят трябва да въведе горе-посоченият адрес, след което ще бъде отведен на конкретна страница (фиг. 4.1).

### 

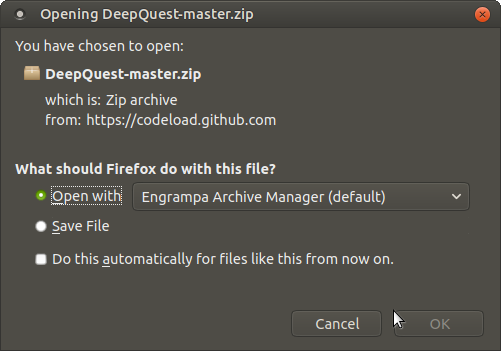
### фиг. 4.1 Стандартен изглед на хранилище в GitHub

След като страницата зареди, потребителят трябва да натисне зеления бутон намиращ се в горния десен ъгъл на екрана с текст “Clone and download” (фиг. 4.2)

### 

### фиг. 4.2 Бутон за клониране и изтегляне

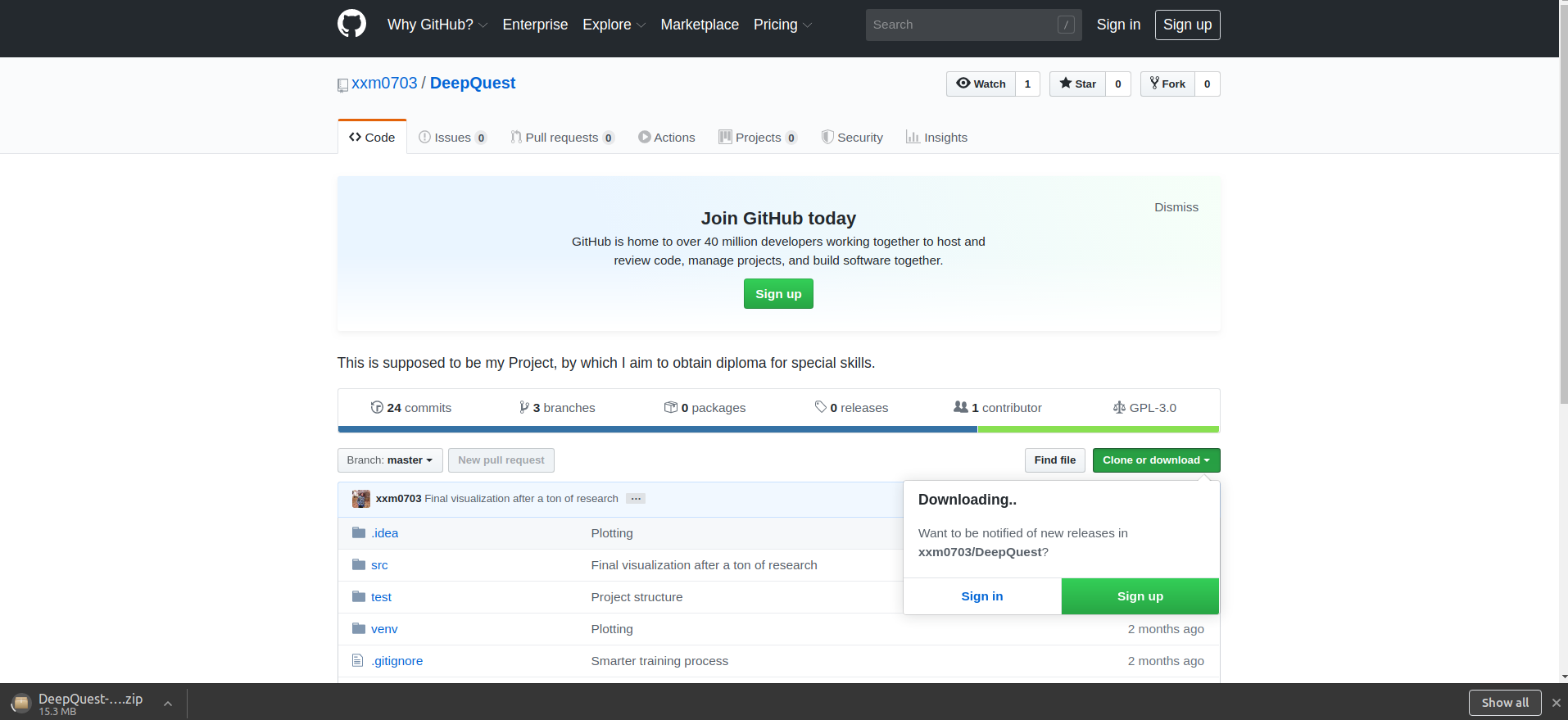
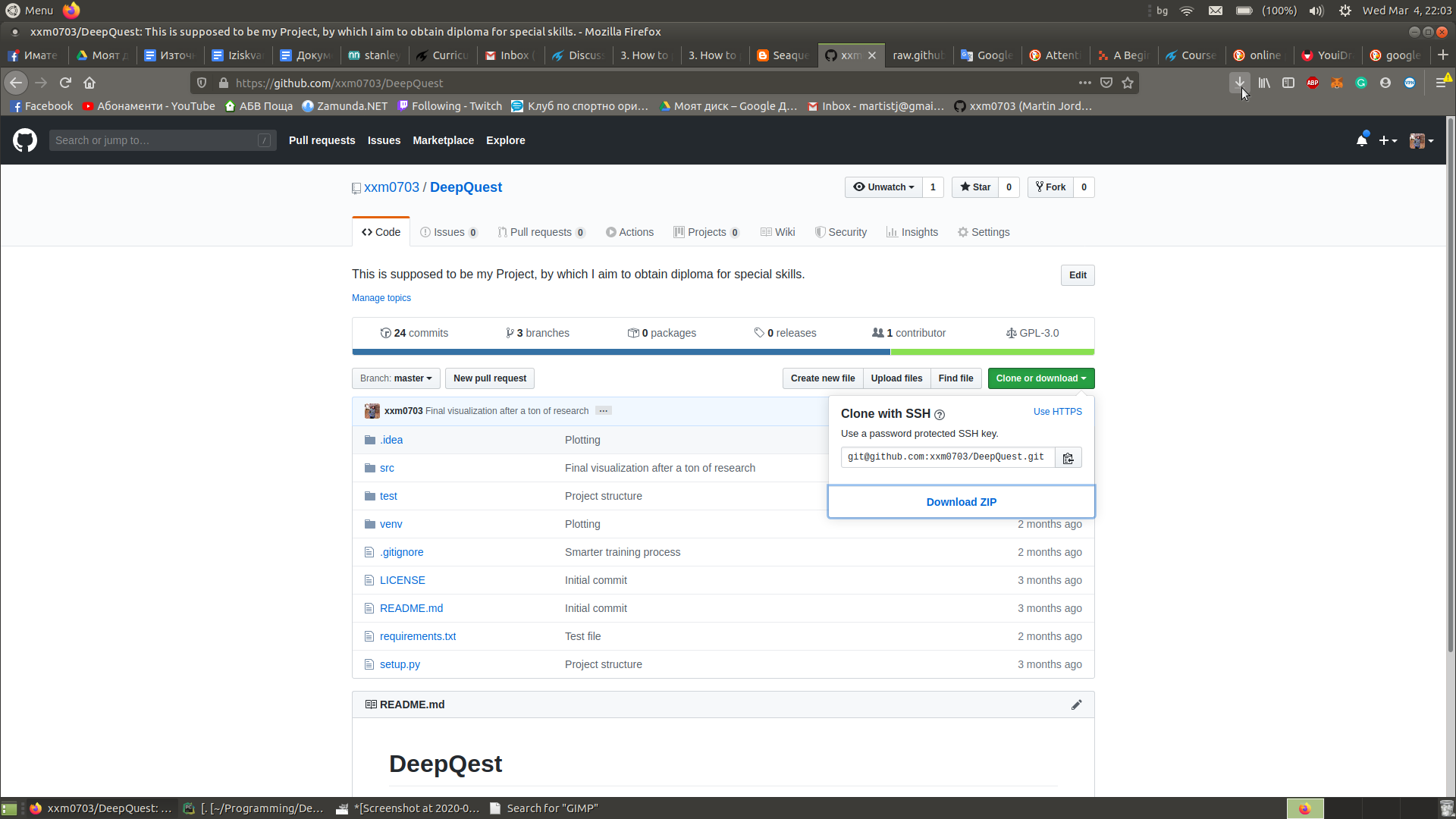
Преди архивът да започне своето изтегляне, браузърът представя екран за избор на директория в която да бъде записан файлът или с каква програма да бъде отворен (фиг. 4.3)



### фиг. 4.3 Прозорец за записване или отваряне на архива

### 

След като изборът е направен, архивът ще започне да се сваля (фиг 4.4)



### 

### фиг. 4.4 Поле за свалящи се файлове - Chrome (ляво); Firefox (дясно)

## Сваляне чрез терминал с git

Потребителят трябва първо да копира адреса на хранилището (= адрес на пректа в GitHub + “.git”). След като адресът е копиран в буфера, потребителят трябва да се увери че има инсталиран **git** (linux)

