Увод

Целта на този дипломен проект е използването на дълбоки невронни мрежи в предвиждането на нуклеотидни секвенции, чувствителни към серията от транскрипционни фактори ZNF. Ще бъдат проведени експерименти върху генерирани и реални данни с един и множество транскрипционни фактори за установяването на подходящи архитектури за невронни мрежи, свързани с подобен тип задачи.

В хода на този дипломен проект потребителят ще бъде запознат с концепцията за машинното обучение и разликите му с класическото програмиране, историята на машинното обучение и мястото на дълбокото машинно обучение в него. Допълнително ще бъде разгледана и математическата страна на дълбокото машинно обучение, типични проблеми, с които разработчиците на невронни мрежи се сблъскват всекидневно и отражението им върху конкретната задача, която този дипломен проект цели да реши.

Накрая потребителят ще разбере кои са стъпките, които трябва да последва при желание да се сдобие с множеството от данни, файловете с проведените експерименти и най-вече файловете, съдържащи самите модели на невронни мрежи.

1. Въведение в невронните мрежи

В тази глава потребителят ще бъде запознат с концепцията за машинното обучение и разликите му с класическото програмиране, историята на машинното обучение и мястото на дълбокото машинно обучение в него. Ще бъде разгледана математиката, стояща зад невронните мрежи, видове невронни мрежи, проблемите, с които разработчиците в сферата на дълбокото машинно обучение с сблъскват и намерените досега решения на тези проблеми.

1.1. История на дълбокото машинно обучение

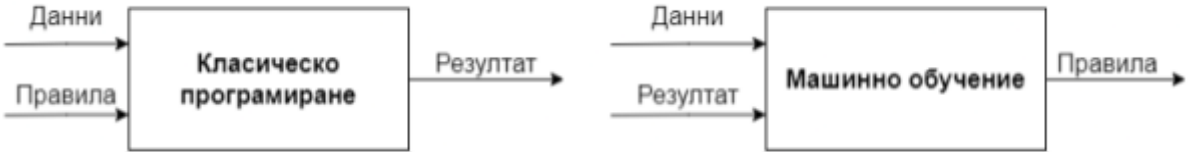
За да се разбере повече за това какво всъщност представляват невронните мрежи и защо са възникнали ще се наложи историята на сферата на изкуствения интелект да бъде проследена още от самото ѝ начало през 50-те години на миналия век, когато неколцина пионери в нововъзникналата област на компютърните науки започват да си задават въпроса “А можем ли да накараме компютрите да мислят?” - въпрос с множество разклонения, който бива изследван и до ден днешен[1].

Ако периода на развитие на сферата на изкуствения интелект (ИИ) може да се раздели на подпериоди, то тези подпериоди биха били два - от 1956 г. (когато по идея на Джон Маккарти се организира първия летен семинар на тази тематика) до към средата на 80-те години на миналия век и от средата на 80–те години до днес[1].

По време на първия период на развитие на изкуствения интелект в дебелите книги човек не би могъл да намери никъде думите “обучение” или

“трениране” - думи, характерни за съвременните методи за съставяне на ИИ модели. Това е така, защото по това време учените, занимаващи се в тази сфера са смятали, че биха могли да имитират човешката мисъл с набор от точно определени правила, които да обработват и манипулират информация, съхранена в бази от данни. Това е била и доминиращата парадигма за създаване на т.нар. експертни системи, а исторически днес този подход за създаване на ИИ се назовава “символично ИИ”[1].

По време на втория период започва вече същинското развитие на машинното обучение. За да се придобие по-добра представа за това в какво се състои процесът на машинното обучение е възможно той да бъде сравнен с процеса на класическото програмиране (фиг. 1.1).



Фигура 1.1. Разлика между класическото програмиране и машинното обучение

Докато класическото програмиране се състои в намирането на резултат при определен вход и спрямо определени правила, то при машинното обучение се търсят определени правила и зависимости, спрямо които входните данни съответстват на изходните такива. Разликите не спират дотук. Процесът на намиране на правила и зависимости в машинното обучение е итеративен, (моделът се “обучава”/”тренира”), докато при класическото програмиране алгоритмите биват изрично програмирани да извършват дадена задача. Всъщност, машинното обучение изисква три основни неща: входни данни (няма особени ограничения по какъв начин трябва да са репрезентирани - може входните данни да са в табличен вид, изображения, текст, звукозапис и др.), примери за очаквани резултати (например субтитри за видео, подадено на входа, или пък очаквана цена на

автомобил спрямо техническите и класификации) и метод за оценка на начина на представяне на алгоритъма. И точно в това се състои процесът на обучение на алгоритъма - досегашното му представяне се оценява и оценката се връща като обратна връзка, спрямо която алгоритъмът да регулира работата си[1].

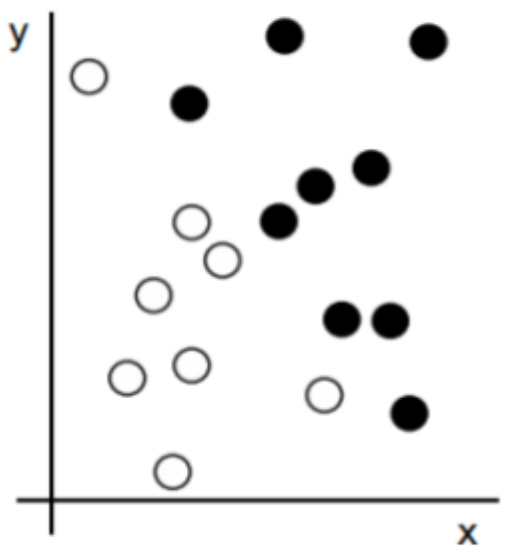
Въпреки че машинното обучение процъфтява чак през 90-те години на миналия век, то се превръща в най-разпространената и предпочитана под-сфера на ИИ. Причините са много и най-разнообразни - развитието на интернет, достъпа до повече данни, разработката на по-бърз хардуер, добре развитата сфера на статистическите науки и др. Въпреки наличието на статистика и математически методи в сферата на машинното обучение, то си остава основно инженерна специалност, която има изцяло практическа насоченост, целяща основно емпирични заключения и зависеща изключително много от напредъка на наличния хардуер и софтуерните продукти.

1.2. Разлики между стандартното и дълбокото машинно обучение

След като накратко се запознахме с историята на сферата на изкуствения интелект, развитието и основните разлики между машинното обучение и класическото програмиране, следва да се уточни какво представляват невронните мрежи и кое е довело до създаването и развитието им като концепция.

Невронните мрежи представляват под-сфера на машинното обучение, която е още наричана “дълбоко машинно обучение”. Терминът “дълбоко машинно обучение” подсказва, че методите на стандартното машинно

обучение са станали недостатъчни за целите на индустрията в определен момент след 90-те години на миналия век. А всъщност задачите, които изпълнява стандартното машинното обучение, могат да бъдат групирани в две основни групи - класификационни (моделът трябва да намери подходящи правила, според които да може да разделя и групира данните според определни признаци) и регресионни (намирането на функционални зависимости между две или повече случайни величини). По това време наистина популярни методи за машинно обучение са т.нар. "методи на ядрото"[1]. На фиг. 1.2 са изобразени точки, принадлежащи на дадено двумерно пространство.



Фигура 1.2. Примерно представяне на данни в двумерно пространство. Белите точки представляват клас 1, а черните - клас 2.

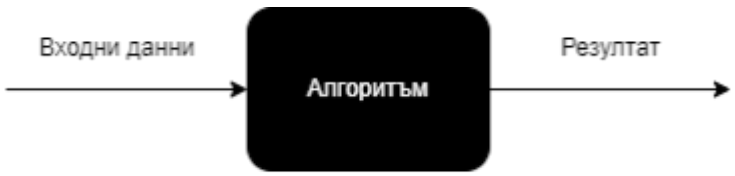
Ако задачата е класификационна, разработчикът на модел за машинно обучение би искал да намери начин да раздели белите точки от черните посредством някакво правило (в случая функция, описваща крива линия). Но колкото повече данни са налични, толкова по-трудно ще бъде разделянето на данните, представени във фиг. 1.2.

Тук се намесват методите на ядрото, които се основават на теоремата на Томас Кавър[2], че линейно трудно разделими или неразделими данни

могат с голяма вероятност да бъдат преобразувани в линейно разделими такива, като биват проектирани в многомерно пространство. Най-известния от тези методи е методът на опорните вектори, разработен в лабораториите “Бел” през 1990 г. и публикуван през 1995 г. Методът на опорните вектори дава много високи резултати и работи безотказно, като представя данните в многомерно пространство с краен брой операции. Би следвало тогава човек да се запита “Защо учените и инженерите прибягват до идеята за дълбокото машинно обучение, след като съществува перфектния алгоритъм от стандартното такова?”. Отговорът е следният: сложността на метода на опорните вектори варира от $O(N^2)$ до $O(N^3)$ - тоест бързината на този метод не мащабира никак добре спрямо броя данни, които му се подават. Освен това възможностите на най-добрия наличен хардуер тогава, през 90-те години, са с хиляди пъти по-малки в сравнение с дори най-лошия наличен хардуер днес.

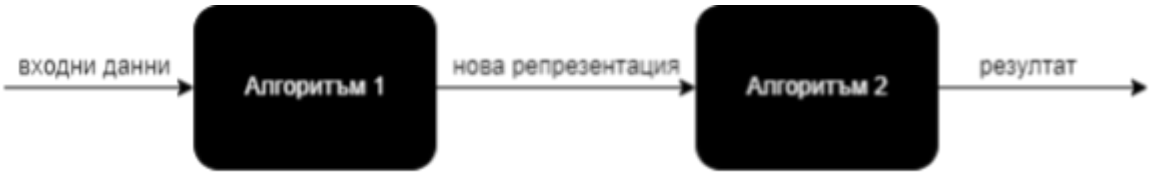
1.3. Дълбоко машинно обучение

Следва да бъде изяснено какво представляват дълбокото машинно обучение и невронните мрежи. Нека за момент алгоритмите в стандартното машинно обучение бъдат представени като черна кутия (фиг. 1.3).



Фигура 1.3. Алгоритъм за машинно обучение, представен като черна кутия.

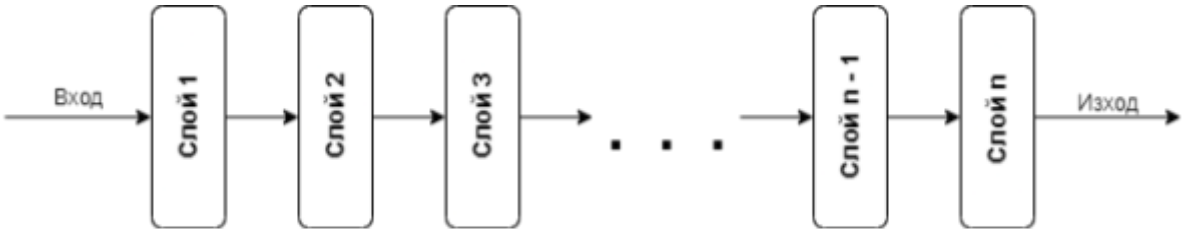
Вътре в черната кутия данните претърпяват някакъв вид математическа трансформация, при което получаваме резултат. Може да се каже, че резултатът е и определен вид репрезентация на входните данни. Дали тогава е възможно тази репрезентация да бъде препредадена като входни данни на следваща черна кутия, като например във фиг. 1.4?



Фигура 1.4. Изходните данни от първата черна кутия влизат като входни такива във втората.

Няма такова правило, което да забранява последователното свързване на такива черни кутии. Във втората черна кутия може да се използва коренно различен алгоритъм за математическа трансформация и да се получи коренно различна репрезентация. В сферата на дълбокото машинно обучение е прието тези “черни кутии” да се наричат слоеве.

Следвайки тази логика, добавяйки нови и нови слоеве към тази последователност дава възможност да се изглади последователност от представяния на данните. Това представлява в същността си дълбокото машинно обучение - търсене на последователност от репрезентации на входните данни, които да стават все по-смислени и по-смислени (фиг. 1.5).

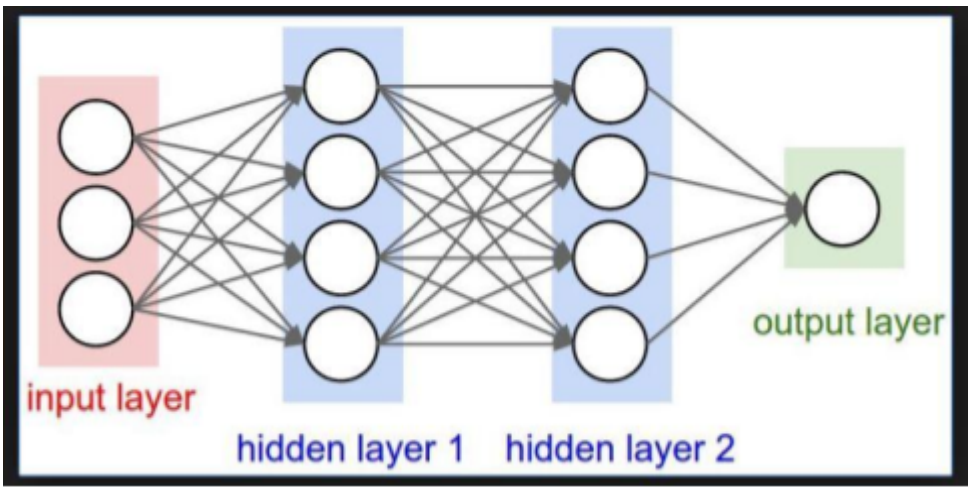


Фигура 1.5.

След като вече ни е известно как се постигна дълбокото машинно обучение, е време да разгледаме по-подробно детайлите**.**

1.4. Слоевете в една невронна мрежа

Досега слоевете бяха разглеждани доста абстрактно, като за тях бе споменато само, че извършват определени математически операции. Един слой съдържа множество възли, които се наричат още неврони, перцептрони или единици[3]. Специфичното при невронните мрежи е, че невроните в един слой нямат връзки помежду си. Всеки неврон от даден слой има връзки единствено с невроните от предходния слой и следващия такъв. Слоевете може да бъдат разделени на три групи: входни, изходни и скрити. Входните слоеве подават данните във вид, удобен за обработка, невроните от входните слоеве нямат връзки с предишни слоеве. Изходните слоеве извеждат краен резултат, техните неврони нямат връзки със следващ слой. Останалите слоеве, намиращи се между входните и изходните, се наричат скрити слоеве. Те имат връзка както и с предходен, така и със следващ слой (фиг. 1.6).



Фигура 1.6. Стандартна структура на една невронна мрежа.

Невроните от скритите слоеве и изходния такъв са тези, които извършват математическите операции с данните. И всъщност от тях зависи какъв вид трансформация ще се прилага върху данните с всеки следващ слой. Самите математически операции се различават от тези, използвани в стандартното машинно обучение. Тъй като в стандартното машинно обучение често разработчиците разполагат само с един слой за трансформация на данните, а в повечето случаи зависимостите между входните и изходните данни не са единствено линейни, се налага да да се прилагат сравнително по-сложни и по-бавни алгоритми.

Тъй като идеята на дълбокото машинно обучение е да съществуват последователности от трансформации върху данните, процесът може да бъде разбит на повече, но по-прости стъпки. За основа се използва линейната функция: $f(x) = ax + b$. Но ако всеки неврон в скритите слоеве използваше само линейната функция за операция, то накрая би се образувала съставна линейна функция, което не помага особено в търсенето на нелинейни зависимости. Поради тази причина като втора стъпка при изчислението невроните подават резултата от линейната функция като параметър на нелинейна.

1.5. Основни математически операции и начин на трениране

Както бе споменато още 1.1, в началото всеки алгоритъм за машинно обучение трябва да открие самостоятелно определени правила, по които определени входни данни да дават точно определени изходи. Невронната мрежа не прави изключение, а процесът на намиране на тези правила е сходен с възпитанието в ранна детска възраст - гради се на базата на принципа "награда-наказание". Обучението е итеративно, като във всяка стъпка се повтарят 2 етапа - forward propagation и backpropagation[3].

Forward propagation е първата стъпка от обучението на една невронна мрежа. Както бе споменато в предната подглава, основата е линейната функция $f(x) = ax + b$. Линейната функция притежава грубо казано 3 числови стойности - една променлива, един параметър и свободния член. Резултатът $f(x)$ се наричат функционална стойност. Променливата и функционалната стойност принадлежат на две отделни множества и се съпоставят посредством някакво правило $f(x)$. Оттук и от идеята на машинното обучение следва, че невронната мрежа ще търси подходящи параметри, които най-точно биха определили това правило - в случая на линейната функция това са старшият коефициент a и свободният член b.

Следва да бъде уточнено как точно се намират тези параметри. Първоначално те се инициализират произволно спрямо някакво разпределение. Изключително важно е да се спомене, че параметрите за всяка единица в невронната мрежа трябва да се различават от останалите, защото в противен случай всички единици в невронната мрежа ще правят абсолютно едни и същи изчисления, при което няма да хванат отделните

зависимости между входните данни. С цел абстракция засега линейната функция ще бъде ограничена до 1 входен параметър, като по-нататък ще бъде обяснено как се извършват изчисленията за функции на повече от 1 променлива. При входни данни x_0 на първия слой се получава изходният резултат от (1.1).

$$y_1 = a_1 x_0 + b_1 \quad \textbf{(1.1)}$$

От фигури 1.5 става ясно, че изходните данни за n-тия слой ще бъдат входни такива за n+1-ия такъв, откъдето следва (1.2).

$$y_2 = a_2 x_1 + b_2 = a_2(a_1 x_0 + b_1) + b_2 \quad \textbf{(1.2)}$$

Тази операция се повтаря за всеки един слой, докато накрая не се получи краен резултат $\hat{y}$. Това представлява и процесът forward propagation. Следващата логична стъпка е $\hat{y}$ да бъде сравнен с реалната стойност y. Но само със сравнение няма да се постигне нищо особено, трябва метрика или правило, които да казват доколко $\hat{y}$ се различава от y. Понеже в дадения пример функционалните стойности са непрекъснати ($y \in R$), то е възможно да се вземе разстоянието между $\hat{y}$ и y, като има разновидности на тези правила, които взимат модул или квадрат от разстоянието. Такива правила/функции се наричат функции на загубата (loss/cost functions). Нека функцията на загубата бъде подобна на тази от (1.3).

$$L(\hat{y};\, y) = \frac{1}{n} \sum_{i=1} (\hat{y}_i - y_i)^2 \text{, където } n \text{ е общият брой записи} \quad \textbf{(1.3)}$$

След като грешката между направените от нас предсказания и реалните стойности е изчислена и налична е време коефициентите за всеки един неврон във всеки слой да бъдат поправени. Все пак за да съществува възможно най-точен модел трябва грешката между предсказаните и реалните резултати да е минимална. Това е и стъпката, която се нарича back-propagation. Но как точно биха се променили параметрите в линейните функции?

Последното изречение съдържа две ключови думи: функция и промяна. А както бе споменато в началото, процесът е итеративен, тоест промяната е постепенна. На помощ идват производните на функциите, които всъщност показват с каква скорост се променят функционалните стойности спрямо входните параметри. Тъй като по-рано стана ясно, че крайния резултат намираме чрез сложни функции, то в сила идва правилото за диференциране на сложни функции, което гласи, че ако

$$h(x) = f(g(x)),$$

то

$$h'(x) = f'(g(x))g'(x)$$

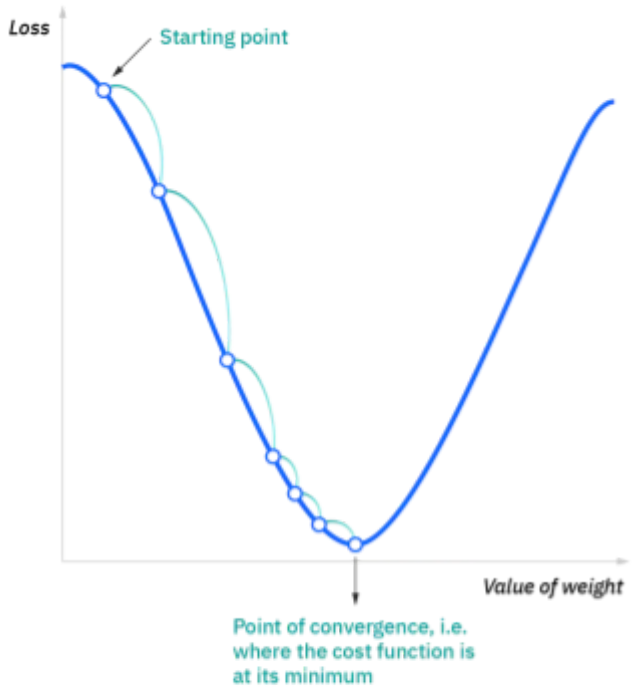
Засега това е единствения възможен начин, по който е възможно стойностите на параметрите за всички слоеве в невронната мрежа да бъдат обновени. Но процесът на намиране на производните не спира само до диференциране на сложни функции. Както вече бе споменато, една линейна функция има два параметъра: старшият коефициент a и свободният член b. За всеки слой от невронната мрежа трябва да бъдат намерени частните производни за всеки един от тези коефициенти, в случая два параметъра. Промяната в резултата на линейната функция в зависимост от двата параметъра може да се разглежда по този начин:

При промяна в a: $f(x) = ax + (\text{нещо, независещо от а})$

При промяна в b: $f(x) = (\text{нещо, независещо от } b) + b$

Така се изчисляват частните производни спрямо съответно a и b - фиксира се точно определен параметър, а останалото се счита за константа. Така се изчисляват $\frac{df}{da}$ и $\frac{df}{db}$. Както бе споменато обаче, крайния резултат е получен вследствие на композиране на множество линейни функции. След като са намерени производните на последната функция по веригата е редно да се продължи назад, за да се коригират и параметрите в останалите слоеве. Комбинирайки диференцирането на сложни функции с частните производни се получава, че $\frac{df}{dc} = \frac{df}{dg}\frac{dg}{dc}$ и $\frac{df}{dh} = \frac{df}{dg}\frac{dg}{dh}$, където с c и h са означени старшият коефициент и свободният член на следващата по веригата обратно линейна функция, която ще бъде условно означена с $g(x)$. По същият начин се изчисляват производните за всички останали параметри във всеки от слоевете.

Производните на всеки от параметрите са вече готови и е известно с каква скорост се променя всеки от тях. С други думи градиентите на функциите в невронната мрежа са били изчислени. Първоначалната цел бе да се намали доколкото е възможно грешката между предсказаните от модела резултати и реалните такива. Най-известният такъв алгоритъм е т.нар градиентно спускане (фиг. 1.7), който има за цел да намери локален или глобален минимум на дадена функция. Намирайки минимум на функция моделът намалява значително грешката между предсказани и реални резултати, което повишава неговата точност, което е основната цел на един разработчик на невронни мрежи.



Фигура 1.7. Градиентно спускане

Името на алгоритъма подсказва, че при търсенето на минимум той ще се "спуска" итеративно по склоновете на функцията, докато не намери минимум. За да разбере обаче накъде да се спуска се налага той да има представа от наклоните на функцията в дадена точка. Тук влизат и изчислените преди това градиенти. Ако градиентите показват в кои посоки функцията нараства, то ако при взимане на техните противоположни стойности моделът ще установи в кои посоки функцията намалява. Това представлява и обновяването на стойностите на параметрите на функцията. За всяка итерация на градиентното спускане получаваме (1.4). Параметърът α регулира скоростта при градиентното спускане (от англ. learning rate)

$$b := -\alpha\frac{df}{db}\ ;\ a := -\alpha\frac{df}{da} \qquad \textbf{(1.4)}$$

В сферата на машинното обучение случаите, в които моделът разполага само с една променлива на входа са единични и човек много рядко

може да срещне такива. Действително за всеки неврон от даден слой те представляват нещо подобно на (1.5). За удобство и с цел генерализация изчисленията за даден слой се представят като в (1.6).

$$f(x) = a_1x_1 + a_2x_2 + \ldots + a_{n-1}x_{n-1} + a_nx_n + b \qquad \textbf{(1.5)}$$

$$z = W_nX_n + b_n \qquad \textbf{(1.6)}$$

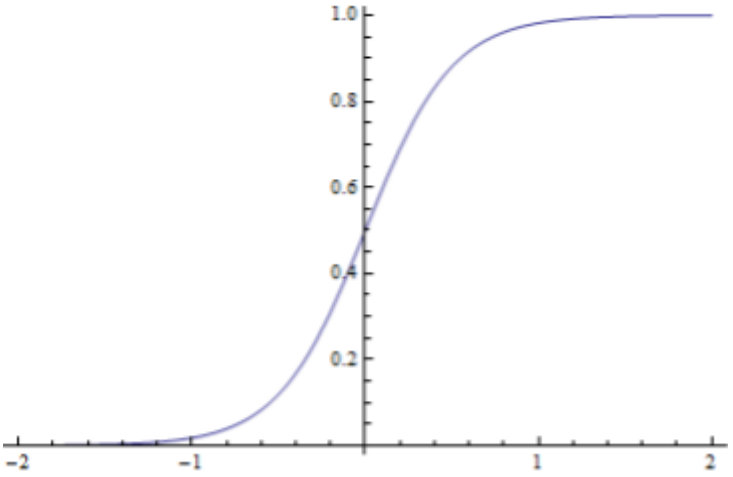
В (1.6) индексът n отбелязва поредността на слоя, W представлява матрица, съдържащи всички параметри на линейната функция за всяка единица от слоя, X е матрица, съдържаща всички входни данни на линейната функция за всяка единица от слоя, а b е вектор, съдържащ свободните членове за всеки неврон от слоя. Респективно, производните спрямо W и b се отбелязват с $\frac{df}{dW}$ и $\frac{df}{db}$.

1.6. Активационни функции

Както по-рано бе споменато, композирането на линейни функции не би помогнало особено много при намирането на нелинейни връзки между данните. Затова резултата от всяка линейна функция се подава като параметър на нелинейна такава. В света на машинното обучение се използват неголям брой нелинейни функции и техни разновидности, които обаче вършат много добра работа и дават добри резултати.

Първата такава е сигмоидалната функция (от англ. sigmoid). Бележи се най-често с $\sigma(x)$ и нейното правило е $\sigma(x) = \frac{1}{1+e^{-x}}$ (фиг 1.8). Резултатите, които тя дава, са в интервал от 0 до 1. Като резултат 0 и 1 могат да означават

много неща - като например наличие или отсъствие на дадено свойство, качество, зависимост и т.н. Ако връзките на невроните между слоевете представляват възможните "пътечки", по които могат да преминат входните данни по пътя си към резултата, то с $\sigma(x)$ моделът де факто би могъл да регулира тези пътечки - при резултат 1 той би "казал", че даден неврон се е "събудил" или "активирал", т.е засечено е наличие на дадено свойство или качество.

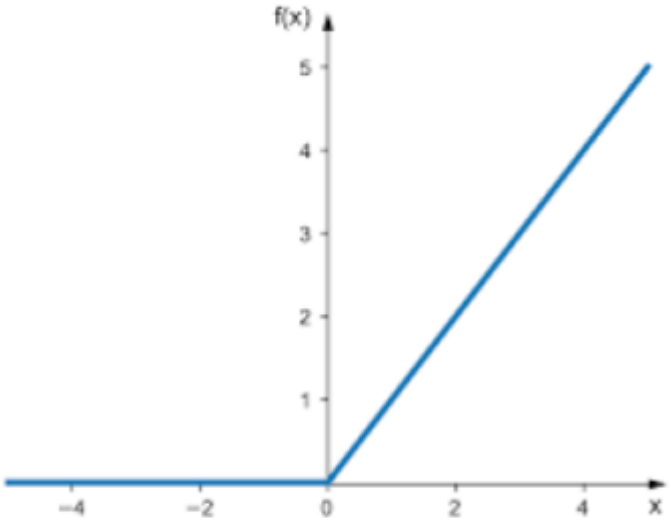


Фигура 1.8. Графика на сигмоидалната функция.

Подобно на останалите функции, активационните такива също минават през процеса на градиентно спускане, т.е. изчисляват се и техните производни. Един основен недостатък на сигмоидалната функция е тези "плата", които тя образува при подадени много големи или много малки числа. В тези плата градиентите са равни или много близки до 0, което прави итерацията през тях мъчителна и времеемка. Затова сигмоидна функция се използва само на определени места, като например изходния слой на невронната мрежа.

Алтернатива на сигмоидалната функция се явява ReLU (от англ. Rectified Linear Unit)[5], както и нейните разновидности. ReLU няма специфична нотация, но в същността си представлява $f(x) = max(0, x)$ -

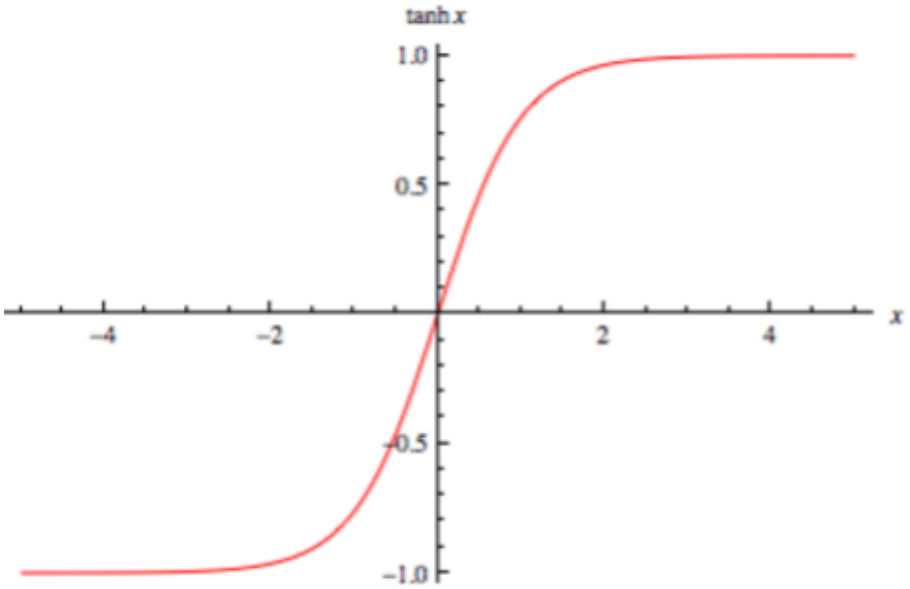
т.е за стойности, по-малки от 0, изходът от ReLU е 0, а за стойности, по-големи от нула, резултатът е самата стойност (фиг. 1.9). Доказано е, че ReLU и $\sigma(x)$ дават подобни резултати, като ReLU решава част от проблемите с градиентното спускане - градиентът при $x > 0$ е равен на 1 и е константа. Проблемът с градиент, равен на нула, при отрицателните числа, си остава, затова са измислени разновидности на ReLU, които също активно се използват. Такива са например Leaky ReLU, ELU и Parameterized ReLU. При Leaky ReLU $f(x) = max(\alpha x, x)$, където $\alpha << 1$. При ELU $f(x) = max(\alpha(e^x - 1), x)$, където $\alpha << 1$. При Parameterized ReLU $f(x) = max(\alpha, x)$, където отново $\alpha < 1$. И в трите случая α се счита за хипер параметър и в повечето случаи е добре да се настройва допълнително.



Фигура 1.9. Графика на ReLU функцията.

Друга често използвана активационна функция е хиперболичния тангенс. Често се анотира като $tanh(x)$ и нейното правило е $tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}$ (фиг. 1.10). Резултатите, които тя дава, са в интервал от -1 до 1. Тази активационна функция се среща изключително често при

рекурентните невронни мрежи, за които ще споменем по-нататък. Подобно на сигмоидната такава, хиперболичният тангенс може да отразява две състояния - наличие и липса, или пък отношение - положително и отрицателно такова. Също като сигмоидалната функция, $tanh(x)$ страда от нулеви градиенти при подадени твърде големи или твърде малки числа.



Фигура 1.10. Графика на функцията на хиперболичният тангенс

Досега изброените активационни функции вършат много добра работа при т.нар. бинарна/двоична класификация, където изходният резултат може да бъде един от две възможни стойности за дискретна променлива.

Но когато възможните стойности за дискретна променлива станат повече от две, тези активационни функции не се справят толкова добре. Активационна функция, която се справя добре с тази задача, е softmax. Често softmax се анотира по подобен начин на сигмоидалната функция - с $\sigma(X)$. За разлика от останалите функции, softmax приема вектор от стойности - големината на вектора е равна на броя възможни класове. Softmax се прилага итеративно върху всяка една стойност от вектора, като $\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$, т.е.

изчислява се $f(x) = e^x$ и резултата се нормализира със сбора на всички експоненти във вектора. Резултатът от softmax е вектор, в който всяка стойност представлява резултат в интервала [0;1], което всъщност съответства на вероятността резултатът да принадлежи на даден клас. Сборът на стойностите във вектора задължително е равен на 1.

1.7. Често срещани проблеми при дълбокото машинно обучение

Проблемите, които "тровят" дълбокото машинно обучение, са небезизвестни за всеки един, занимаващ се с моделиране и трениране на невронни мрежи. Част от тези проблеми са наследени от стандартното машинно обучение, част от тях са специфични за тази под сфера на изкуствения интелект. И въпреки че в рамките на последните 10-15 години общността стигна до оптимизации и частични решения, тези проблеми остават в по-голяма или по-малка форма и са нещо, с което един разработчик на модели за дълбоко машинно обучение трябва да свикне[4].

Един от тези проблеми се нарича bias-variance tradeoff и той съществува и в стандартното машинно обучение. Първо обаче трябва да се обясни какво са bias и variance. Bias напълно безпроблемно е възможно да се обвърже с термина пристрастие или наклонност, като пристрастието представлява осредната разлика между резултатите, които ни дава моделът, и действителните такива. Колкото по-голяма е разликата, толкова по-пристрастен е моделът. Или с други думи, моделът опростява прекалено много представата си за данните и не се интересува от тях. При необходимост от определяне на нови входни данни, несрещани досега,

моделът ще изкара резултат, който до голяма степен е изграден на базата на пристрастието на модела и който се различава съществено от реалния такъв.

Ако пристрастието на модела представлява едната крайност, то голямата вариация/дисперсия (от англ. variance) представлява другата крайност. Накратко моделът обръща твърде много внимание на данните дотолкова, че ги научава наизуст и не успява да генерализира. При подаване на нови данни моделът не "знае" какво да прави с новите входни данни и, подобно на един пристрастен модел, произвежда грешни резултати.

Често при дълбокото машинно обучение термините bias и variance се срещат и като underfitting и overfitting. Като основна причина за underfitting може да се счита размера на невронната мрежа и по-точно - твърде малка невронна мрежа. Когато се ограничава броя неврони във всеки слой, както и броя слоеве, ефективно се намалява капацитета на невронната мрежа да открива зависимости в данните. Това резултира в модел, който не може да научи нищо за данните си и придобива наклонности.

Overfitting е по-често срещаният от двата проблема що се отнася за дълбоко машинно обучение. По-честите причини за overfitting за както твърде параметризирана невронна мрежа, така и недостига на трениращи данни. При наличието на твърде много параметри невронната мрежа научава твърде много зависимости, някои от които са твърде специфични и не могат да се отнесат към генералното поведение на данните. При подаване на нови данни, които спазват генералното поведение, но се отнасят различно помежду си, твърде големият модел не открива нищо познато сред тях и не може да изкара адекватно предсказание.

Редно е да се отбележи, че overfitting може да се постигне дори и с малък модел. При наличие на малко записи за трениране е възможно лесно да се открият и отразят специфични зависимости между тези записи, които да се поберат в капацитета на дори и един малък модел. Но при липсата на много данни е пожелателно разработчика на модела да се замисли дали

задачата не може да бъде решена с алгоритми от стандартното машинно обучение.

Общността е стигнала до две решения, които могат до голяма степен сравнително да намалят ефектът на overfitting. Първият от тях се използва и в стандартното машинно обучение и се нарича регуляризация. В (1.3) в подглава 1.5 бе дефинирана функция на грешката, която представляваше:

$$L(\hat{y};\ y)\ =\ \frac{1}{\sum_{i=1}^{n}}(\hat{y}_i\ -\ y_i)^2$$

Регуляризацията представлява допълнителна стойност, която се добавя към функцията на загубата, а крайната нотация на функцията изглежда подобна на (1.7).

$$L(\hat{y};\ y)\ =\ (\hat{y}\ -\ y)^2\ +\ \lambda \sum_{i=1}^{n} |w_i| \qquad \textbf{(1.7)}$$

Сумата, която се добавя към стойността на функцията на загубата, е сбор на параметрите на невронната мрежа, скалирани с хипер параметъра λ. Но как помага добавянето на сбора на параметрите на невронната мрежа при справянето с overfitting? Градиентното спускане се стреми да намали възможно колкото се може повече функцията на грешката. Така написана, функцията на грешката се състои от 2 части и алгоритъмът може да намаля и двете части, за да постигне по-малка грешка. Така моделът допълнително ще коригира параметрите си, като остави само тези, които носят някаква информация, а останалите ще занули. По този начин се намалява капацитета на невронната мрежа и се предотвратява overfitting.

Регуляризацията има две норми - първа и втора норма. Първата норма е известна и като Lasso и представлява формулата по-горе. Втората норма, чиято нотация може да се забележи в (1.8), взима параметрите на невронната мрежа на квадрат и се нарича Ridge.

$$L(\widehat{y};\ y)\ =\ (\widehat{y}\ -\ y)^{2}\ +\ \lambda \sum_{i=1}^{n} w_i^2 \qquad \textbf{(1.8)}$$

Второто решение се нарича dropout[6]. Dropout представлява един вид "неутрализиране" на част от изходите на една невронна мрежа на случаен принцип. Dropout се прилага върху отделни слоеве, като единственият параметър, който се подава, е каква част от невроните да бъдат неутрализирани. Както бе споменато при активационните функции, когато на изхода си даден неврон даде резултат 0, то оттам нататък тази 0 няма да влияе на крайното решение на невронната мрежа. При dropout след единиците произведат резултат по подобен начин изкуствено част от резултатите се умножават с 0 на произволен принцип. Dropout се прилага само по време на трениране с цел елиминиране на всякакви прекалено силни корелации между неврони на съседни слоеве, което би довело до overfitting.

Друг често срещан проблем при невронните мрежи представляват т.нар. експлодиращи и чезнещи градиенти (от англ. vanishing and exploding gradients). От името става ясно, че това е проблем, който засяга градиентите (производните) на функциите при процеса back-propagation. Както бе споменато в подглава 1.5 процесът back-propagation впряга в себе си 2 основни компонента - изчисляване на градиенти и функция на грешката. Изчисляването на градиентите е базирано на две основни правила: правилата за диференциране на съставни функции и частни производни на една функция. Ако конструираният от разработчика модел притежава много слоеве, десетки слоеве, изчислението на градиентите на по-първите слоеве

става по-тежко и по неприятно. В (1.9) е изведено изчислението на примерен трети слой на примерна невронна мрежа с 20 слоя, без да се включват активационните функции.

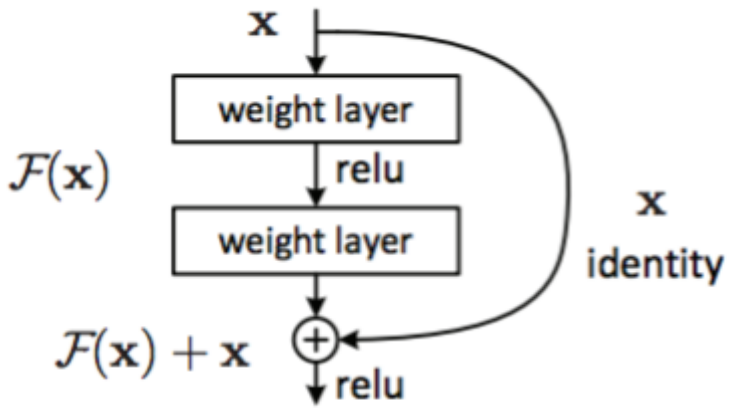
$$\frac{dL}{df_3} = \frac{df_4}{df_3}\frac{df_5}{df_4}\frac{df_6}{df_5}\frac{df_7}{df_6} \cdot \cdot \cdot \frac{df_{19}}{df_{18}}\frac{df_{20}}{df_{19}}\frac{dL}{df_{20}} \quad \textbf{(1.9)}$$

За да се изчисли тази производна е необходимо да се умножат общо 18 други, а при последователното умножение на много числа често се получава твърде малка или твърде голяма числова стойност, която допълнително затруднява от своя страна изчислението на всички останали производни по веригата. А градиентното спускане обновява параметрите на слоевете, като изважда съответните им градиенти, скалирани с коефициента $\alpha \ll 0$. Много малките градиенти (чезнещи градиенти) резултират в почти нулева корекция по съответните им коефициенти, което ще доведе до многократно забавяне в процеса на търсене на минимум на грешката. Много големите градиенти (експлодиращи градиенти) пък създават опасност от движение по функцията на грешката с твърде големи стъпки, като така моделът може изключително дълго време да се лута около даден минимум на функция, прескачайки го всеки път.

Общността е стигнала до решения, които частично решават този проблем. Първото от тези решения е по-внимателно произволно инициализиране на параметрите в началото[4]. Или по-точно - скалирането на всички параметри в слоя с определена константа. Каква константа се използва за скалиране зависи и от типа на активационната функция, която се използва. Накратко, при ReLU активации се използва най-често He[7], при който константата е равна на $\frac{2}{n}$, където n е броя променливи, влизащи като входни данни в конкретния слой, а при сигмоидалната функция и

хиперболичният тангенс се използва най-често Glorot[8], където пък константата е равна на $\sqrt{\frac{2}{n+m}}$, където n отново е броя променливи, влизащи като входни данни в конкретния слой, а m е броя на изходните променливи от съответния слой.

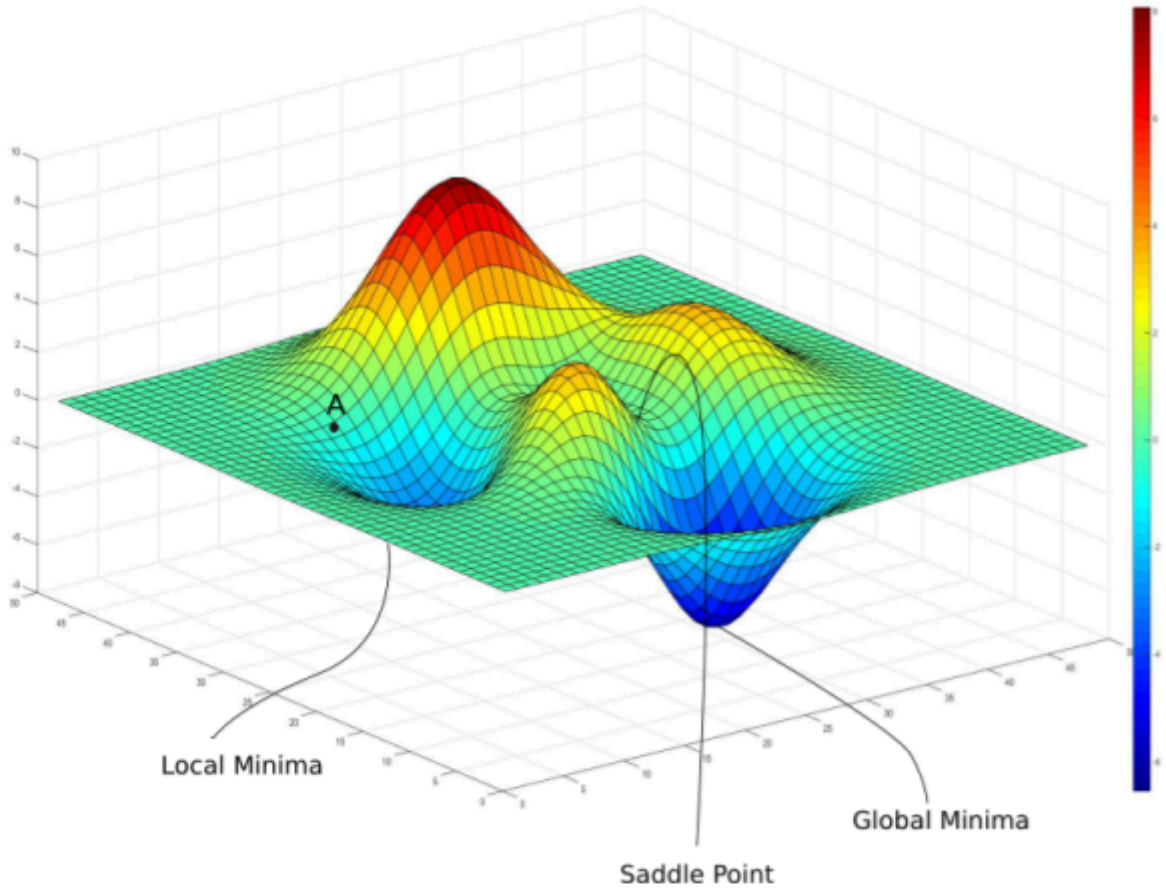
Друго решение за справяне с този проблем е прилагането на т.нар. остатъчни връзки (фиг. 1.11) между несъседни слоеве (от англ. Residual Connections[9]). Идеята е точна и ясна - добавяйки параметри от предишни слоеве дава възможност на активационната функция както да включи предишен контекст за данните, така и да включи параметри, засегнати до по-малка степен от многото умножения. Така се неутрализират до известна степен чезнещите градиенти.