или **git CLI** (Windows), aко това не е факт, то той може да бъде инсталиран с командата:

|  |
| --- |
| $ sudo apt-get install git |

След като горната стъпка е изпълнена, следва “клонирането” на същинския проект, това става със следната команда:

|  |
| --- |
| **$ git clone <url-address>** |

## Инсталация на виртуална среда

Ако потребителят няма инсталирана версия на **virtual environment**, то тя може да бъде изтеглена и инсталирана с помощта на пакетният мениджър под Ubuntu - **apt**, командата е следната:

|  |
| --- |
| **$ sudo apt-get install python-virtualenv** |

Важно е клиентът да създаде (или използва, ако вече има създадена), виртуална среда, преди да инсталира необходимите библиотеки**.** Това става с една от следните команди:

|  |
| --- |
| **$ virtualenv <DIR>** |

**-** При създаване на нова среда

|  |
| --- |
| **$ . ./DIR/bin/activate** |

**-** При активация на съществуваща среда

## Инсталация и изисквания

Потребителят трябва да има инсталирана версия на Python, препоръчително е версията да бъде Python3.6, но и по-стари версии биха работили стига да не са Python2.\*. Препоръчително е, машината на която ще бъдат изпълнявани скриптовете да притежава поне 8GB RAM, в случай че ще бъде тренирана, и 4GB ако единствено ще бъде демонстриран.

След като проектът бъде свален, по някой от двата посочени начин, задължително трябва да бъдат инсталирани необходимите модули за пълното функциониране на скриптовете. С цел улеснение на клиента, всички необходими библиотеки и техните версии са описани в файл на име **requirements.txt**, с помощта на **Python Package Installer (pip)**, те могат да бъдат инсталирани по следния начин:

|  |
| --- |
| **$ pip install -r /path/to/requirements.txt** |

## 

## Изпълнение на програмата

Когато всички предни стъпки са изпълнение, клиентът може да стартира изпълнението на някой от скриптовете. При желание да види постижението на алгоритъма, файлът предвиден за изпълнение е с име **visualizer.py,** за да бъде изпълнен той, трябва потребителят да влезе в директорията **src**:

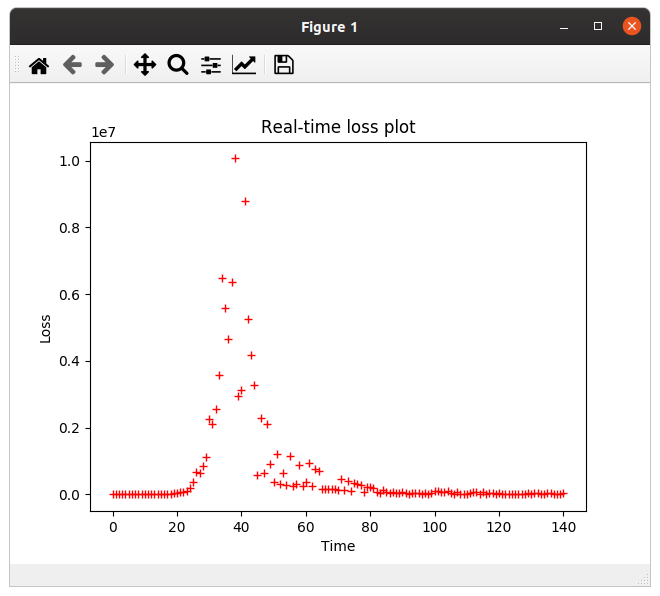
|  |
| --- |
| $ cd ./src $ python visualizer.py |

На екранът на потребителя трябва да се отвори прозорец в който да започне визуална симулация на играта, стандартно екранът изглежда, като на фиг 4.5

### фиг. 4.5 Стандартен изглед към симулацията

A в случай че потребителят иска да развива алгоритъма, файлът, който трябва да бъде изпълнен в този случай е под името **trainer.py** и процедурата по неговото изпълнение е сходна:

|  |
| --- |
| $ cd ./src $ python trainer.py |

В този случай на екранът на потребителят ще се покаже прозорец генериращ графика на ефективността на алгоритъма, в реално време(фиг.4.6). По абсцисната ос се развива времето, при всяко добавяне на точка тя се измества с 1-ца по х оста. На ординатата се намира, мярката за отклонение от потенциално най-доброто решение, по този начин колкото по-добре се представя алгоритъмът, толкова по-близо до абсцисата ще се намират точките.