Фигура 1.11. Остатъчна връзка между два несъседни слоя в невронна мрежа

Както стана ясно по-горе твърде големите стъпки при градиентното спускане пречат на модела при намирането на локален минимум, но това не е единственият случай, при който сближаването до този минимум е проблемно и отнема повече време, отколкото е необходимо. На фиг. 1.7 бе показана функция на една променлива, но пространствата, през които трябва

да итерира един модел, са много по-комплексни и са функция на множество променливи. Възможно е една функция да има множество минимуми, или пък моделът да заседне в седлови точки и т.н. (фиг. 1.12).



Фигура 1.12. Графика на функция, зависеща от 2 променливи. Функцията може да съдържа множество екстремални точки, или пък седлови точки, в околностите на които градиентите заемат и положителни, и отрицателни стойности.

Заради броя на пространствата, които моделът трябва да опише, е възможно в търсенето си на минимум в това пространство той да започне да се “лута” около минимума, да “обикаля” наляво-надясно, евентуално да го прескочи и т.н.

Този проблем не е по-малко съществен от bias-variance tradeoff или чезнещите/експлодиращи градиенти, защото тези “лутания” около минимум са много времеемки, а времето, необходимо за трениране на дадена невронна мрежа, е правопропорционално на размера ѝ. Ако по някакъв

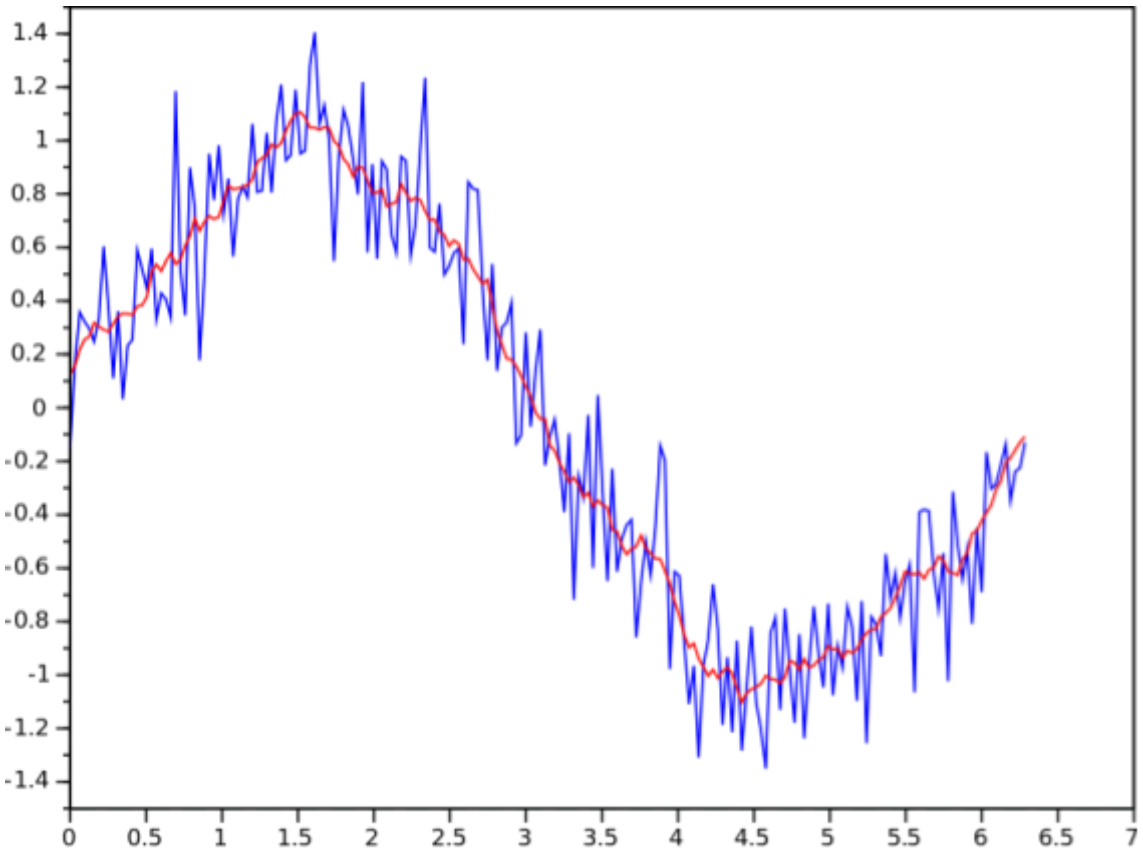
начин не се оптимизира тренирането това би довело до забавяне с часове, дни и дори седмици при тренирането на модели. А забавянията освен ресурс и време резултират в неактуален модел, защото както всичко останало, така и данните, които се генерират, се обновяват и прогресират във времето.

Универсален оптимизационен алгоритъм, до който общността е стигнала след множество проучвания и изследвания, е Adam[10] (от англ. Adaptive momentum estimation), като самият алгоритъм съчетава в себе си два други оптимизационни такива - експоненциално средно претеглени стойности (от англ. Exponentially Weighted Moving Averages, или EWMA) и RMSprop (Root Mean Square Propagation).

EWMA е статистически метод, който има за цел да моделира стойностите на поредица от времеви данни като ги осреднява спрямо n на брой стъпки назад във времето. Ако трябва да се изчисли стойността v на дадена променлива в стъпка t от времето, то изразът за v_t би имал вид на уравнението в (1.10).

$$v_t = \beta v_{t-1} + (1 - \beta) v_t \quad \textbf{(1.10)}$$

Параметърът β е стойност в интервала $[0; 1]$. За да се изчисли стойността на v_{t-1} пък ще са необходими данни за v_{t-2} и т.н. до v_1. Процесът е рекурентен, а спрямо колко стъпки назад във данните разработчика би искал да се осредняват стойностите се регулира с β. При стойност на β, близка до 1, алгоритъмът ще осреднява спрямо повече стъпки във времето, съответно графиката би изглеждала по-гладка (фиг. 1.13).



Фигура 1.13. Примерна графика на EWMA върху последователни данни. Синята графика представляват самите данни, докато червената е осредненатa им стойност спрямо n на брой предишни стъпки.

Статистическият алгоритъм EWMA, комбиниран с изчислените градиенти на невронната мрежа, дават възможност на модела да "набере скорост" докато се прилага градиентно спускане, като така моделът взима в предвид производните от предишни итерации. Затова тази част от Adam, отговаряща на EWMA, се нарича още и момент на импулса (от англ. momentum). При заместване на v_t с производните $\frac{df}{dW}$ и $\frac{df}{db}$ се получават (1.11) и (1.12).

$$\frac{df}{dW}_t = \beta\frac{df}{dW}_{t-1} + (1 - \beta)\frac{df}{dW}_t \quad \textbf{(1.11)}$$

$$\frac{df}{db}_t = \beta\frac{df}{db}_{t-1} + (1 - \beta)\frac{df}{db}_t \quad \textbf{(1.12)}$$

Вторият алгоритъм, използван заедно с EWMA в Adam, е RMSprop[11]. RMSprop може да се разглежда като допълнение към стандартното градиентно спускане, като при този алгоритъм градиентите на невронната мрежа се скалират допълнително с EWMA на квадрат, както в (1.13) и (1.14).

$$\frac{df}{dW}_t := -\alpha \frac{\frac{df}{dW}_t}{\sqrt{\beta \frac{df}{dW}_{t-1} + (1-\beta)\frac{df}{dW}_t^2} + \varepsilon} \tag{1.13}$$

$$\frac{df}{db}_t := -\alpha \frac{\frac{df}{db}_t}{\sqrt{\beta \frac{df}{db}_{t-1} + (1-\beta)\frac{df}{db}_t^2} + \varepsilon} \tag{1.14}$$

Къде стои логиката зад тези скалирания? Известно е, че свободният член b задава стойност по подразбиране на функцията при променливи равни на 0, но също така извества и нейната графика. В пространства със сравнително много измерения тези измествания стават още по-забележими. Когато моделът ни извършва градиентно спускане той би могъл да попадне в дравнително големи осцилации по измествания, без да се придвижва особено много по основните направления. Затова делейки производната на свободния член на нейният EWMA на квадрат, моделът дели производната на свободният член на някакво сравнително голямо число, което резултира в малки стъпки встрани за алгоритъма. Обратно, ако производните на параметрите са твърде малки, то при делението ще се получи голямо число и алгоритъмът ще встъпва с по-големи стъпки в основното направление. Така се регулира осцилацията встрани и допълнително се засилва моментът на импулс в основната посока на придвижване. Параметърът ε съществува в случай, че подкоренната величина стане равна на нула и ε е от порядъка на 10^{-8}.

Комбинирайки (1.11) с (1.13) и (1.12) с (1.14) се получават (1.15) и (1.16).

$$\frac{df}{dW}_t := -\alpha \frac{\beta_1 \frac{df}{dW}_{t-1} + (1-\beta_1)\frac{df}{dW}_t}{\sqrt{\beta_2 \frac{df}{dW}_{t-1} + (1-\beta_2)\frac{df}{dW}_t^2} + \varepsilon} \quad \textbf{(1.15)}$$

$$\frac{df}{db}_t := -\alpha \frac{\beta_1 \frac{df}{db}_{t-1} + (1-\beta_1)\frac{df}{db}_t}{\sqrt{\beta_2 + (1-\beta_2)\frac{df}{db}_t^2} + \varepsilon} \quad \textbf{(1.16)}$$

Всъщност (1.15) и (1.16) представляват в същността си Adam. Вземат се EWMA стойностите на градиентите и се разделят на корен от квадрата им. Така се редуцират осцилацията по вторичните направления по време на градиентното спускане и допълнително се забързва самият процес на достигане на минимум. В (1.15) и (1.16) единственият параметър, който се настройва, е α, а β_1 и β_2 са със зададени стойности по подразбиране съответно от 0,9 и 0,99. Проучвания са доказали, че тези стойности вършат работа в по-голямата част от случаите, но при нужда могат и те да бъдат допълнително настроени.

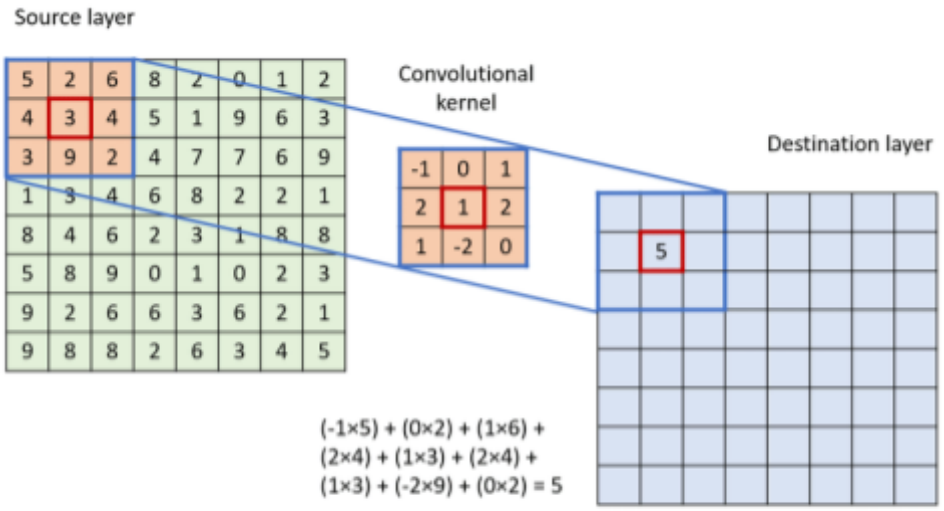
1.8. Конволюционни невронни мрежи

Конволюционните невронни мрежи са клас невронни мрежи, който най-често се използва за анализ на визуални данни - снимки, видеа, подадени кадър по кадър, и т.н. В основата им стои операцията конволюция.

Тъй като изображенията могат да бъдат представени като многоканални матрици, съдържащи стойности от 0 до 255, матричните

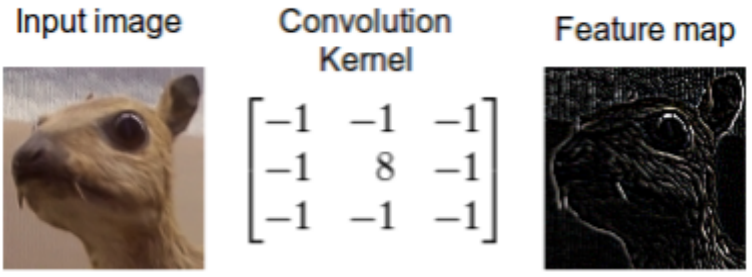
операции се оказват много сполучлив начин за трансформация на входните данни. Това представлява и операцията конволюция.

Операцията конволюция представлява следното: върху дадена матрица $X(mxn)$ с числа ще се прилага друга квадратна матрица с много по-малки размери (например с размер 3), запълнена с произволни числа. Втората матрица се нарича филтър. Филтърът започва да се прилага стъпка по стъпка за всяко парче с размери $3x3$ от изображението, като стойностите от изображенията се умножават със съответните стойности на филтъра и сборът им се присвоява на централният пиксел в това парче. Тази стъпка е итеративна и се прилага за всички $3x3$ парчета от изображението (фиг. 1.14).



Фигура 1.14. Операция по конволюция

Неслучайно втората матрица се нарича филтър. Прилагането на филтър върху входното изображение при конволюциите дава същия резултат като прилагане на ефектни филтри към обектива на фотоапарат - променя съдържанието на изображението. Как го променя зависи от числата в самите филтри, в зависимост от тях филтъра ще прилага различни трансформации, като например във фиг. 1.15.

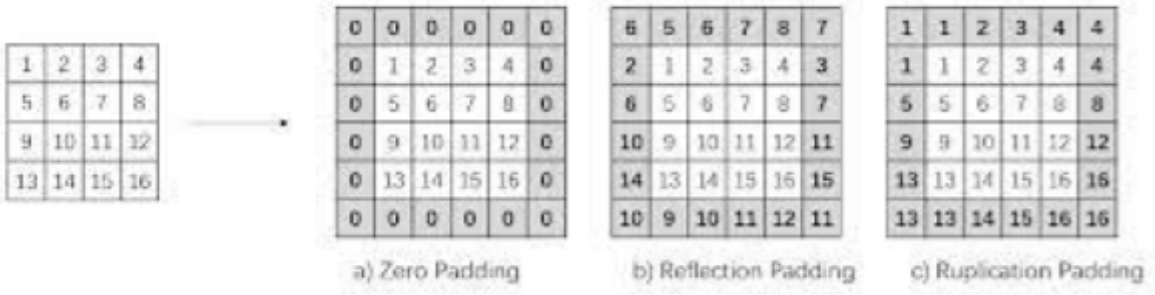


Фигура 1.15. Инициализиран по този начин, филтърът маркира наличието на хоризонтални и вертикални контури в изображението

По този начин, прилагайки различни филтри върху оригиналното изображение, конволюционните невронни мрежи са в състояние да "забелязват" повтарящи се отличителни белези и структури където и да е в изображението, давайки им предимство при анализа на изображения.

Има една особеност при операцията конволюция. Тъй като резултатът от поелементното умножение се присвоява на средния пиксел, граничните пиксели в изображението отпадат. В горния пример, филтърът е с размер $3x3$, което резултира в изображение с размер $(m-2)x(n-2)$. В зависимост от размера на входните изображения липсата на няколко реда и колони от гранични пиксели не е от особено значение, но при по-малки изображения намаляването на размера по този начин може да е нежелан ефект. Затова съществува операция по увеличаване на размера на снимката с p реда/колони от пиксели от всяка страна на изображението, където $p = \frac{n-1}{2}$. С n се означава размера на филтъра. С какви стойности се инициализират добавените пиксели не е от особено значение, но съществуват няколко варианта - може да се инициализират с огледални на граничните пиксели стойности, с нули и т.н (фиг. 1.16). Подобни конволюции се наричат "same", защото запазват размера на оригиналните

изображения. Ако не се добавят гранични пиксели, конволюцията се нарича “valid”.

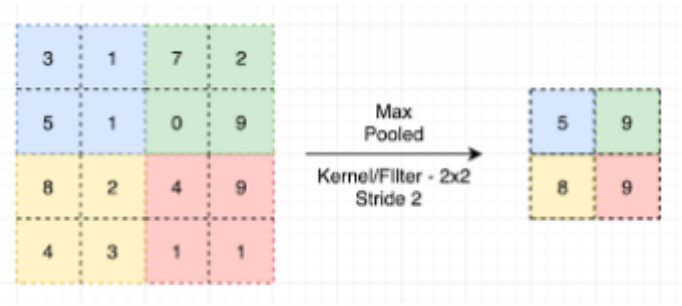


Фигура 1.16. Примери за same padding.

По-рано бе споменато, че стойностите във филтъра са произволни, но от друга страна те трябва да са специфични, за да може филтъра да засича правилно определени характеристики в изображението. Възниква въпроса как стойностите са произволно инициализирани и са същевременно и специфични. Отговорът е следният: стойностите във филтъра всъщност са параметрите, които моделът научава. Подобно на вече познатия ни напълно свързан слой, извършващ линейна регресия, параметрите във всяка единица се инициализират произволно в началото.

В конволюционните невронни мрежи обикновено операциите по конволюции са често последвани от още един вид операция върху изображения. Тя се нарича обединяване (pooling) и работи върху вече филтрираните версии на изображенията (фиг. 1.17). Подобно на конволюциите се вземат парчета от филтрираните изображенията. За разлика от конволюциите обаче, при обединяването не се прилага филтър върху тези парчета, а се взима определена стойност (най-често максималната) и се присвоява на единичен пиксел. Взетите парчета от изображението не трябва да се застъпват.

Резултата от обединението е значително намалено по размери изображение. Възниква въпросът: Намаляването на размера на изображението не води ли до значителна загуба на контекст? Отговорът е следният: обединението се прилага върху вече филтрирани изображения, които ще имат силни активации в областите, където са срещнали специфични белези/черти. При обединяване областите със силни активации се запазват, а намалявайки размера на филтрираното изображение се премахват областите без активации, които не носят никаква съществена информация.

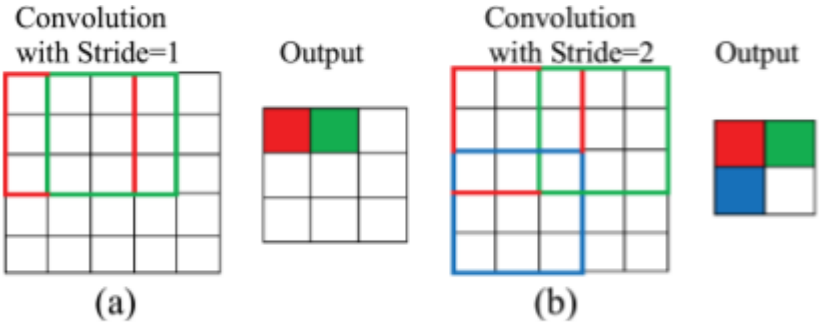


Фигура 1.17. Операция по обединяване

Обединяването е полезна операция, която ефективно намалява размерността на данните без загуба на съществена информация. Особено приятното при него е липсата на каквито и да е параметри за трениране, тъй като единствената математическа операция използвана е сравнение на стойности.

Възниква друг въпрос: С колко се намалява изображението при операцията обединение? Зависи от размера на парчето, върху което се прилага. Стандартно за размер се използват четни стойности, най често 2 или 4. При обединяване на региони с размери axa новият размер на изображението ще е $\frac{n}{a}\ x\ \frac{m}{a}$.

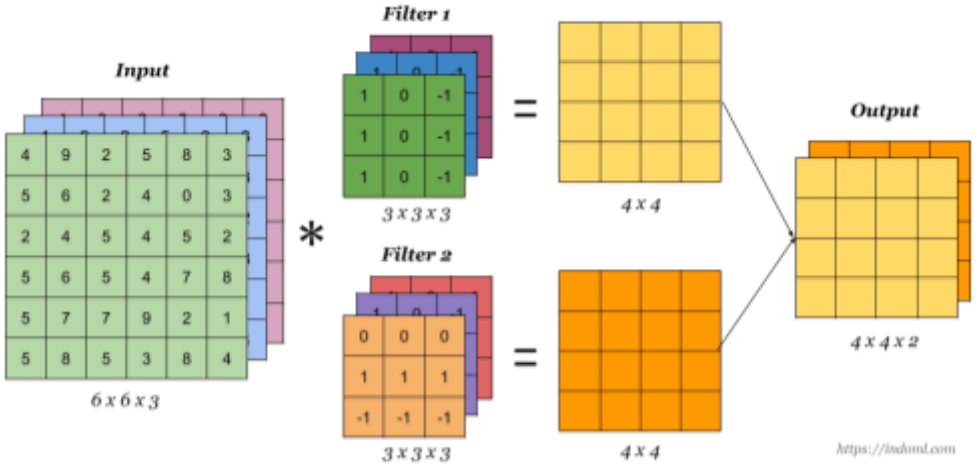
Типични хипер параметри, които е необходимо да се оптимизират както при конволюционните слоеве, така и при обединяващите слоеве, са размерът на матриците и разкрачът (от англ. stride). Разкрачът показва през колко пиксела разстояние един от друг се прилагат конволюциите и обединенията. Стандартно зададена стойност за разкрача е 1 при конволюционните слоеве, а при обединяващите такива разкрачът е с размера на самата матрица. "Виновникът" за намаляването на размера на изображенията при операцията обединение е всъщност разкрачът (фиг. 1.18).



Фигура 1.18. Различно зададени стойности на разграча при операцията конволюция последват в изходно изображение с различни размерности

В началото бе споменато, че цветовите канали на входните изображения се представят като отделни матрици. Ако се вземе предвид стандартно RGB изображение, то размерността му като входни данни ще е $(n, m, 3)$, защото RGB използва 3 канала за представяне на цветовете. По същата логика, едно черно-бяло изображение ще бъде представено с размерност $(n, m, 1)$, защото черно-белите изображения имат 1 цветови канал. Операциите по конволюция и обединение се прилагат върху всеки един слой независимо. При конволюциите броя филтри, които всеки неврон прилага върху всяко входно изображение, е равен на броя канали на входното изображение (фиг. 1.19). Ако на входа на първи слой се вкарват

стандартни RGB изображения и имаме 2 единици в слоя, то всяка от тези 2 единици ще прилага отделен филтър за всеки канал. Резултатът от конволюцията при отделните неврони си остава едноканално изображение. Изходът на първи слой ще е с размерност $(n, m, 2)$, тъй като всеки неврон прилага различни филтри върху входните изображение, създавайки по този начин нови канали.



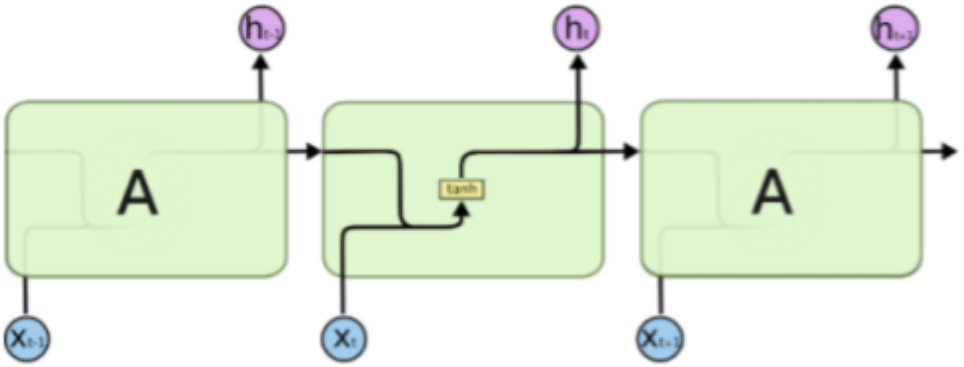
Фигура 1.19. Ако на входът на конволюционен слой с 2 неврона се подават триканални изображения, то всеки неврон ще извършва по 3 конволюции, а на изходът се получава двуканално изображение.

Следвайки логиката от напълно свързаните слоеве е редно всеки слой който прилага някакъв вид трансформация, да бъде последван от някакъв елемент на нелинейност. Конволюционните слоеве не правят изключение. Най-често използвана активационна функция е ReLU, заедно с 2 познати нейни разновидности: Leaky ReLU и ELU. Активациите се прилагат за стойностите на всеки един пиксел от резултатното изображение независимо един от друг.

1.9. Рекурентни невронни мрежи

Рекурентните невронни мрежи са втория основен клас невронни мрежи. Те имат за цел да намират зависимости в зависещи през времето последователности от данни. За такъв тип данни могат да се считат: аудио записи, последователности от символи, най-различни видове текст, времеви редове. Рекурентните невронни мрежи имат множество приложения в сфери като машинния превод, предвиждане движението на цените на стоките и акциите на фондовите борси, разпознаване на глас, анализ на настроенията и др, генерирането на текст или музика и др.

С цел придобиване на по-добра представа за смисъла на този тип невронни мрежи е добре да се започне от цялостната структура на отделен слой, както и структурата на един самостоятелен неврон. Зад структурата на рекурентните невронни мрежи стои една много ясна и логична идея: За да бъдат правилно трансформирани данни, зависещи едни от други през времето, най-вероятно ще се наложи за всяка една стъпка от обучението да се използва вече придобития досега опит от минали стъпки. За да се предвиди цената на някои стоки на фондовата борса например ще е необходим контекст за движението на цените от минали дни. Други подобни примери са машинният превод и генерирането на текст, при които генерирането на всяка следваща дума ще зависи пряко от това, което вече е казано в предишни стъпки. Затова и отделните неврони в един рекурентен слой биха изглеждали подобно на фиг. 1.20.



Фигура 1.20. Стандартна структура на единица от рекурентен слой

Вече всеки неврон има не по един вход и изход, а по два - по един допълнителен вход и изход за вече придобития контекст по време на тренирането.

Време е да бъдат въведени някои нотации, често използвани при обяснението на рекурентните невронни мрежи. Подобно на останалите невронни мрежи, с X и y се бележат входни данни и резултат. Но има и някои нововъведения. С H се бележи досега придобития по време на обучението контекст. Понеже рекурентните невронни мрежи се занимават с обработка на данни във времето, то към X, y и H се добавя долен индекс t, означаващ поредността на конкретната стъпка във времето. Може отделните единици в рекурентния слой да бъдат считани за самостоятелни стъпки във времето. Така например за втората единица е известно, че е втората стъпка във времето, следователно $t = 2$, при което означенията за входни данни и входен контекст биха били X_2 и H_1, а означенията за резултат и изходен контекст биха били y_2 и H_2.

Следват трансформациите, които всяка клетка в обикновения рекурентния слой изпълнява, както и проблемите, с което един разработчик на рекурентни невронни мрежи ще се сблъска при използването на обикновени рекурентни слоеве, защото такива има.

Уравненията, в 1.5 за намиране на резултата при напълно свързаните слоеве, са валидни и тук, но леко видоизменени. Тъй като вече се взема предвид и контекст, подаван от минали времеви стъпки, който също трябва да бъде обновен, изчисленията биха били като в (1.17) и (1.18).

$$H_t = \sigma(W_h H_{t-1} + W_x X_t + b_t) \quad \textbf{(1.17)}$$

$$y_t = tanh(W_y H_t + b_y) \quad \textbf{(1.18)}$$

Накратко - придобитият досега контекст се обогатява с казаното от поредния неврон за подадените му входни данни. На базата на новия контекст се пресмята и резултата, който невронът ще изведе, а новият контекст се предава нататък по слоя. Новите нотации са: W_h, W_y и b_y и те са параметри, които се тренират за всяка стъпка през времето и са споделени между всички единици в слоя. Те се наричат още “скрити състояния” или “скрит слой” на клетката.

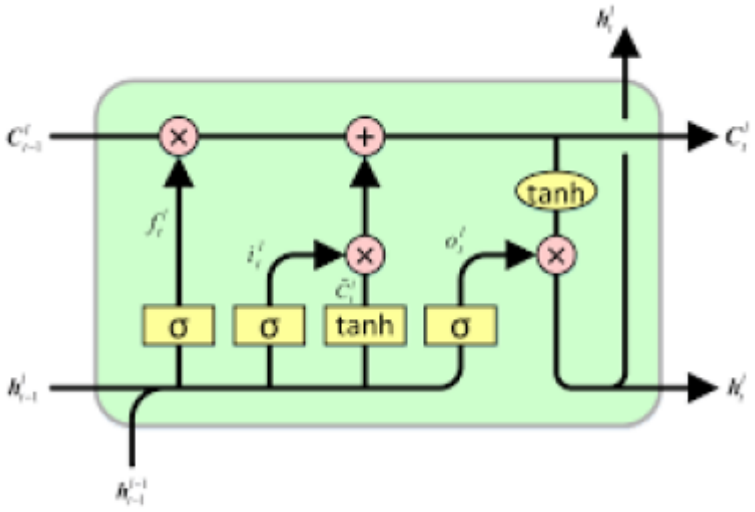
Както се вижда от формулата, използването на интуиции за данните от минали стъпки във времето изглежда компютационно по-скъпо - операциите с матрици са 3 (при типичните напълно свързани невронни мрежи съществува 1 матрична операция), което, колкото и незначително да е, се натрупва с добавянето на нови единици към слоевете. Наличието на повече матрици с параметри автоматично означава, че ще е необходимо и тренирането на повече параметри.

Рекурентните невронни мрежи наистина внасят нов поглед върху обработката на данни във времето и различният размер на входните данни не ги притеснява по никакъв начин. Но рекурентните невронни мрежи страдат от два много сериозни проблема. Първият бе вече споменат - те са компютационно по-скъпи. Но той като че ли не е толкова сериозен, в

сравнение с втория такъв - чезнещите градиенти и в частност запазването на дългосрочни зависимости.

Чезнещите градиенти са очевиден проблем - досега придобитият контекст се умножава 2 пъти и минава през 2 активационни функции за всяка клетка в конкретния рекурентен слой. И всъщност чезнещите градиенти са изключително често срещано явление при рекурентните слоеве. Което се оказва фатално за тях, защото обезсмисля основната им цел - да вземат предвид досега натрупания контекст за входните данни при трансформациите.

Подобно на всички останали разгледани досега проблеми, този също има решение, то е разработено в периода 1995-2000 г. като основната част от решението е разработена от Йозеф Хохрайтер и Юрген Шмидхюбер, като в последствие Феликс Герс и Фред Кърминс правят някои допълнения. Решението се нарича дълга/дълготрайна краткотрайна памет (фиг. 1.21, от англ. Long Short-Term Memory, или LSTM за по-кратко)[12].



Фигура 1.21. Стандартна структура на LSTM неврон

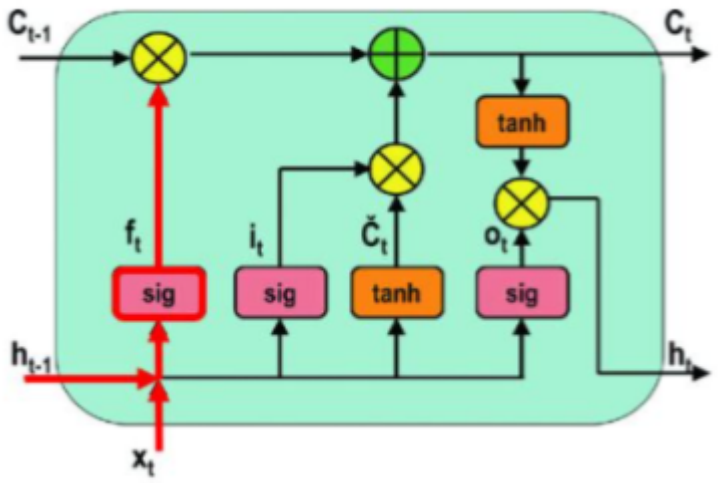
В сравнение с обикновените рекурентни клетки, LSTM имат по-усложнена структура. Те притежават няколко скрити състояния, давайки възможност на контекста и данните да преминат през различни трансформации, които целят разрешаването на различни проблеми, свързани с контекста и входните данни. Или казано по друг начин - LSTM "помнят". Тази своеобразна памет се нарича cell state и ще бъде отбелязана с C_t. Това състояние на клетката, подобно на контекста, се предава и обновява на всяка стъпка през времето. При LSTM допълнителните състояния, наречени още gates (от англ) са 4 на брой и имат вариации на (1.19).

$$\Gamma = \sigma(W_h H_{t-1} + W_x X_t + b) \quad \textbf{(1.19)}$$

С Γ се означава съответния гейт, а W_h, W_x и b са коефициенти, специфични за всеки отделен такъв. Следва обяснение на това какво прави всеки един гейт и защо го прави.

Като за начало ще бъде разгледан т.нар forget gate, чиято анотация е налична в (1.20) (фиг. 1.22, бележи се с Γ_f). Както може би подсказва името, Γ_f решава спрямо входните данни кой предишен контекст заслужава внимание и кой не. Сигмоидалната функция определя това, като дава за функционална стойност число, близко до 1 или 1, ако контекста е съществен, и число, близко до 0 или 0, ако не е.

$$\Gamma_f = \sigma(W_{fh} H_{t-1} + W_{fx} X_t + b_f) \quad \textbf{(1.20)}$$



Фигура 1.22. С червен цвят е удебелен пътят на входните данни и контекста при преминаването си през forget gate

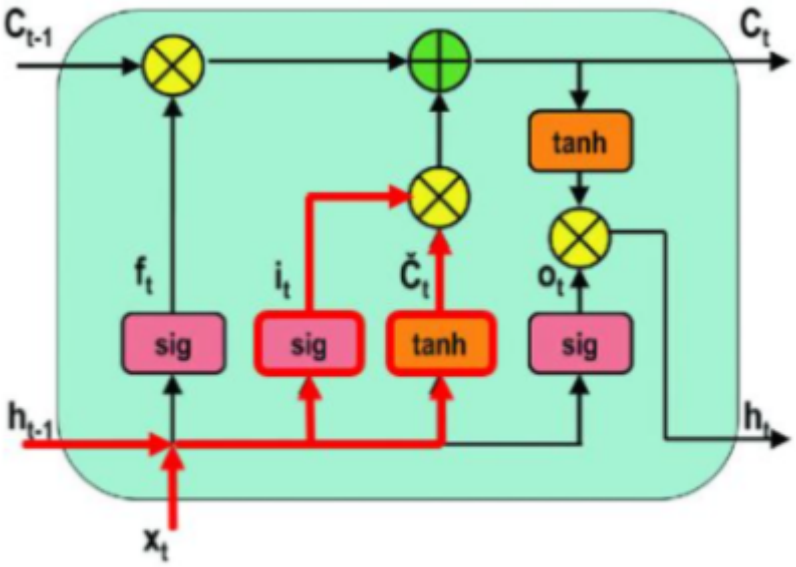
Следващ по ред е т.нар input gate (фиг. 1.23), където се извършват операциите (1.21) и (1.22). Моделът е вече решил каква част от контекста да махне, следва да реши каква част от новия контекст иска да запише и как да я асоциира. В (1.21) W_{ih} и b_i определят какво ново би записал моделът, като тази информация ще има стойност близо до 1 или 1. И обратното - ненужната информация ще получи стойности, близки до 0.

Но само тази операция не би била достатъчна за модела за да разбере изцяло новия контекст. Невронната мрежа, например, трябва да може да различи изразите “много добър” и “много лош”, защото първото словосъчетание изразява положителна асоциация, а второто - отрицателна такава. Това отношение всъщност ще представлява новото състояние на клетката и то трябва да бъде отбелязано в общия контекст. Затова съществува и операцията (1.22).

$$\Gamma_i = \sigma(W_{ih}H_{t-1} + W_{ix}X_t + b_i) \quad \textbf{(1.21)}$$

$$C_t = tanh(W_{ch}H_{t-1} + W_{cx}X_t + b_c) \quad \textbf{(1.22)}$$

При (1.22) може да се забележи разлика в активационната функция, която в случая е хиперболичен тангенс. Това е така, защото хиперболичният тангес има функционални стойности в интервала [-1; 1], които лесно биха могли да значат “много добър” или “много лош”.



Фигура 1.23. С червен цвят е удебелен пътят на входните данни и контекста в input gate

Γ_f, Γ_i и C_t са вече изчислени, всичко нужно за да обновяване на състоянието на клетката е налично. Остава самото обновление. При комбинирането на Γ_i и C_t се получава т.нар update gate (Γ_u). В част от обяснителните статии на тема LSTM Γ_u се пропуска, но неговата нотация е отбелязана в (1.23).

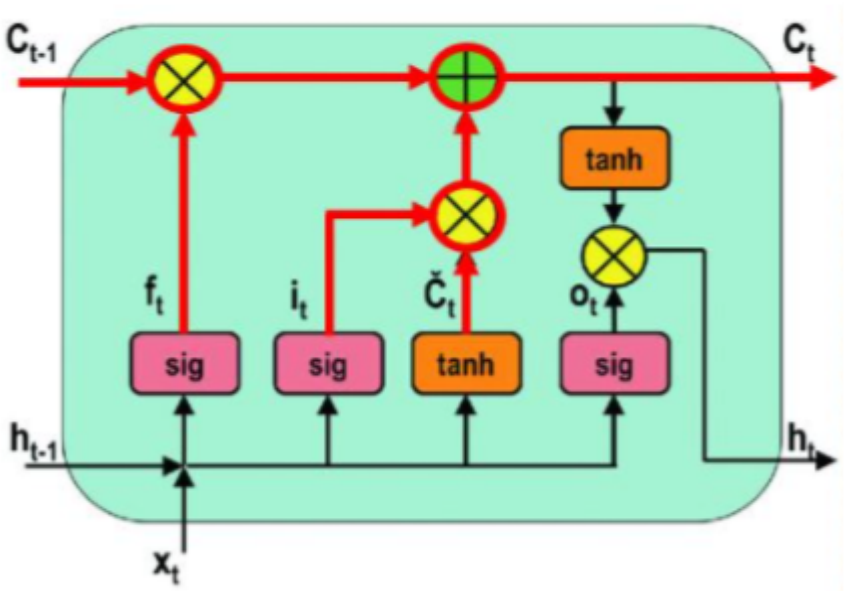
$$\Gamma_u = \Gamma_i C_t \quad \textbf{(1.23)}$$

Следващата стъпка за да обновим състоянието на клетката е да то да се комбинира с ново получената информация. Преди това обаче моделът

трябва да зачисти непотребната вече информация, което става като комбинира Γ_f с предишното състояние на клетката C_{t-1} (фиг. 1.24). В зависимост дали е използвано уравнение (1.23) биха се получили две уравнения за C_t - (1.24) и (1.25).

$$C_t = \Gamma_f C_{t-1} + \Gamma_u \quad \textbf{(1.24)}$$

$$C_t = \Gamma_f C_{t-1} + \Gamma_i C_t \quad \textbf{(1.25)}$$

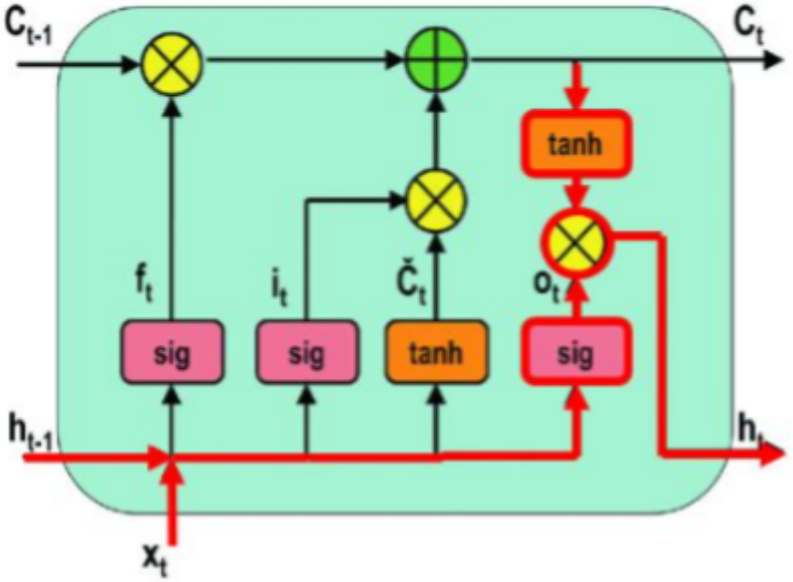


Фигура 1.24. С червен цвят е нагледно представено обновлението в състоянието на клетката.

Клетката е вече обновила състоянието си, готова е да го предаде на следващата клетка по веригата. Остава само да комбинираме състоянието на клетката. Редно е обаче клетката да добави това свое състояние към общият контекст. Това се случва в т.нар output gate (фиг. 1.25, бележи се с Γ_o). В (1.26) се изчислява Γ_o, а в (1.27) Γ_o се комбинира с вътрешното състояние на клетката.

$$\Gamma_o = \sigma(W_{oh}H_{t-1} + W_{ox}X_t + b_o) \quad \textbf{(1.26)}$$

$$H_t = \Gamma_o \tanh(C_t) \quad \textbf{(1.27)}$$



Фигура 1.25. Финалната стъпка в изчисленията н LSTM клетката - добавяне на нейното състояние към общия контекст.

Събрани на едно място всички стъпки би изглеждали по този начин:

$$\Gamma_f = \sigma(W_{fh}H_{t-1} + W_{fx}X_t + b_f)$$

$$\Gamma_i = \sigma(W_{ih}H_{t-1} + W_{ix}X_t + b_i)$$

$$C_t = tanh(W_{ch}H_{t-1} + W_{cx}X_t + b_c)$$

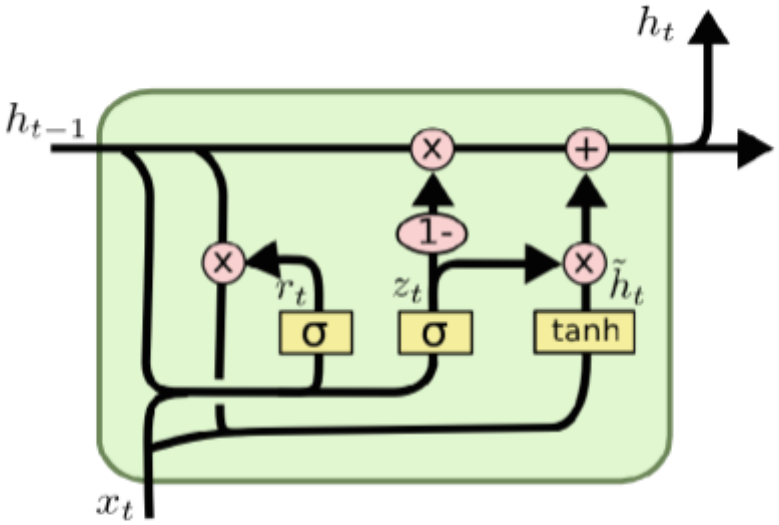
$$C_t = \Gamma_f C_{t-1} + \Gamma_i C_t \cup C_t = \Gamma_f C_{t-1} + \Gamma_u \mid \Gamma_u = \Gamma_i C_t$$

$$\Gamma_o = \sigma(W_{oh}H_{t-1} + W_{ox}X_t + b_o)$$

$$H_t = \Gamma_o \, tanh(C_t)$$

На пръв поглед става ясно, че LSTM клетките, в сравнение с обикновените рекурентни такива, заемат повече памет и изискват в пъти повече изчислителна памет - т.е модел, базиран на LSTM ще се тренира много по-бавно от обикновените такива. За сметка на това обаче LSTM решава проблема с чезнещите градиенти. И все пак има междинно решение, което решава част от проблемите с чезнещите градиенти, но заема по-малко памет и е изчислително по-евтин от LSTM - това са Gated Recurrent Units[13] (фиг. 1.26).

Разликата между GRU и LSTM се състои в липсата на output gate и състояние на клетката при GRU - операциите се извършват директно върху контекста. GRU са разработени през 2014 г. и при сравняване на резултати от проведени експерименти GRU показват подобна на LSTM успеваемост. Тъй като GRU са по-малки е по-удобно да се използват като алтернатива на LSTM, когато се налага за обработката на резултати и връщането на отговор в реално време, където бързината е от значение, или когато е налично не толкова голямо количество данни за трениране.