### фиг. 4.6 Графика на ефективността на алгоритъма

# Заключение

Като последна бележка към този проект, може да се спомене, че желаният резултат е постигнат. Невронна мрежа бе проектирана и тествана множество пъти докато бъдат намерени най-подходящите параметри. Допълнително от това е съставен клас спомагащ визуалното представяне на ефективността на алгоритъма в реално време.

При последвало бъдещо развитие могат да бъдат имплементирани две основни функционалности, при намиране на повече компютърна мощ, би било възможно подаването на цветна картина като входен параметър (във формат RGB). Ако концепцията за ResNet бъде доизяснена в бъдеще може да бъде добавена отново, като в този случай би се очаквало значително подобрение в представянето на алгоритъма.

# Източници

1. <http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf>
2. <https://towardsdatascience.com/self-learning-ai-agents-part-i-markov-decision-processes-baf6b8fc4c5f>
3. <https://keon.io/deep-q-learning/>
4. <https://keras.io/>
5. <https://numpy.org/doc/1.17/user/quickstart.html>
6. <https://stackoverflow.com/questions/687261/converting-rgb-to-grayscale-intensity>
7. <https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673>
8. <https://missinglink.ai/guides/keras/keras-conv2d-working-cnn-2d-convolutions-keras/>
9. <https://missinglink.ai/guides/keras/using-keras-flatten-operation-cnn-models-code-examples/>
10. <https://www.retrogames.cz/play_221-Atari2600.php>
11. <https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9>
12. <https://arxiv.org/pdf/1312.5602.pdf>
13. <http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf>
14. <https://natureofcode.com/book/chapter-10-neural-networks/>
15. <https://cs231n.github.io/convolutional-networks/>
16. <https://www.wired.com/2016/04/openai-elon-musk-sam-altman-plan-to-set-artificial-intelligence-free/>
17. <https://stackoverflow.com/a/15720891>
18. <https://arxiv.org/pdf/1511.04119.pdf>
19. <https://arxiv.org/pdf/1512.03385.pdf>

# Приложение

Github хранилище: <https://github.com/xxm0703/DeepQuest>

# CD:

# Съдържание

[**Увод**](#_4tlkncari9x2) **1**

[**Проучване**](#_bf9mmodfss8v) **3**

[Neuroevolution through Augmenting Topologies](#_vetmwro4z4ql) 3

[PyTorch framework](#_psintb2etix8) 6

[SciKit-learn framework](#_bj7tnksb50iz) 8

[Pandas library](#_g9f30th3o4rs) 10

[Cartpole Deep Q-Learning](#_ua8tcdeozcyg) 12

[NeuroEvolution with MarI/O](#_xhbabhrn9xm3) 13

[**Изисквания**](#_bd9zy14atr50) **15**

[Основа на проектът](#_lcromwfx5rbg) 16

[Изисквания към приложението](#_tqsfysoi3bh8) 19

[Съображения за избор на програмни средства и развойната среда](#_sqirdcl0r8mk) 21

[GitHub система за контрол и управление на версиите](#_p2b6qs253c8o) 22

[Visual Studio Code текстов редактор](#_qrkasot4fr5f) 22

[Keras библиотека за невронни мрежи](#_vv7c99m0yvru) 22

[Gym библиотека за симулация на аркадни игри](#_5hbihloq7ugg) 23

[**Програмна реализация на “DeepQuest”**](#_t6qxg4fufqxl) **24**

[Дизайн на алгоритъма](#_nevptmndd031) 24

[Невронни мрежи](#_q8ipwemykhse) 25

[Конволюционен слой](#_ufgynxbu3zf8) 29

[Пуулинг](#_fz9ep9epo46) 32

[Визуализиране на стойности](#_bphyv1l3hk5v) 34

[Deep Q-Learning](#_1qnuhrakbsbj) 35

[Attention mechanism](#_24zn4i67mxb) 37

[Residual neural network](#_cexd6crdn3ke) 38

[Същинска разработка](#_q0iwv13pnjl7) 39

[**Ръководство на потребителя**](#_b3nylj2bgb0w) **45**

[Изтегляне на проекта](#_s9qb1osvp2yc) 45

[Сваляне през браузъра](#_1vof5ebhubhd) 45

[Сваляне чрез терминал с git](#_ec7v0hhidpd5) 49

[Инсталация на виртуална среда](#_e6xs8gd0f35t) 49

[Инсталация и изисквания](#_ctbhmpo59u1n) 50

[Изпълнение на програмата](#_mzuzdsrva81d) 51

[**Заключение**](#_gpgo6z927s28) **54**

[**Източници**](#_m52bpmqk3uzi) **55**

[**Приложение**](#_y3kshaip6m3v) **57**

[**Съдържание**](#_3nufchcqsaud) **58**