Фигура 1.26. Стандартна структура на GRU клетка

2. Технологии, използвани за реализация на невронни мрежи

2.1. Налични технологии и развойни среди

Създаването на софтуер за разработване и трениране на невронни мрежи е изключително трудна и времеемка задача поради ред причини - математически операции, файлови операции, оптимизиране на работата на процесора и/или графичния процесор и т.н. Затова в огромна част от случаите повечето разработчици на ИИ използват вече готов софтуер.

Най-често използваният разпространен програмен език е Python. Съществуват и специфични за статистическото програмиране езици като R, но Python се използва много, много по-често. Python е сравнително лесен език за ползване дори и от начинаещи, код на Python е сравнително лесно четим и Python позволява на разработчика да не се замисля върху изпълнението на кода на ниско ниво, каквито езици са например C/C++.

Сред библиотеките, най-често използвани при разработването на модели от дълбокото машинно обучение, са Tensorflow+Keras и Pytorch.

Pytorch е разработен от ИИ екипа на Facebook още през 2017 г., като библиотеката предлага лесни начини за отстраняване на грешки от кода, бърз е, справя се добре с голям брой данни и се интегрира добре с AWS. Библиотеката е писана на Lua. Някои от минусите на Pytorch е, че кодът може да бъде трудно четим и сложен, а програмният интерфейс е на ниско

ниво. Pytorch е подходящ за хората, които решат да разработват индивидуални проекти.

Tensorflow пък е разработен от екипът на Google още през 2015 г., като това е библиотека, даваща възможност за създаване на всякакви модели с машинно обучение. Tensorflow предлага няколко начина за създаване на невронни мрежи и дава възможност на разработчика да вгради собствен код в модела си. Интегрира се добре с Google Cloud, като библиотеката има версии за трениране както върху централен процесор, така и върху видеокартата. Самата библиотека е писана както на Python, така и на CUDA и C++ (с цел възможност за трениране върху графична карта и оптимизация на матричните операции). Предлага инструменти за следене и настройване на модела в реално време, но в сравнение с Pytorch по-трудно биха се отстранили евентуални грешки. Ако за самостоятелни проекти Pytorch се оказва по-добрият избор, то индустрията би предпочела Tensorflow пред Pytorch.

Всъщност обаче най-разпространената библиотека за разработка на невронни мрежи е Keras. Създадена е 8 месеца преди Tensorflow, като проектът е с отворен код и към него са допринесли много и различни разработчици. Keras се явява програмният интерфейс на много библиотеки за машинно обучение, като Tensorflow, Microsoft Cognitive Toolkit (преди известен още като CNTK), Theano и PlaidML. Библиотеката е изцяло писана на Python, кодът е лесно четим, предлага програмен интерфейс на високо ниво, архитектурите на невронни мрежи са често симплистични и изключително рядко ще се наложи даден разработчик да тръгне да проследява и отстранява грешки. От друга страна, понеже библиотеката е писана изцяло на Python, тренирането, forward propagation и backpropagation са много, много бавни процеси. Затова обаче пък Keras се интегрира лесно с множество други библиотеки, най-вече с Tensorflow.

Наличният софтуер е разнообразен и разполага с много функционалност. Но както стана на въпрос още в глава първа, за дълбокото машинно обучение е необходим както софтуер, така и хардуер. Тук вариантите са два - или разработчика разполага с достатъчно добър собствен хардуер, или трябва да ползва чужд ресурс. Затова съществуват и различни развойни среди, на които е възможно да се създава и тренира невронна мрежа.

Ако разработчика разполага със собствен софтуер, то развойната среда за предпочитане е Jupyter Notebook. Това е уеб апликация за създаване и споделяне на компютационни документи и разполага с лесен и интуитивен програмен интерфейс. Може да бъде инсталиран отделно, или като част от Anaconda Navigator. Anaconda Navigator е графичен потребителски интерфейс за компютър, създаден от Anaconda, който позволява лесното менажиране на софтуерни пакети и зависимости. Anaconda Navigator идва и с куп други приложения, полезни за хора, занимаващи се с анализ на данни. Jupyter Notebook позволява инстанцирането на работни среди както върху централният процесор, така и върху графичната карта.

Ако разработчика не разполага със собствен софтуер има няколко опции за развойни среди. Kaggle, Paperspace, AWS Sagemaker и Google Colab предоставят облачни услуги за създаване и трениране на невронни мрежи, където обаче ресурсите са лимитирани. Използването на графични ускорители безплатно е лимитирано, след което ресурсите се освобождават и при невнимание разработчика може да загуби досега разработвания модел, а AWS автоматично ще започне да таксува използването на ресурсите им след превишаване на лимита от часове.

За целите на тези експерименти ще бъде използвана библиотеката Tensorflow, а развойната среда - Google Colab. Частично ще бъдат използвани и други библиотеки, като Matplotlib и Numpy.

Numpy е библиотека към Python, чиято цел е улесняване работата с вектори и матрици от данни, както и прилагането на най-разнообразни операции върху тези данни. Самата библиотека има дълга история, като Numpy излиза с името Numeric още през 1995 г., а 10 години по-късно името е сменено на Numpy. Библиотеката е толкова разпространена и доказала се във времето, че други библиотеки за обработка на данни като Pandas, както и Tensorflow, Pytorch и Keras основават работата си с масиви и матрици на Numpy. Библиотеката е подробно документирана и сравнително лесна за използване дори и от начинаещи. Numpy е писана основно на Python и C, а операциите с масиви и матрици в Numpy са оптимизирани и се изпълняват с стотици до хиляди пъти по-бързо в сравнение с операции, написани на обикновен Python.

Matplotlib е библиотека, обслужваща Python и Numpy, имаща за цел изобразяване на всякакъв вид графики. Библиотеката е написана през 2003 г., като нейният програмен интерфейс е обектно ориентиран. Самата библиотека се базира на инструменти за графичен потребителски интерфейс като Tkinter. Често се срещано е други видове инструменти за визуализация да се използват като потребителски интерфейс с повече възможности, който под повърхността си да работи с Matplotlib, каквито например са Seaborn, GTK и Basemap. От 2020 г. Matplotlib е спрял да поддържа Python 2, като в момента библиотеката работи само на версии на Python 3.

2.2. Постановка на проблема и вече налични решения

Белтъците с цинкови пръсти са сред най-разпространените групи белтъци, които изпълняват множество функции. Поради голямото им

структурно разнообразие, те имат способността да взаимодействат с множество други белтъци, както и с ДНК, РНК и ПАРП(поли (АДФ-рибоза) полимераза, ензим, регулиращ множество жизнени процеси, като цялостността на ДНК, изразяване на гените и клетъчното делене). В следствие на това свое свойство, белтъците с цинкови пръсти участват активно в регулацията на множество клетъчни процеси, транскрипцията, разлагането на други видове белтъци, съкращаването на актиновия белтък в мускулите, движението на клетки в следствие на химични или механични процеси и много други. Известни факти за белтъците с цинкови пръсти са, че участват в обособяването на четирите вида тъкани (съединителна, епителна, мускулна и нервна), като заедно с това изявяват роля при образуването на тумори, развитието на ракови заболявания и формирането на метастази. Те стоят в основата на част от неврогенетичните заболявания.

Форматът на данните, с които ще се борави, е следният: всеки запис представлява нуклеотидна последователност от 1000 нуклеотида. Както е възможно да намери в повечето учебници по биология, ДНК е съставена от 4 вида нуклеотиди: аденин (А), гуанин(Г), цитозин(Ц) и тимин(Т), като навързани един за друг, тези нуклеотиди съставят една последователност/верига. За разлика от РНК, ДНК е съставена от 2 такива вериги, които обаче са комплементарни една на друга - А от първата верига се свързва само с Т от втората, а Г - само с Ц. Затова е възможно да се разбере съдържанието на една ДНК разполагайки дори само с 1 нейна верига.

Нуклеотидите са биохимични съединения. Те са краен брой изброими стойности и означават определени свойства. Затова да превърнем нуклеотидите в числови стойности не би имало смисъл: не възможно да съберат А и Ц, да се умножат А по Т или пък да сравняват по стойност. Те са дискретни/категорийни променливи. Затова данните са представени чрез т.нар one-hot кодиране. Всеки нуклеотид представлява масив от четири

елемента със стойности 0 или 1 за всеки един от тях. Така кодираме A като `[1, 0, 0, 0]`, Г като `[0, 1, 0, 0]`, Ц като `[0, 0, 1, 0]` и Т като `[0, 0, 0, 1]`.

В света на дълбокото машинното обучение е всеизвестно, че нерядко дадена задача може да бъде разрешена чрез използването на различни типове подходи. В контекста на конкретния проблем срещнах няколко различни подхода при решаването на подобни задачи. Единия от тях е проектът DeepSea[14], който залага изцяло на използването на конволюционни невронни мрежи и постига висока точност върху нови данни.

Преди всичко за обучението на конкретните невронни мрежи се използват нуклеотидни последователности, където редът в данните има значение. Затова е логично да е възможно прилагането на рекурентни невронни мрежи върху данните. За разлика от текстовите данни обаче, при нуклеотидите последователностите не са умопомрачително много - ДНК (и РНК) имат за цел да кодират белтъчни последователности, изградени от общо 20 известни аминокиселини. При налични 4 нуклеотида необходимият брой нуклеотиди, необходим за кодирането на 1 аминокиселина, е $4^x > 20; x \in N \Rightarrow x = 3$, което означава, че са необходими 64 нуклеотидни комбинации от по 3 нуклеотида (кодони) за кодирането на аминокиселините, при това някои аминокиселини притежават по повече от 1 начин за кодиране. Или по друг начин казано - "генетичният" език е много по-опростен и подреден в сравнение с естествените езици. Представена чрез one-hot кодиране, в образуваната 4x1000 матрица ще има региони, при които позиционирането на стойностите 0 и 1 ще се повтаря. А идеята на конволюционните невронни мрежи е точно такава - да забелязват често срещани особености у данни, представени в матричен вид.

Възможно е обаче да се погледне на нуклеотидните последователности и по друг начин. Нека за пример имаме нуклеотидната последователност ГГЦ-ГГА-ГЦТ-ГГЦ-ГАЦ-ТЦЦ. В естествения език можем думите да се считат за основни смислови единици, които като се комбинират, съставят изречения, а всяко изречение може да се раздели на думи, които носят някакъв контекст. Ако целият полипептид представлява нашето “изречение”, то отделните кодони в него ще представляват думите, носещи някакъв контекст (ГГЦ и ГГА - глицин, ГЦТ - аргинин, ТЦЦ - серин и т.н).

3. Изследване на архитектури на невронни мрежи за разрешаването на поставените проблеми

В следващата глава ще бъде по-подробно разгледано разработването и тестването на различни видове архитектури на невронни мрежи за решаването на дефинираните в самото начало задачи.

3.1. Използване на конволюционни невронни мрежи за класификация върху два класа

Преди да съществува невронна мрежа, или каквото и да е машинно обучение изобщо, трябва да са налични данни, и то немалко. Линк към

данните ще има в референциите[16]. Самите данни са във .h5 файлов формат, като това представлява разширение на HDF формата (от англ. Hierarchical Data Format). Това представлява файлов формат за съхранение и подредба на големи поредици от данни. В Python библиотеката, използвана за файлови операции е h5py. Отварянето и четенето на файловете се повтаря за всички данни, върху които ще бъдат проведени експерименти.

За да може обаче данните да бъдат консумируеми по време на трениране те трябва да се прочетат във удобен за Tensorflow начин. Затова съдържанието на .h5 файловете се записва в Numpy масив.

```
train_data = h5py.File(r'/content/drive/MyDrive/Machine
Learning/Thesis/MA0035_4_m3_train.h5', 'r+')
test_data = h5py.File(r'/content/drive/MyDrive/Machine
Learning/Thesis/MA0035_4_m3_test.h5', 'r+')

train_data_binlabels, train_data_features = np.array(train_data['binlabels']),
np.array(train_data['data'])
test_data_binlabels, test_data_features = np.array(test_data['binlabels']),
np.array(test_data['data'])
```

Данните обаче се подават в неподходящи измерение. За всеки запис има матрица с размер $1000x4$, а за да бъдат едномерните конволюции валидни и смислени, те трябва да бъдат приложени върху масив $4x1000$. Затова всеки един запис се транспонира.

```
train_data_features = np.array([i.T for i in train_data_features])
test_data_features = np.array([i.T for i in test_data_features])
```

Понеже всеки един незатворен файл е неосвободен ресурс, след като вече четенето на данни е приключило се затварят файловите дескриптори.

```
train_data.close()
```

```
test_data.close()
```

Както стана ясно във втора глава, размерът на данните е матрица с 4 реда и 1000 колони. Също така бе изяснено, че генетичните последователности е могат да бъдат разглеждани по два начина - от гледна точка с помощта на конволюционни невронни мрежи, но и от гледна точка на последователности. При конволюционните мрежи е изключително важно да се настрои размерността на входните данни предварително, затова входът на невронната мрежа ще бъде:

```
      input_layer = Input(shape = (1000, 4))
```

Тъй като данните за трениране, с които моделът от първото изискване разполага, не са голямо количество, първото решение, което бе взето относно бъдещият модел, е че той не бива да бъде твърде голям, защото това ще доведе единствено до overfitting. Затова първоначално невронната мрежа е съставена от три слоя - първите два слоя конволюционни, а третият - напълно свързан такъв.

```
hidden_layer = Conv1D(filters = 20, kernel_size = 4, padding = "same",
activation = "relu")(input_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Conv1D(filters = 30, kernel_size = 4, padding = "same",
activation = "relu")(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Flatten()(hidden_layer)
hidden_layer = Dense(units = 10, activation = "relu")(hidden_layer)
```

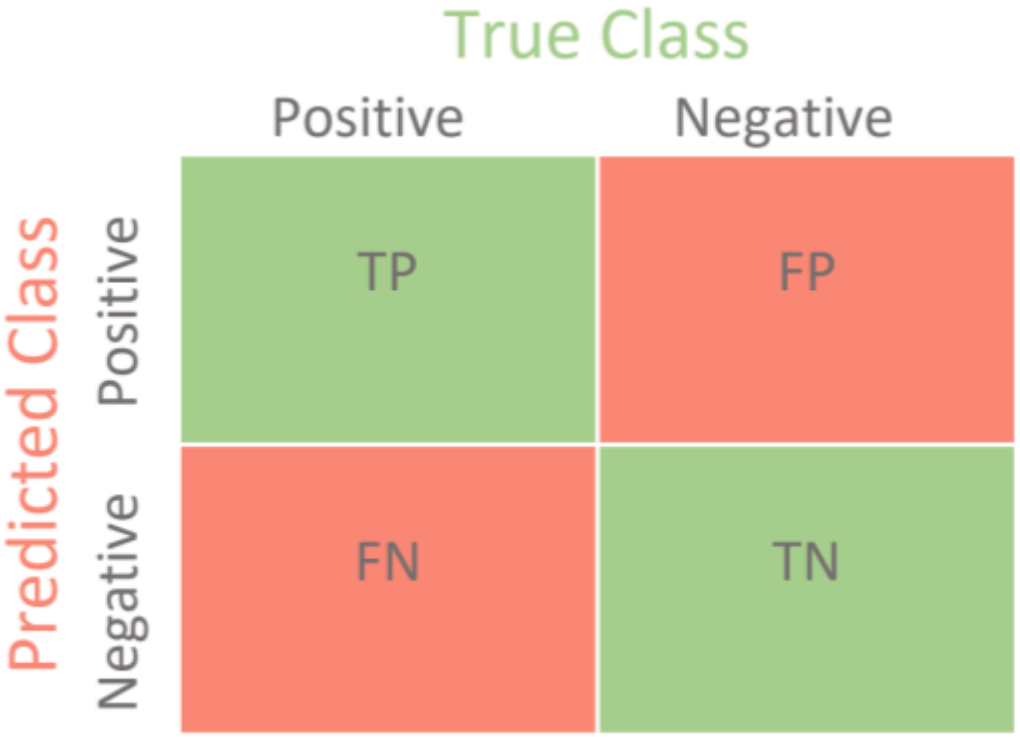
Понеже задачата е класификация по един транскрипционен фактор, модела трябва да бъде настроен да връща на изхода си число, което има две

стойности - 0 (транскрипционен фактор липсва) или 1 (транскрипционен фактор е наличен). Напълно свързания последнен скрит слой от невронната мрежа се връзва към изход с един неврон със сигмоидална активационна функция:

```
output = Dense(1, activation = "sigmoid")(hidden_layer)
```

След това се създава обект от тип модел, на когото като параметри се подават входния и изходния слой. Моделът бива компилиран със стандартни за бинарната класификация настройки - функция на загубата е двоична крос ентропия, функцията на загубата се оптимизира с Adam, а за метрика се използва AUC.

Метриката AUC (Area Under Curve, накратко от Receiver Operating Characteristics Area Under Curve, или ROC AUC) е изключително често използвана метрика при модели за машинно обучение, целящи бинарна класификация. Това всъщност представлява вероятностна крива, която е разграфена на прагове, като на базата на тези прагове се измеря ефективността на модела в разпознаването на двата класа. Произходът на метриката произлиза от т.нар матрица на объркванията (фиг. 3.1), използвана за изчисляването на множество от метрики при бинарните класификации.



Фигура 3.1. Стандартно означение на матрица на объркването

При матрицата на объркванията се съотнасят предсказаната от алгоритъма категория и реалната такава. Ако алгоритъмът е предсказал клас 1 и в действителност записът съответства на клас 1, то тази съпоставка се нарича True Positive. При съответствие и на двете с клас 0 се нарича True Negative. Ако алгоритъмът е предсказал клас 1, но в реалност записът съответства на категория 0, то съпоставката се нарича False Positive или грешка от тип 1. Обратното - ако моделът е предсказал 0 там, където записът действително е 1, съпоставката се нарича False Negative или грешка от тип 2.

Тези съпоставки и числените им съотношения се оказват важни в задачи, където целта е разпознаване на аномалии, където случаите на аномалии ще са много много малка част от всички данни. За ROC-AUC метриката се визуализира отношението между True Positive Rate (3.1) по ординатната ос и False Positive Rate (3.2) по абсцисната.

$$True\ Positive\ Rate\ (TPR) = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Negative\ (FN)} \quad \textbf{(3.1)}$$

$$False\ Positive\ Rate\ (FPR) = \frac{False\ Positive\ (FP)}{True\ Negative\ (TN) + False\ Positive\ (FP)} \quad \textbf{(3.2)}$$

В Tensorflow ROC-AUC метриката съществува наготово, затова безпроблемно може да бъде използвана при компилирането на модела. Метриката поддържа и някои други аргументи, които ще бъдат от полза в по-нататъшните експерименти.

```
single_class_model = Model(inputs = input_layer, outputs=output)

single_class_model.compile(
    loss = BinaryCrossentropy(),
    optimizer = Adam(beta_1 = 0.95),
    metrics = [AUC(name = "AUC")]
)

multilabel_model.summary()
```

При извикване на метода summary() ще се изведе информация относно всеки слой.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 20)          340

 max_pooling1d (MaxPooling1D  (None, 250, 20)          0
 )

 conv1d_1 (Conv1D)           (None, 250, 30)           2430
```

```
 max_pooling1d_1 (MaxPooling  (None, 62, 30)           0
 1D)

 flatten (Flatten)           (None, 1860)              0

 dense (Dense)               (None, 10)                18610

 dense_1 (Dense)             (None, 1)                 11

=================================================================
Total params: 21,391
Trainable params: 21,391
Non-trainable params: 0
_________________________________________________________________
```

Моделът има около 21 хил. параметъра и повечето от тях идват от последният слой. Невронна мрежа с тези размери може да се счита за миниатюрна, но дали е достатъчна ще се разбере само след трениране. Моделът ще се тренира за 50 стъпки максимум, като ако AUC върху валидационните ни данни не се повишава повече от 3 итерации, обучението се спира. Валидационните данни представляват 5% от трениращите. При всяка итерация разбъркваме трениращите данни.

```
callback = [EarlyStopping(patience = 3)]
single_class_model_history = multilabel_model.fit(train_data_features,
train_data_binlabels, epochs = 50, batch_size = 1000, validation_split = 5e-2,
shuffle = True, callbacks = callback)
```

Стъпките по процесът на трениране, както и стойността на функцията на грешката и стойността на метриката за всяка стъпка се записват в история на тренирането, откъдето в последствие може да бъде взета и използвана според желанията на разработчика. За целта е съставена кратка функция, която създава графика-полигон, отразяваща промяната на стойността на

функцията на загубата и стойността на AUC през епохите. При извикването си тази функция изобразява фиг. 3.2.

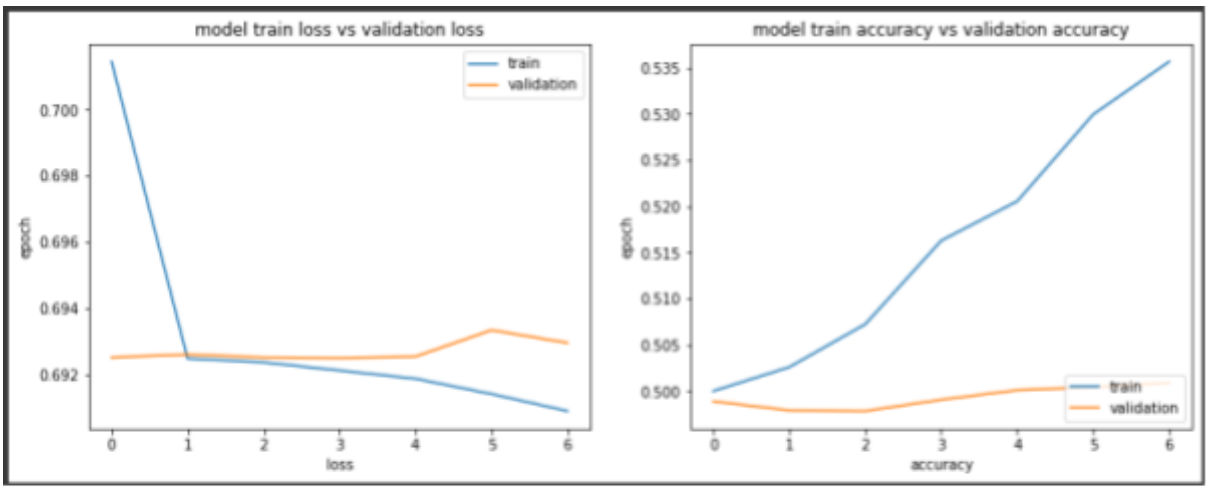
Фигура 3.2. Представени са две графики. На графиката са изобразени движенията на стойностите на функцията на грешката през епохите за съответно трениращите и валидационните данни, а отдясно - AUC-ROC върху трениращите и валидационните данни.

Моделът не се представя никак добре. Трениран е за 7 епохи и в рамките на това трениране той не научава нищо за данните. Той сериозно проявява белезите на underfitting.

Ако човек се замисли върху размерността на данните и параметрите на първият конволюционен слой, може да заключи, че най-вероятно филтрите не вземат контекст от достатъчно много на брой нуклеотиди. Като първа мярка може да се увеличи големината на филтъра при първия конволюционен слой на 10 и допълнително да се увеличи размера на обединяващият слой след това.

```
hidden_layer = Conv1D(filters = 20, kernel_size = 10, padding = "same",
activation = "relu")(input_layer)
hidden_layer = MaxPooling1D(pool_size = 10, strides = 10)(hidden_layer)
```

След трениране на модела отново се получават резултатите от фиг. 3.3.



Фигура 3.3. Моделът още от началото започва да научава част от данните наизуст, но поради малкия си капацитет не може да научи никакви съществени зависимости.

Изглежда като че ли при модела има някакво раздвижване, но все още има затруднения да стартира обучение. Забелязва се, че съществува наклонност освен към underfitting и към overfitting. Като че ли моделът не може да изгради ясна представа за данните от тези два слоя. Затова ще бъде включен още един конволюционен слой, като той ще има същата размерност на филтъра като първия, а при третия размерността на филтъра ще бъде зададена на 8 пиксела. Допълнително ще се намали броя неврони при всеки слой и се добавя Dropout регуляризация от 10% след всеки слой.

```
hidden_layer = Conv1D(filters = 16, kernel_size = 10, padding = "same",
activation = "relu")(input_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
hidden_layer = Conv1D(filters = 24, kernel_size = 10, padding = "same",
activation = "relu")(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
hidden_layer = Conv1D(filters = 32, kernel_size = 8, padding = "same",
activation = "relu")(hidden_layer)
```

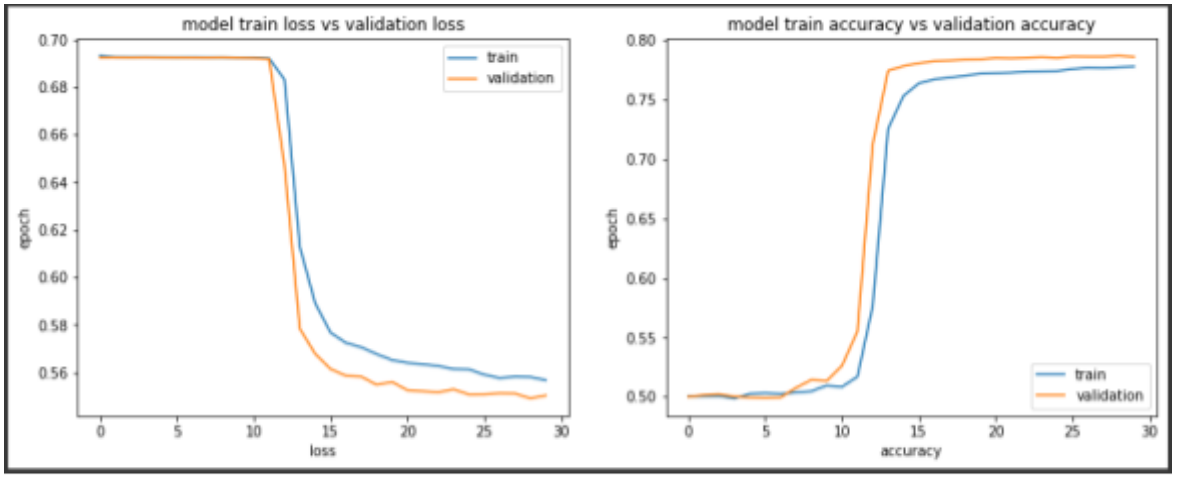
```
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
hidden_layer = Flatten()(hidden_layer)
hidden_layer = Dense(units = 32, activation = "relu")(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 16)          656

 max_pooling1d (MaxPooling1D  (None, 250, 16)          0
 )

 dropout (Dropout)           (None, 250, 16)           0

 conv1d_1 (Conv1D)           (None, 250, 24)           3864

 max_pooling1d_1 (MaxPooling  (None, 62, 24)           0
 1D)

 dropout_1 (Dropout)         (None, 62, 24)            0

 conv1d_2 (Conv1D)           (None, 62, 32)            6176

 max_pooling1d_2 (MaxPooling  (None, 15, 32)           0
 1D)

 dropout_2 (Dropout)         (None, 15, 32)            0

 flatten (Flatten)           (None, 480)               0

 dense (Dense)               (None, 32)                15392

 dropout_3 (Dropout)         (None, 32)                0

 dense_1 (Dense)             (None, 1)                 33
```

```
=================================================================
Total params: 26,121
Trainable params: 26,121
Non-trainable params: 0
_________________________________________________________________
```

Понеже в модела има регуляризация, при конкретното трениране ранното спиране при подобряваща се точност ще бъде деактивирано. За сметка на това епохите за трениране ще са по-малко - ще са 30.

След последващо трениране се получават резултатите от фиг. 3.4.



Фигура 3.4. След добавяне на допълнителен конволюционен слой с разширен филтър моделът е в състояние да генерализира 78% от характеристиките на трениращите данни и 80% от характеристиките на тестовите такива.

Резултатите са в пъти по-добри спрямо предишните опити. На моделът отново му е необходимо известно време за да "загрее", но нека не забравяме, че това все пак е конволюционна невронна мрежа. Нужни са първоначални корекции по тежестите на филтрите. Моделът достига 77,8% точност при трениращи данни и 78,61% при валидационни такива. При последваща валидация върху тестови данни се получават следните резултати:

```
313/313 [==============================] - 2s 5ms/step - loss: 0.5428 - AUC: 0.8037
[0.5427682399749756, 0.8036550879478455]
```

Резултатите върху данни, които алгоритъмът не е виждал, е дори по-високо - 80,5%. Нека да се отбележи, че това все пак са генерирани данни, което означава, че разпределенията и на трениращите, и на тестовите данни са почти еднакви. Върху реални данни се очаква моделът да се справя толкова добре, колкото се справя при валидация. При сравнително малкото количество данни, с което моделът разполага, това е много добър резултат, като си заслужава параметрите на този модел да бъдат запазени, като могат да послужат за основа за по-нататъшни архитектури, които биха имали за цел да решават подобни проблеми.

3.2. Използване на невронни мрежи от смесен тип за класификация на два класа

Подобно на конволюционният подход, при смесеният такъв съществуват вече готови разработки върху подобни проблеми за решаване, като една такава разработка е проектът DanQ[2], където разработчикът е използвал 3 огромни по капацитет слоя и е изградил конволюционно-рекурентна невронна мрежа. Входните данни първо минават през конволюционна невронна мрежа, а след това трансформираните с филтър репрезентации се подават на рекурентна такава.

В контекста на точка 3.1 това означава, че при смяна на последния конволюционен слой с рекурентен би трябвало да изведат също добри резултати. Архитектурата, която ще бъде използвана за следващият

експеримент, се състои от 2 конволюционни слоя, последвани от двупосочен рекурентен такъв, който не връща скритото си състояние, а само краен резултат.

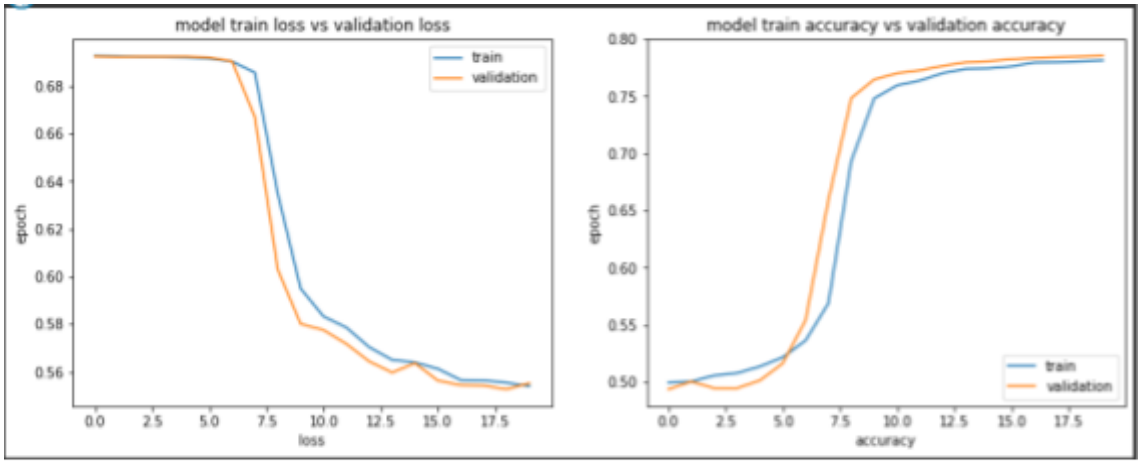
```
hidden_layer = Conv1D(filters = 16, kernel_size = 10, padding = "same",
activation = "relu")(input_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Conv1D(filters = 32, kernel_size = 4, padding = "same",
activation = "relu")(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Bidirectional(LSTM(units = 32))(hidden_layer)
hidden_layer = Dense(units = 32, activation = "relu")(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
```

За компилация се използват същите настройки от моделът в 3.1. При трениране максималният брой епохи за трениране се настройват на 20. При извикване на `summary()` се извежда следната информация:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 16)          656

 max_pooling1d (MaxPooling1D  (None, 250, 16)          0
 )

 conv1d_1 (Conv1D)           (None, 250, 32)           2080

 max_pooling1d_1 (MaxPooling  (None, 62, 32)           0
 1D)

 bidirectional (Bidirectiona  (None, 64)               16640
 l)

 dense (Dense)               (None, 32)                2080
```

```
 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 21,489
Trainable params: 21,489
Non-trainable params: 0
_________________________________________________________________
```

Моделите са сравними по големина, въпреки че този е по-малък. Както се вижда близо 80% от параметрите на невронната мрежа се намират в нашият двупосочен LSTM слой. LSTM са наистина скъпички, но вършат страхотна работа. Резултатите от фиг. 3.5 го потвърждават.



Фигура 3.5. Смяната на конволюционен слой с рекурентен не намалява способностите на модела в генерализирането на основните характеристики у нуклеотидните последователности.

Смесената архитектура постига 78,13% точност върху трениращи данни и 78,54% върху валидационни. Времето за трениране е обаче сравнително повече: 30 стъпки * 6 секунди на стъпка, или около 3 минути, за конволюционната мрежа, и 20 стъпки по 80 секунди, или почти 27 минути,

за смесената мрежа, която е дори и по-малка по брой параметри. Резултатът при оценка върху тестови данни е 80,52% точност:

```
313/313 [==============================] - 5s 16ms/step - loss: 0.5322 - AUC: 0.8052
[0.532249927520752, 0.805203914642334]
```

Изводът от направеното сравнение е, че двата подхода - чисто конволюционният и смесеният - са съпоставими що се отнася до точност на предсказанията. Първата архитектура обаче достига до тази точност за деветократно по-малко време. Има и друга разлика - включването на LSTM слой в модела не му позволи да научи малкото на брой данни, с които разполагаме. Както се убедихме по-рано, случай на underfitting, примесен с overfitting, е възможен, но при вторият модел самата структура на LSTM, както и малкият размер на цялостния модел, не го позволиха, затова Dropout регуляризация бе сложена само преди изходния слой. Въпреки разликата във времената за трениране смесената невронна мрежа има предимство - LSTM слоя не е филтър, неговите контекст и състояния се променят в зависимост от входните данни и не са статични както филтрите.

3.3. Използване на невронни мрежи от смесен тип за класификация на повече от два класа

Вследствие на проведените експерименти бе установено, че четирислойна невронна мрежа от смесен тип се справя сравнително добре върху данни с един транскрипционен фактор. Семейството транскрипционни фактори, както бе споменато във втора глава, са най-често срещаните транскрипционни фактори.

Следващата логична стъпка към решаването на задачата е да се разработи модел, който дава достатъчно добри резултати върху множество от ZNF транскрипционни фактори, не само върху един. За тази задача са подбрани един милион генерирани нуклеотидни последователности за трениране и 100 хил. такива за тестване. Нуклеотидните последователности са от същия размер като в 3.1 - матрица $4x1000$. Броят на транскрипционните фактори е 29.

Преди обаче да се пристъпи към експериментите е редно да се уточни как точно се тренира модел, който да може да разпознава повече от 2 класа, в конкретния случай - 29.

Когато дадени записи трябва да бъдат разпределяни в повече от 2 категории/класа трябва разработчикът да си зададе следният въпрос: “Възможно ли е даден запис от данните върху конкретната задача да бъде причислен към повече от 1 категория?”. Ако задачата е да се разпознаят колкото се може повече обекти на изображение, а изображението например представлява оживено кръстовище рано сутрин, от моделът се очаква да може да различи наличието на хора, автомобили, светофари, пътни знаци и т.н. В такъв случай за съответните категории моделът трябва да се произнесе с резултат 1, а за останалите, които не присъстват в изображението - 0. Такъв вид класификация се нарича класификация по множество признаци (от англ. multilabel classification). Ако пък задачата на разработчика е да определи какво животно е заснето и на изображението има котка, то моделът трябва да се произнесе с резултат 1 само за невронът, който отговаря за изход “котка”, а при всички останали изходи резултатът трябва да е 0. Такъв вид класификация върху повече от една категория, където записът може да представлява точно 1 категория, се нарича многокласова класификация (от англ. multiclass classification).

Следващата стъпка за решаването на задачата е да се прецени какъв вид класификация върху множество категории е необходима за конкретният

проблем. Химически някои от транскрипционните фактори се различават само по една аминокиселина, като например цистеин и хистидин, като така е възможно участието на няколко транскрипционни фактора в образуването на един протеин. Т.е. възможно е в една нуклеотидна последователност да се срещнат по няколко транскрипционни фактора, в някои случаи може такива дори и да не присъстват. Затова типът класификация върху много категории, който ще бъде използван, е multilabel класификация.

Остава да бъде изяснено как се оценява точността на multilabel модел. Тъй като транскрипционните фактори съществуват независимо един от друг, то оценката може да бъде сведена до осредняване на 29 бинарни класификации. В глава 3.1 бе споменато, че класът AUC в Tensorflow разполага с допълнителни параметри. Един от този параметри е булев флаг с името `multi_label`, чиято стойност по подразбиране е зададена на `False`. Задавайки на този флаг стойност `True`, AUC ще оценява индивидуално точността на модела за всеки един от тези 29 транскрипционни фактора и ще върне осреднена стойност.

```
multilabel_model.compile(
    loss = BinaryCrossentropy(),
    optimizer = Adam(),
    metrics = [AUC(name = "AUC", multi_label = True)]
)
```

Останалите настройки за компилиране на модела си остават същите както преди.

Относно модела - при наличието на 10 пъти повече данни както за трениране, така и за тестване, е редно неговият капацитет да бъде увеличен. Затова броят единици за всеки слой ще бъдат увеличени.

```
hidden_layer = Conv1D(filters = 35, kernel_size = 10, padding = "same",
```

```
activation = “relu”)(input_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Conv1D(filters = 35, kernel_size = 4, padding = "same",
activation = “relu”)(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Bidirectional(LSTM(units = 70))(hidden_layer)
hidden_layer = Dense(units = 35, activation = “relu”)(hidden_layer)
hidden_layer = Dropout(0.1)(hidden_layer)
```

Както бе споменато по-рано, след като наличните транскрипционни фактори са общо 29, то е редно в изходния слой броя неврони да е 29, а не 1. Но освен броя неврони нищо друго не се променя. Както вече бе установено, всеки един транскрипционен фактор съществува независимо от останалите, затова активационната функция си остава сигмоидална.

```
output = Dense(29, activation = "sigmoid")(hidden_layer)
```

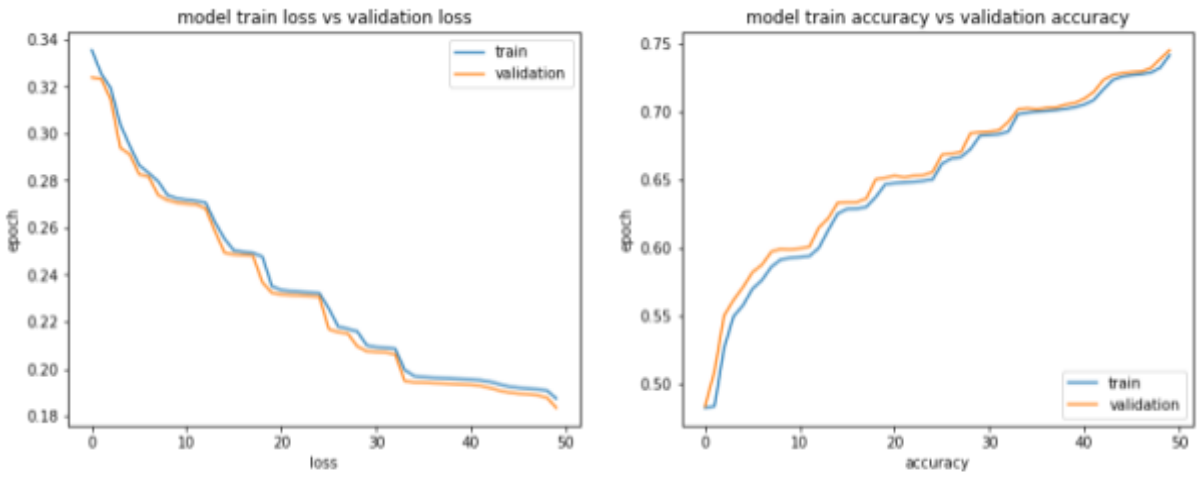
При извикване на summary се получава следната статистика за модела:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 35)          1435

 max_pooling1d (MaxPooling1D  (None, 250, 35)          0
 )

 conv1d_1 (Conv1D)           (None, 250, 35)           4935

 max_pooling1d_1 (MaxPooling  (None, 62, 35)           0
 1D)

 bidirectional (Bidirectiona  (None, 140)              59360
```

```
1)

dense (Dense)                (None, 35)                4935

dropout (Dropout)            (None, 35)                0

dense_1 (Dense)              (None, 29)                1044

=================================================================
Total params: 71,709
Trainable params: 71,709
Non-trainable params: 0
_________________________________________________________________
```

Огромна част от параметрите се намират в LSTM слоя, нищо ново. Възможно е обаче невроните в напълно свързаният слой да не са достатъчно много за да поемат капацитета на LSTM слоя, което може да доведе до затруднено учене.

Невронната мрежа се пуска за трениране с абсолютно същите настройки. Резултатите след тренирането са изобразени на фиг. 3.6.



Фигура 3.6. Размерът на невронната мрежа последва в затруднение при тренирането.

За 50 епохи невронната мрежа не успява да достигне плато, но същевременно не проявява признаци на overfitting. Вероятно причината затова е точно малкият размер на последния слой на невронната мрежа. Този експеримент ще бъде проведен на ново, само че този път в последния слой броят на единиците ще бъде удвоен на 70.

```
hidden_layer = Conv1D(filters = 35, kernel_size = 24, padding = "same",
activation = “relu”)(input_layer)
hidden_layer = MaxPooling1D(pool_size = 5, strides = 5)(hidden_layer)
hidden_layer = Conv1D(filters = 70, kernel_size = 10, padding = "same",
activation = “relu”)(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Bidirectional(LSTM(units = 70))(hidden_layer)
hidden_layer = Dense(units = 70, activation = “relu”)(hidden_layer)
```

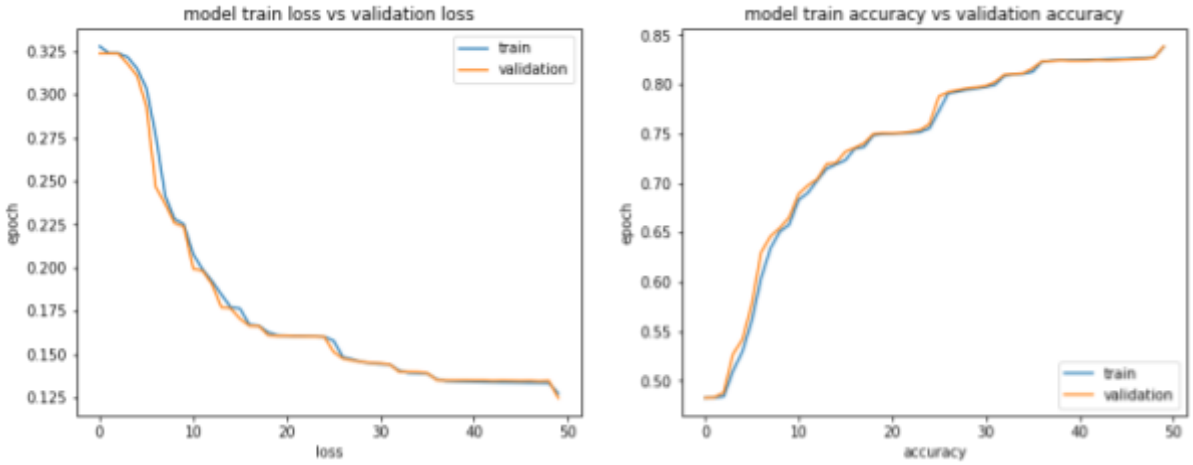
Броят параметри не се очаква да се промени особено много, тъй като се удвояват само тези при последния слой.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 35)          3395

 max_pooling1d (MaxPooling1D  (None, 200, 35)          0
 )

 conv1d_1 (Conv1D)           (None, 200, 70)           24570

 max_pooling1d_1 (MaxPooling  (None, 50, 70)           0
 1D)

 bidirectional (Bidirectiona  (None, 140)              78960
 l)
```

```
 dense (Dense)                (None, 70)                9870

 dense_1 (Dense)              (None, 29)                2059

=================================================================
Total params: 118,854
Trainable params: 118,854
Non-trainable params: 0
_________________________________________________________________
```

Като допълнителна настройка при метода за оптимизация Adam() параметърът β_1 ще бъде със стойност 0,95 - т.е да отчете момента на импулса на движение по градиентите от последните 20 итерации, вместо от последните 10.

След трениране на невронната мрежа се получават резултатите от фиг. 3.7.



Фигура 3.7. Моделът този път достига плато, но като че ли все още грешката при валидация намалява заедно с тази при трениране.

Резултатите показват, че невронната мрежа все още е в състояние да научава зависимости между данните, защото не се забелязва overfitting. Хипотезата, че последният слой не беше достатъчно голям, се оказа

правилна. Оттук може да се заключи, че ще е необходим дори още по-голям по брой параметри модел. За следващият такъв се променят единствено броят единици за всеки един от слоевете.

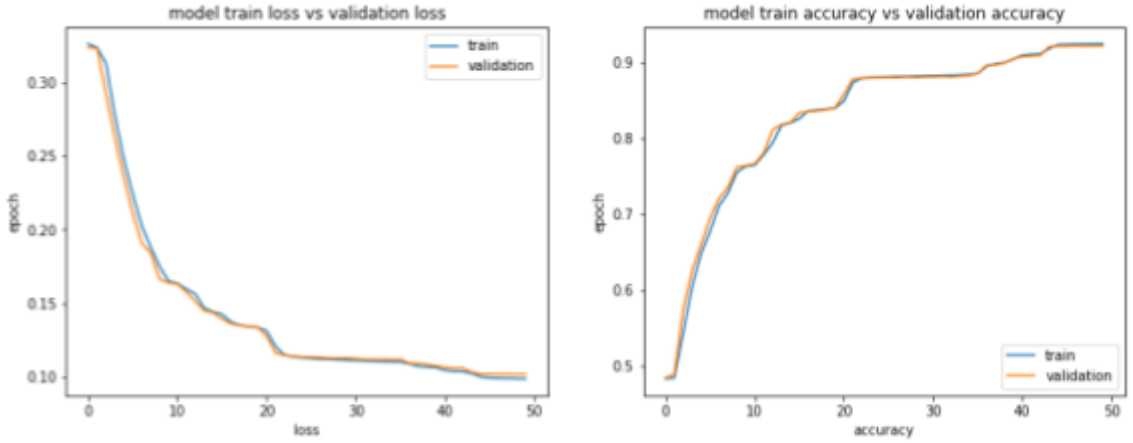
```
hidden_layer = Conv1D(filters = 50, kernel_size = 10, padding = "same",
activation = “relu”)(input_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Conv1D(filters = 100, kernel_size = 4, padding = "same",
activation = “relu”)(hidden_layer)
hidden_layer = MaxPooling1D(pool_size = 4, strides = 4)(hidden_layer)
hidden_layer = Bidirectional(LSTM(units = 100))(hidden_layer)
hidden_layer = Dense(units = 100, activation = “relu”)(hidden_layer)
```

Новата статистика за модела е следната:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 50)          2050

 max_pooling1d (MaxPooling1D  (None, 250, 50)          0
 )

 conv1d_1 (Conv1D)           (None, 250, 100)          20100

 max_pooling1d_1 (MaxPooling  (None, 62, 100)          0
 1D)

 bidirectional (Bidirectiona  (None, 200)              160800
 l)

 dense (Dense)               (None, 100)               20100

 dense_1 (Dense)             (None, 29)                2929

=================================================================
```

```
Total params: 205,979
Trainable params: 205,979
Non-trainable params: 0
_________________________________________________________________
```

Новият модел има близо 60% по-голям капацитет в параметри спрямо предходния. Резултатите от тренирането са изобразени на фиг. 3.8.



Фигура 3.8. Моделът образува плато при около 92% точност както върху трениращи, така и върху валидационни данни.

Тренирането на модел с по-голям капацитет се оказа правилния подход. За 50 епохи моделът достигна 92,38% точност върху трениращи данни и 92,12% върху валидационни такива. При извикване на `multilabel_model.evaluate()` моделът показва следният резултат:

```
 3125/3125 [==============================] - 58s 18ms/step - loss:
0.1018 - AUC: 0.9211
[0.10182563215494156, 0.9210681319236755]
```

При последващо оценяване върху тестови данни моделът показва 92,11% точност. Със сигурност може да се заяви, че този експеримент е успешен и достига достатъчно висок резултат, като върху генерирани данни

моделът не проявява склонност към overfitting. Нека обаче да не се забравя, че данните са генерирани, а не реални. Разликата между реалните и генерираните данни е, че при реалните данни разпределенията при трениращите и тестовите множества от данни не са абсолютно идентични, като за тази цел разработчика може да проведе тестване на хипотези с T-тест, ANOVA тест и тест на Колмогоров-Смирнов. Въпреки това тежестите на модела ще бъдат запазени за да послужат за основа на по-нататъшни експерименти.

3.4. Използване на невронни мрежи от смесен тип върху реални данни

В следствие на 3.3 е наличен обучен върху 29 транскрипционни фактора модел, който се справя много добре и върху валидационни, и върху тестови данни. Следващата логична стъпка е да се провери как се справя един такъв модел върху реални данни.

За тази цел са подготвени нуклеотидни последователности от хромозоми. Една съществена разлика между реалните данни и тези от 3.3 е в броя на наличните транскрипционни фактори. При генерираните данни те бяха 29. Тук те са 8. Това е съществена разлика в изходния формат на невронната мрежа, тъй като тренираният от 3.3 модел произвежда масив с големина 29 елемента, а конкретните данни изискват той да възпроизвежда такъв с големина 8 елемента.

Това всъщност не е толкова голям проблем, колкото би изглеждал на пръв поглед. В света на дълбокото машинно обучение съществува концепцията за трансферно обучение, където се преизползват вече тренирани модели, създадени за решаването на подобни задачи. Понякога

преизползваните модели искат само лека настройка, дотрениране или смяна на няколко слоя в архитектурата си.

В конкретния случай е необходима смяна на изходния слой. Тъй като невронната мрежа вече има предишен опит върху 29 транскрипционни фактора, то новата задача може да бъде разгледана като подзадача на 3.3. Смяната на изходния слой просто ще пренастрои модела към нов изходен формат, а "пътечките" които се активират при наличието на останалите 21 активационни фактора в скритите слоеве на модела, ще бъдат неактивни.

Зареждането на модела става с извикване на `load_model()` в keras, където се подава пътят към файла с тежестите на модела.

```
multilabel_model = load_model('/content/drive/MyDrive/Machine
Learning/Thesis/copy_convrnn_multilabel_classification_29_factors_92_AUC.h5')
```

Следваща стъпка е замяната на изходния слой. Обектът `multilabel_model` притежава атрибут `layers`, с помощта на когото е възможно индексирането на невронната мрежа по слоеве. Всеки такъв слой има атрибут `output`, който връща изходът на съответния слой. По този начин - чрез индексация и достъпване атрибутите на всеки слой е възможно връзването, замяната и дори вкарването вътре на нови слоеве във вече съществуваща невронна мрежа.

За да се замени изходния слой е необходимо да се достъпи последния скрит такъв и новия слой да бъде свързан към неговия изход. Понеже капацитета на вече тренираната невронна мрежа е много по-голям от това, което се изисква в конкретната задача, между изходния слой и последния скрит такъв ще бъде добавена Dropout регуляризация. Моделът се компилира със същите настройки като този в 3.3.

```
new_hidden_layer = Dropout(0.2)(multilabel_model.layers[-2].output)
```

```
new_hidden_layer = Dense(units = 8, activation = 'sigmoid',
name='dense_1')(new_hidden_layer)

new_multilabel_model = Model(multilabel_model.input, new_hidden_layer)

new_multilabel_model.compile(
    loss = BinaryCrossentropy(),
    optimizer = Adam(beta_1 = 0.95),
    metrics = [AUC(name = "AUC", multi_label=True)]
)
```

Моделът е построен и компилиран. Остава да бъде дотрениран върху новите данни. Но ако моделът бъде директно пуснат в този формат, то всички негови параметри също ше започнат да се тренират наново, а целта е всъщност да се дотренира само новият изходен слой. Както стана ясно по-рано, слоевете си имат атрибути. Друг такъв атрибут е trainable. Това е флаг, който позволява параметрите на даден слой да бъдат “замразени” по време на трениране - т.е. да не се променят. Най-лесният начин това да стане е чрез итерация. Поради тази причина при инстанцирането на новия изходен слой по-горе му бе зададено специфично име, което ще бъде използвано за флаг при проверката дали даден слой трябва да бъде замразен или не.

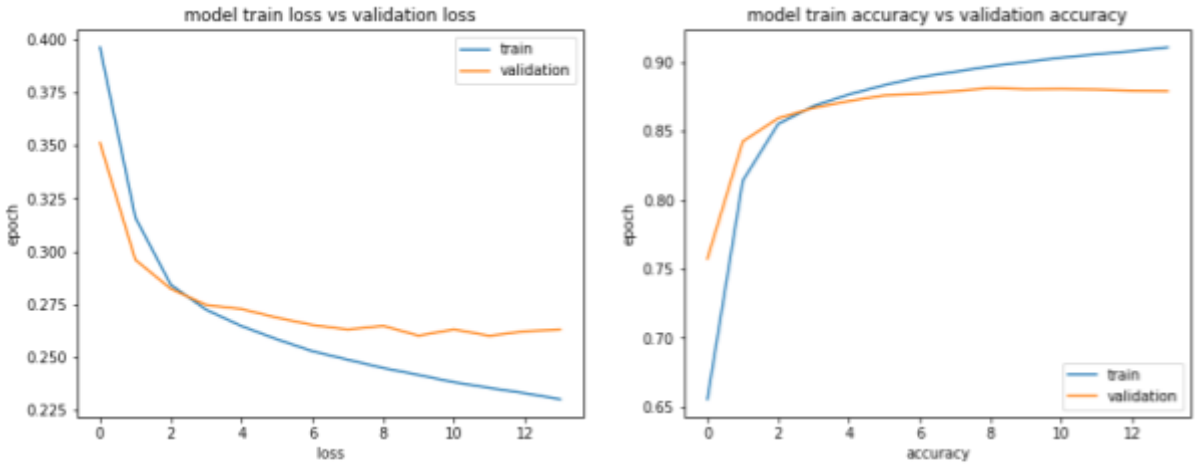
```
for layer in new_multilabel_model.layers:
  if layer.name != 'dense_1':
    layer.trainable = False
```

Извиква се статистика за новият модел.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 50)          2050
```

```
max_pooling1d (MaxPooling1D  (None, 250, 50)          0
)

conv1d_1 (Conv1D)           (None, 250, 100)          20100

max_pooling1d_1 (MaxPooling  (None, 62, 100)          0
1D)

bidirectional (Bidirectiona  (None, 200)              160800
l)

dense (Dense)               (None, 100)               20100

dropout (Dropout)           (None, 100)               0

dense_1 (Dense)             (None, 8)                 808

=================================================================
Total params: 203,858
Trainable params: 808
Non-trainable params: 203,050
_________________________________________________________________
```

Замразяването е било успешно, единствените параметри за трениране принадлежат на изходния слой. Настройките за трениране са същите като в 3.3. Резултатите от тренирането са отразени във фиг. 3.9.



Фигура 3.9. Въпреки наличният Dropout, твърде параметризираната мрежа е в състояние да постигне overfitting. Въпреки това моделът постига изключително високи резултати, използвайки само част от наученото.

На невронната мрежа са и необходими 10 епохи за да започне overfitting. Въпреки това точността върху трениращи данни стига 91%, а върху валидационни - 88%, преди да започне да се влошава. Резултатите при оценка върху тестови данни е следният:

```
2960/2960 [==============================] - 50s 17ms/step - loss:
0.2763 - AUC:8766
 0.[0.2763398289680481, 0.8766348361968994]
```

Резултатът при тестови данни е 87,66% и не се различава съществено от резултата при валидационни данни. Имайки предвид, че моделът ползва сравнително малка част от капацитета си при класификацията на записите, резултатът може да се счита за сравнително висок. Експериментът върху реални данни може да се счита за успешен, невронната мрежа - сравнително точна и със способност да генерализира достатъчно добре върху реални данни, съдържащи различен брой ZNF фактори.

4. Ръководство за потребителя

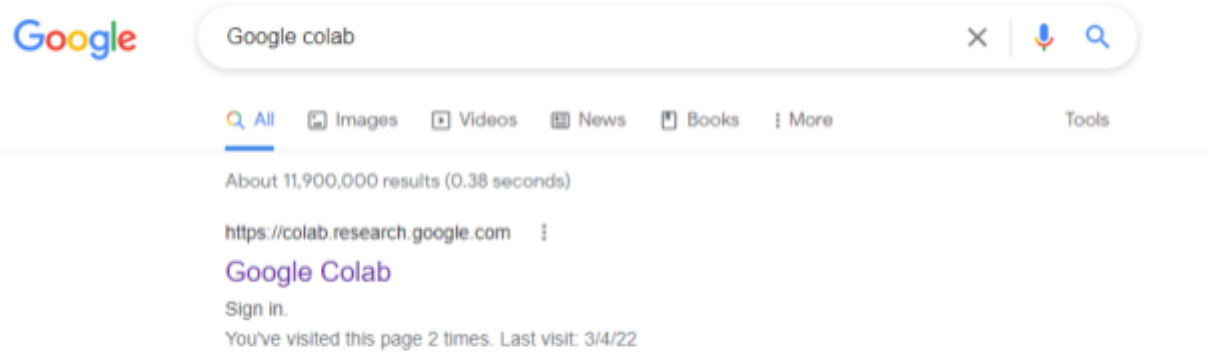
В тази глава ще бъде обяснено как потребителят да настрои и използва работната среда Google Colab, да достъпва необходимите ресурси и да запазва новополучените резултати.

4.1. Настройки и начин на използване на Google Colab

Google Colab (съкратено от Colaboratory) както бе обяснено още във втора глава, е облачната услуга, която Google предоставят за хората, които искат да се занимават с машинно обучение. За целта Google предоставят свои машини с централни процесори и графични ускорители, които потребителите могат да използват безплатно, но за ограничен период от време всеки ден. Като хранилище за ресурсите и данните, които разработчикът зарежда, използва и записва, се използва Google Drive, но съществува възможност за качване и сваляне на ресурси от машината на разработчика. Colab позволява писането на изпълним код на Python, като същевременно предоставя възможността за писане на стилизиран текст и прилагането на мултимедия под формата на изображение и видеа. Писането на код се случва в отделни клетки и всяка от тези клетки е възможно да се изпълнява независимо от останалите.

Първата стъпка към използването на Google Colab (както и всички останали услуги на гугъл), е потребителят да си създаде профил в Google. Подробно въведение към създаването и менажирането на Google профил може да бъде намерено тук.

След създаването на профил потребителят може да пристъпи към използването на Google Colab. Първият резултат при търсене на Colab в интернет търсачката (фиг. 4.1) служи за достъпване на Colab. От фиг. 4.1 се вижда, че алтернатива на търсенето на Google Colab в уеб търсачката е достъпването на https://colab.research.google.com/.

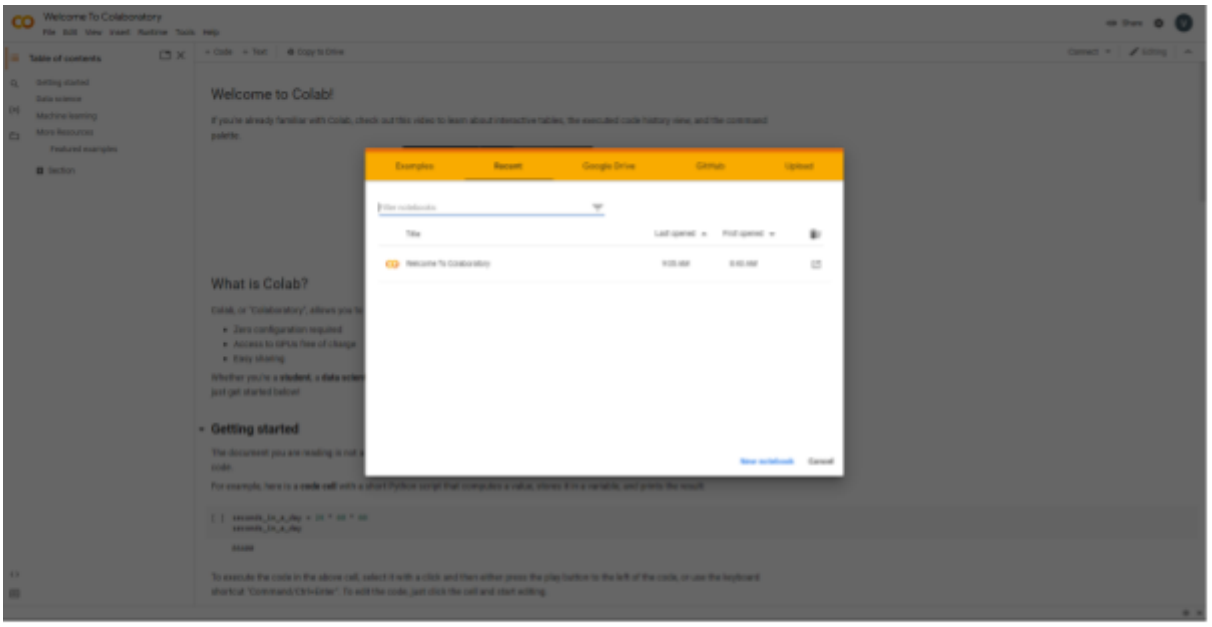


Фигура 4.1. Търсеният от потребителя резултат за достъпване на Colab

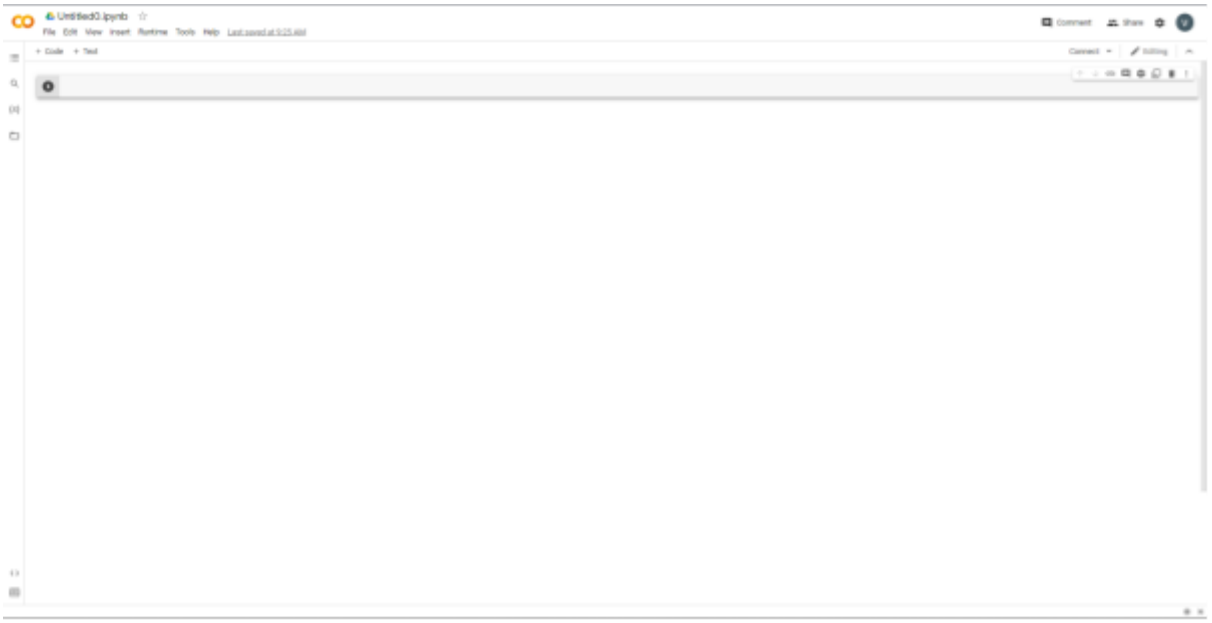
При достъпване на Google Colab от чисто нов профил екрана на потребителя би изглеждал като във фиг. 4.2. Отваря се прозорец, откъдето потребителят може да зарежда jupyter notebooks от Google Drive или да достъпва вече създадените от него в Recent. Други опции за зареждане на jupyter notebooks са през GitHub, където ресурсите се достъпват чрез линк, или пък могат да бъдат зареждани от локалната машина на потребителя през Upload. Examples съдържа няколко генерирани от Google документа, които служат за въведение в използването на различни аспекти на Colab и комбинирането им с външни ресурси.

При натискане на Cancel ще се зареди генериран от Google jupyter notebook, който представлява кратък преглед на възможностите на работната среда с изредени линкове към ресурсите, споменати в Examples.

Създаването на нова инстанция на jupyter notebook, където се случва цялата разработка на модели, става с натискане на бутона New notebook (фиг. 4.3).

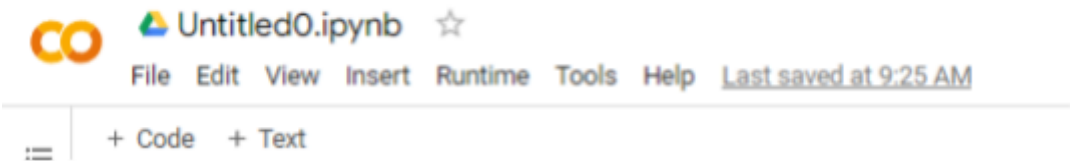


Фигура 4.2.



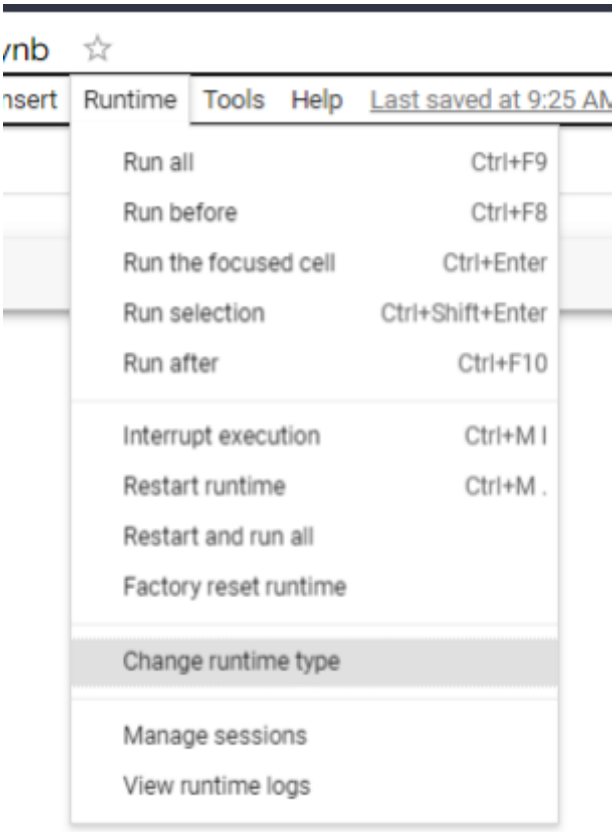
Фигура 4.3. Така изглежда чисто нова инстанция на Jupyter Notebook в Google Colab.

Операциите върху инстанцията на Jupyter Notebook се извършват чрез лентата с менюта в горния ляв ъгъл (фиг. 4.4)



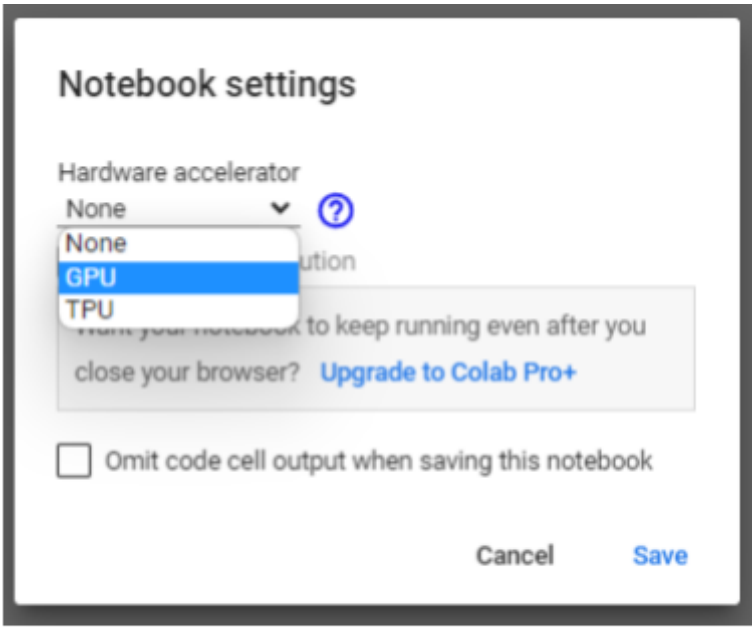
Фигура 4.4. Лентата с менюта в Google Colab

Първата важна настройка върху работната среда е върху какъв ускорител тя работи. Както бе споменато на няколко пъти, за обучението на невронни мрежи централният процесор не е достатъчен. Нужен е хардуер, който да може да извършва огромно количество математически операции с дробни числа. Начинът, по който се избира графичен ускорител, е от менюто Runtime. При натискането на Runtime ще се появи падащо меню с няколко опции. Необходимите настройки се падат в полето Change runtime type.



Фигура 4.5.

В средата на екрана ще се появи меню (фиг. 4.6), в което първото поле за настройка е Hardware accelerator. При натискането на полето за въвеждане ще се появи падащо меню с три опции: None, GPU и TPU. None означава използване само на централен процесор, което не ни върши особена работа. За опция се избира GPU, което е графичен ускорител. За запазване на промените се натиска бутон Save.



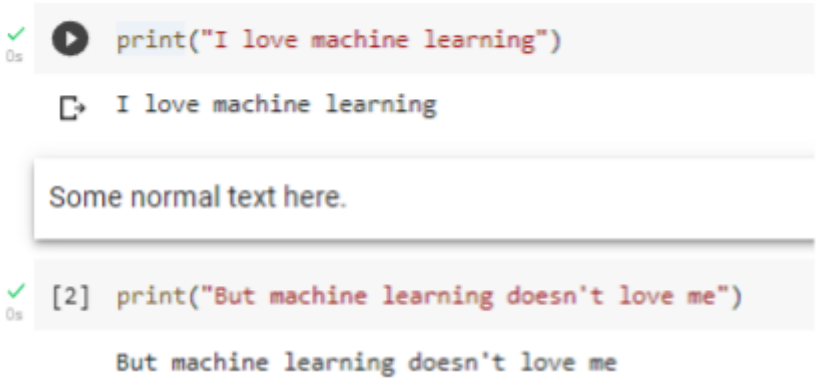
Фигура 4.6

При запазване на промените автоматично се заделя ресурс и средата става активна. Когато е активна в горния десен ъгъл ще бъдат изобразени състоянието на инстанцията, както и заетите от инстанцията ресурси (фиг. 4.7). Изключително важно е да се спомене, че ако средата стои активна без да се извършват процеси в нея, тя ще бъде автоматично изключване след 20-30 мин. неактивност, ресурсите и ще бъдат освободени, а всеки постигнат резултат - загубен, ако не е запазен локално или на облак. Времето за използване на GPU за ден варира и то много - това зависи изключително много от колко активно се използва графичният ускорител и колко често.

Ако потребителят използва изключително често графичните ускорители, то времето за изчакване за ползване постепенно ще се увеличава, а времето за ползване - намалява. В някои случай при твърде честата употреба на графични ускорители времето за ползване е едва час.

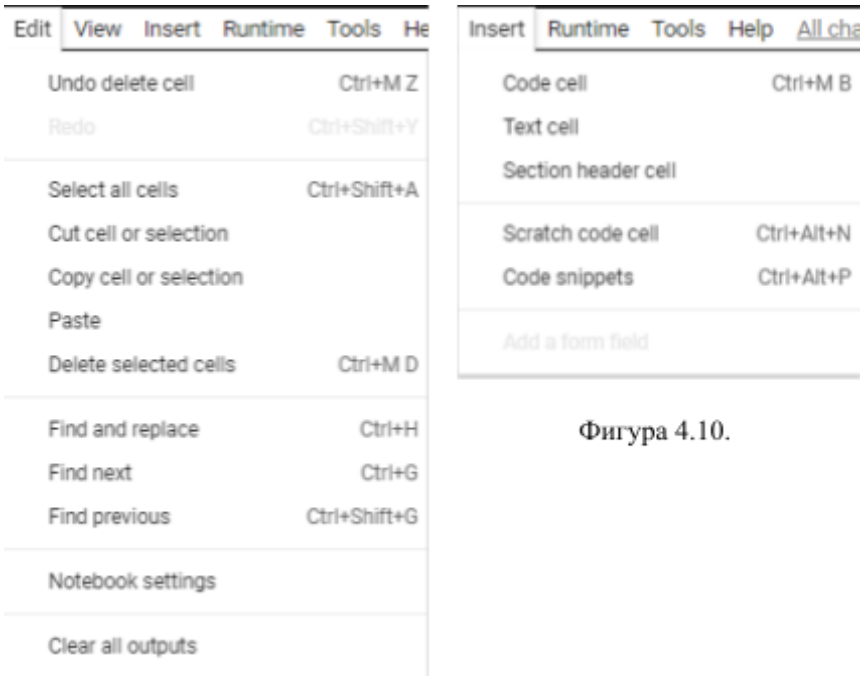
Фигура 4.8

След настройката може да се пристъпи към използването на средата. Във фиг. 4.4 може да се забележи, че под лентата с менютата има два бутона Text и Code. С тях създаваме нови клетки, в които съответно може да се пише съответно текст (да се вмъкват линкове, стилизация, вкарване на мултимедия) и изпълним код. Всяка клетка код може да се изпълнява самостоятелно и да извежда самостоятелно резултати (фиг. 4.9).



Фигура 4.9. Отделните клетки с код се изпълняват независимо една от друга. Вмъкването на клетки с текст между тях не нарушава тяхната изпълнимост

Повече команди за редактиране и форматиране на клетки може да се намери в менютата Edit и Insert (фиг. 4.10).



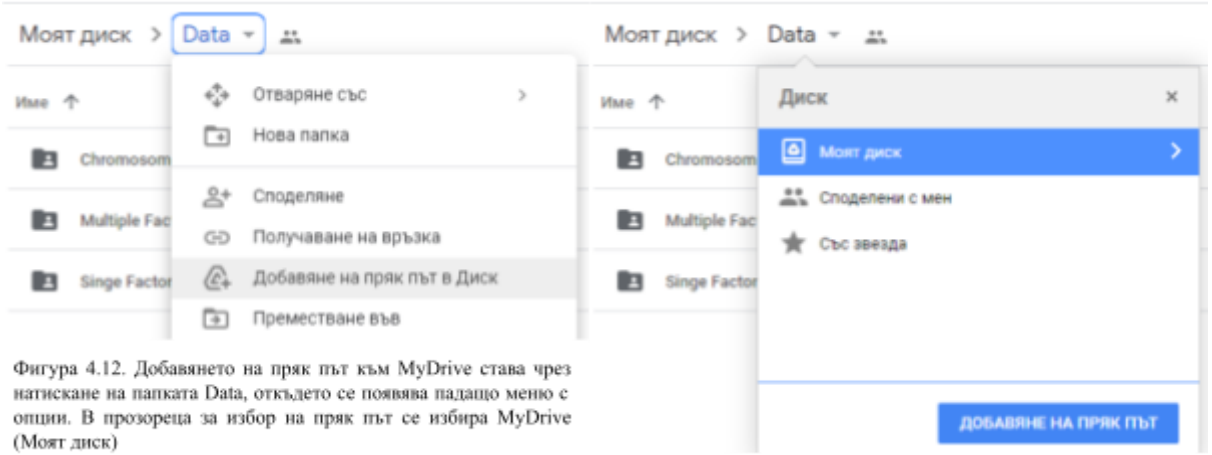
Фигура 4.10.

След като потребителят е приключил работа е време за запазване на резултатите. Има няколко опции за запазване на файла - в Google Drive, в Github, свален на локалната машина като чист Python файл или като Jupyter Notebook. Тези опции са изредени в менюто File (фигура 4.11).

Фигура 4.11

4.2. Зареждане на претренираните модели и .ipynb файловете

Като първа стъпка преди реалното зареждане на моделите и .ipynb файловете трябва да има налични данни. Данните са достъпни през споделен Google Drive линк в референциите[16]. За да може зареждането на файловете в готовите Jupyter Notebooks тетрадки да не произведе грешка потребителят задължително трябва да сложи пряк път за достъп към споделената папка с данни в MyDrive (фиг. 4.12). При навигиране обратно до MyDrive (Моят Диск) трябва да е налична папка, подобна на тази във фиг. 4.13.



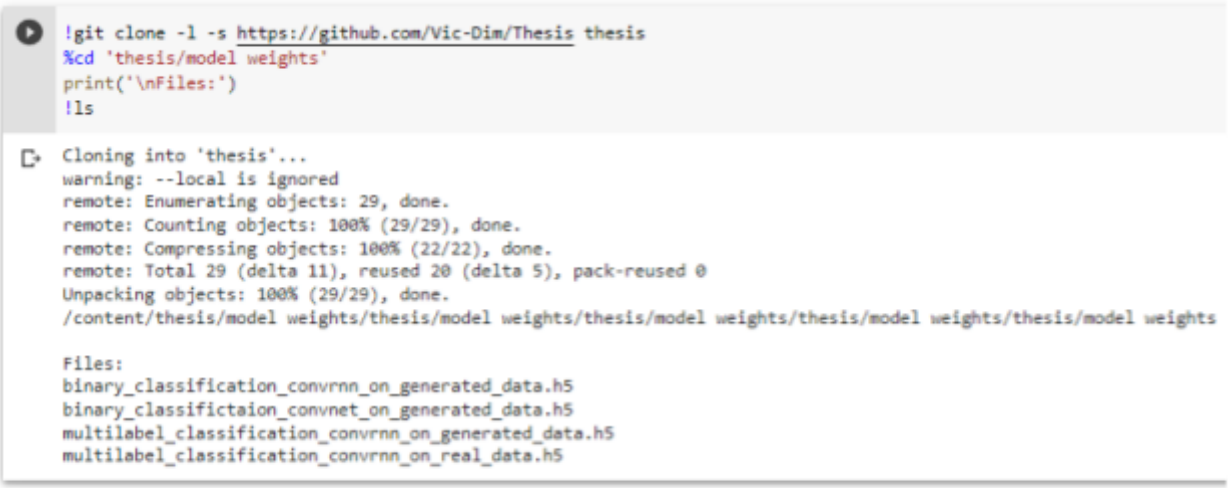
Фигура 4.12. Добавянето на пряк път към MyDrive става чрез натискане на папката Data, откъдето се появява падащо меню с опции. В прозореца за избор на пряк път се избира MyDrive (Моят диск)

Фигура 4.13

След като добавянето на прекия път към данните е готово потребителят трябва да навигира до Github хранилището с файловете, което също ще бъде добавено към референциите[17]. Следвайки този линк потребителят ще попадне в Github хранилището на проекта. В папката notebooks са налични всичките .ipynb файлове, а в model weighs - съхранените модели в .h5 файлов формат.

По-трудни за зареждане са моделите, защото се изискват няколко операции по свалянето на .h5 файловете от Github. Затова ще бъде предоставен snippet, с който автоматично ще може да се достъпват файловете с тежестите и да бъдат директно отваряни в Colab. Този snippet се поставя в клетка с код в Colab, след което се изпълнява (фиг. 4.13)

```
!git clone -l -s https://github.com/Vic-Dim/Thesis thesis
%cd 'thesis/model weights'
print('\nFiles:')
!ls
```



Фигура 4.13. При изпълнение на кодът Github хранилището се клонира локално и се навигира до файловете с тежестите. Самите файлове са изброени след "Files:"

За зареждането на който и да е от моделите е необходимо само да се копира неговото файлово име от изхода на клетката и да се постави в кавички като аргумент на метода `load_model` (фиг. 4.14). Оттам нататък потребителят може да експериментира с готовия модел както си пожелае.

```
model = load_model('binary_classification_convrnn_on_generated_data.h5')
model.summary()
```

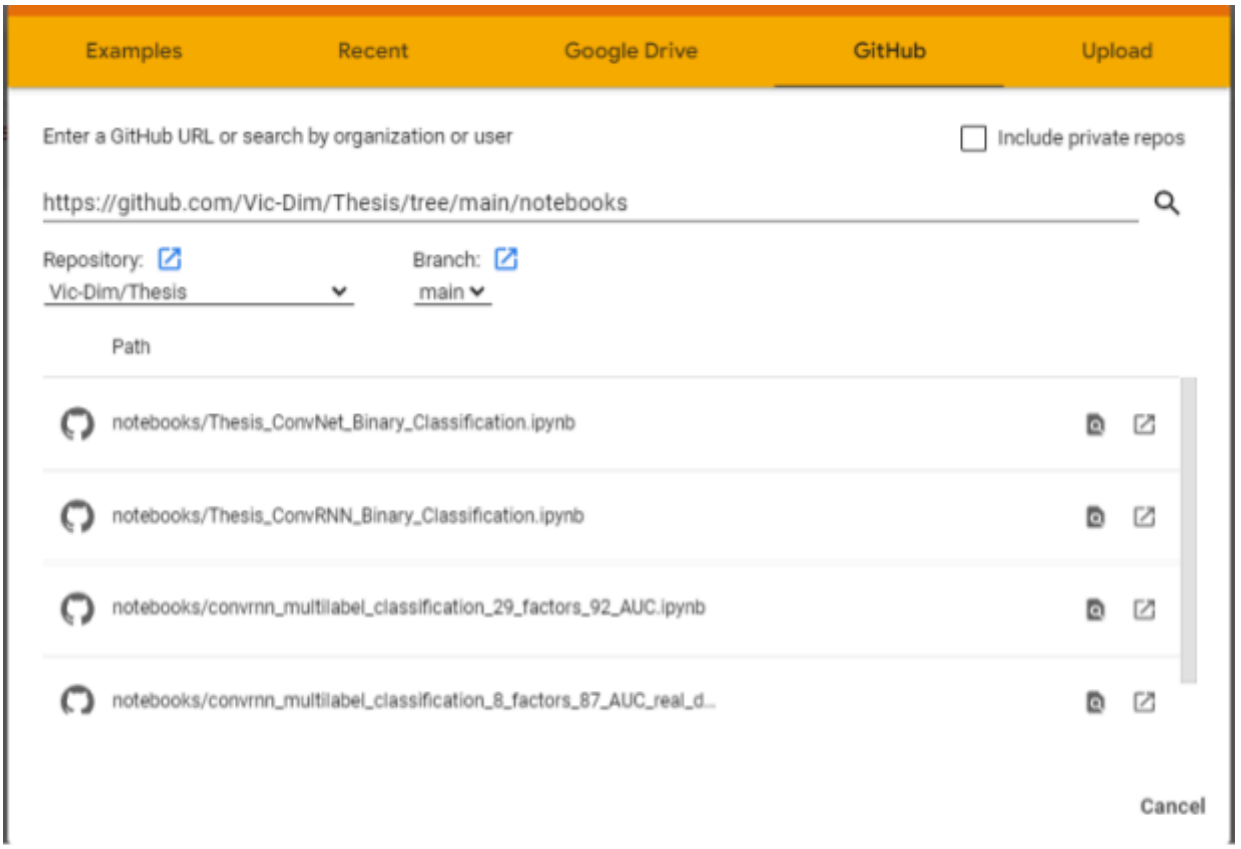
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1000, 4)]         0

 conv1d (Conv1D)             (None, 1000, 16)          656

 max_pooling1d (MaxPooling1D  (None, 250, 16)          0
 )

 conv1d_1 (Conv1D)           (None, 250, 32)           2080

 max_pooling1d_1 (MaxPooling  (None, 62, 32)           0
 1D)

 bidirectional (Bidirectiona  (None, 64)               16640
 l)

 dense (Dense)               (None, 32)                2080

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 21,489
Trainable params: 21,489
Non-trainable params: 0
_________________________________________________________________
```

Фигура 4.14. Моделът се зарежда от свалените локално файлове с тежести. За демонстрация допълнително е изведена статистиката на модела

Ако пък потребителят иска да проследи хода на създаването на модела може лесно да отиде в папката с .ipynb файловете, да селектира линкът към папката и да го копира (фиг. 4.15). Обратно в Colab, при отваряне на менюто File е налична опцията Open Notebook, което ще изведе същият екран като

във фиг. 4.2. Отваряйки Github секцията на това меню ще се появи поле за търсене, в което се поставя копираният линк. При търсене по поставеният линк ще се изредят всичките налични .ipynb файлове в папката (фиг. 4.16), след което потребителят може да отвори желаният файл.

Фигура 4.15. Линкът към Github директорията с .ipynb файловете



Фигура 4.16. Изредени .ipynb файлове от директорията в Github

Заключение

В тази дипломна работа бе подробно разгледана сферата на дълбокото машинно обучение, нейните аспекти, процесите при тренирането на невронните мрежи, проблемите свързани с тях и вече съществуващи оптимизации.

Придобитите знания за дълбокото машинно обучение бяха приложени при решаването на задача от сферата на геномиката, свързана с предвиждането на нуклеотидни секвенции, чувствителни към серията от транскрипционни фактори ZNF. Бяха разгледани и имплементирани няколко архитектури на модели от дълбокото машинно обучение, които се справиха със сравнително висок резултат върху горепоставената задача, трениращи се върху синтетични и реални данни, които съдържат множество транскрипционни фактори.

Въпреки че изследването на архитектури постигна целите, поставени от изискванията към тази дипломна работа, разработването на модел от дълбокото машинно обучение не спира дотук. Бъдещното развитие на този проект би представлявало вместването на разработените дотук модели в приложение и изследване на начините за оценка и сравнение на нуклеотидни последователности, съдържащи транскрипционни фактори ZNF.

Списък на използваните термини

1. Learning rate - скорост градиентното спускане
2. ReLU (Rectified Linear Unit) - няма точен превод
3. Multiclass classification - многокласова класификация
4. Multilabel classification - класификация по множество признаци
5. Vanishing and exploding gradients - Експлодиращи и чезнещи градиенти
6. Underfitting - недообученост
7. Overfitting - наизустяване
8. Bias - наклонност
9. Variance - разпръснатост
10. Residual connections - остатъчни връзки
11. Adam (Adaptive Momentum Estimation) - Адаптивна оценка на момент на импулса
12. Exponentially Weighted Moving Averages - Експоненциално претеглени пълзящи средни стойности
13. RMSprop (Root Mean Square Propagation) - няма точен превод
14. Stride - разкрач
15. LSTM (Long Short-Term Memory) - Дългосрочна краткотрайна памет
16. GRU (Gated Recurrent Unit) - Управляема рекурентна клетка
16. ANOVA (Analysis of variance) test - тест за сравняване на разликите в разпръснатостите на две или повече разпределения
17. HDF (Hierarchical Data Format) - йерархичен формат за запис и съхранение на данни

Референции и използвана литература

1. François Chollet (2021). Deep Learning with Python, Second Edition
2. Thomas Cover (1965). Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition
3. DeepLearning.AI - Neural Networks and Deep Learning
4. DeepLearning.AI - Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization
5. Abien Fred M. Agarap (2018). Deep Learning using Rectified Linear Units (ReLU)
6. Colin Wei et al. (2020). The Implicit and Explicit Regularization Effects of Dropout
7. Kaiming He et al. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
8. Xavier Glorot & Yoshua Bengio (2010). Understanding the difficulty of training deep feedforward neural networks
9. He et al. (2015). Deep Residual Learning for Image Recognition
10. Diederik P. Kingma & Jimmy Lei Ba (2014). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION
11. Fangyu Zou et al. (2018). A Sufficient Condition for Convergences of Adam and RMSProp
12. Hochreiter & Schmidhuber (1997). Long Short-Term Memory
13. KyungHyun Cho et al. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
14. DeepSea
15. DanQ
16. Линк към данните в Google Drive
17. Линк към Github хранилището на проекта

Съдържание

Увод **1**

Въведение в невронните мрежи **2**
История на дълбокото машинно обучение 2
Разлики между стандартното и дълбокото машинно обучение 4
Дълбоко машинно обучение 6
Слоевете в една невронна мрежа 8
Основни математически операции и начин на трениране 10
Активационни функции 15
Често срещани проблеми при дълбокото машинно обучение 19
Конволюционни невронни мрежи 29
Рекурентни невронни мрежи 36

Технологии, използвани за реализация на невронни мрежи **46**
Налични технологии и развойни среди 46
Постановка на проблема и вече налични решения 49

Изследване на архитектури на невронни мрежи за разрешаването на поставените проблеми **52**
Използване на конволюционни невронни мрежи за класификация върху два класа 52
Използване на невронни мрежи от смесен тип за класификация на два класа 63
Използване на невронни мрежи от смесен тип за класификация на повече от два класа 66
Използване на невронни мрежи от смесен тип върху реални данни 75

Ръководство за потребителя **80**
Настройки и начин на използване на Google Colab 80
Зареждане на претренираните модели и .ipynb файловете 87

Заключение **91**

Списък на използваните термини **92**

Референции и използвана литература **93**