МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета програмирање и програмски језици

Визуелизација детектованих карактеристика високог нивоа у конволуцијским неуралним мрежама

Ученик
Владимир Миленковић, 4д

Ментор
Петар Величковић

Београд, мај 2017.

Садржај

1

Увод

Средином двадесетог века, иако је целокупно програмирање, па самим тим и скуп проблема који су решиви, кренуло знатно да напредује, постојала је велика група проблема који нису могли бити решени.

Слика 1.1: Проблем препознавања слова – сва слова на слици су заправо слова А

Пошто знамо да постоји бесконачно различитих рукописа, немогуће је направити стандардизовани алгоритам који би успешно разликовао написано слово А од осталих слова.

Али, за човека, то је много лак посао. Сваки човек би, готово моментално, препознао да су сва слова која се налазе на овој слици различити облици слова А. Како то наш мозак препознаје?

Слово А би се могло детално описивати, од којих дужи се састоји, који су углови између њих и сл., али јасно је да ништа не може да објасни шта је слово А боље од заправо гледања тог слова. Уколико видимо 100 различитих верзија тог слова, верујем да нам неће бити проблем да за неко слово препознамо да ли је А или није, а кад

би нам неко дао само карактеристике, сумњам да бисмо могли тако лако да одредимо.

Дакле, једна од најбитнијих ствари које наш мозак ради је учење. Учење, у смислу да добијамо много различитих улаза, и за сваки имамо резултат, да ли јесте или није то што проверавамо (у овом случају одређено слово), и после довољног броја виђених слова, бићемо врло сигурни за сваки следећи знак, да ли је А или није.

Исту технику желимо да применимо и на наш програм, да направимо наш "мозак" који ће прво бити без икакве информације о било чему, и да га тренирањем на довољном броју улаз-излаз парова, припремимо да може тачно да одговара на задато питање. Такав, искодиран мозак, се зове **неуралне мрежа**.

Јасно је да ће наша неуралне мрежа заправо запажати неке дубље карактеристике слова А, које су толико комплексне да их је немогуће лако искодирати, али су ипак исте и у потпуности одређују слово, јер у супротном не бисмо били у могућности да решимо овај проблем.

Када су прве неуралне мреже направљене, добијени су сјајни резултати. Наши "мозгови" су радили врло тачно (наравно не са ефикасношћу коју данашње неуралне мреже имају), али свакако много боље од било каквих "простих" програма које је пре тога било могуће смислити, где под "простим" програмима подразумевам оне које имају унапред задат скуп инструкција које треба да изврше.

Једно питање је пак мучило програмере. Видимо да то нешто, та нека мрежа, производи тачне резултате, и виђена је шанса за велики продор у целокупном програмирању, али на питање "Шта наша мрежа заправо ради?" нико није био у стању да дâ тачан одговор. Нико то није знао.

Зато се дошло на идеју да неуралну мрежу "пустимо да се изврши"над неком сликом, како бисмо могли да уочимо неке визуелне ефекте, јер је лакше уочити шта се заправо десило са неком сликом него са неким децималним бројевима, за које заправо не знамо ни шта су, ни шта представљају.

Програм који је направљен од стране Гугла, који користи конволуцијску неуралну мрежу како би нашао и појачао одређене облике на некој слици, зове се *DeepDream*, и како се он прави биће тема овог рада. "Нуспојава" која се десила коришћењем *DeepDream*-а на сликама је претварање слика у халуцуногене, као у нашим сновима, са невероватним и немогућим ефектима који изгледају одлично, толико добро да се чак постављало питање да ли су слике које су произведене овим програмом уметност.

Слика 1.2: DeepDream пример 1

Слика 1.3: DeepDream пример 2

Слика 1.4: DeepDream пример 3

2

Теоријски темељи

2.1 Типови учења

Машинско учење је област вештачке интелигенције која се бави алгоритмима који нису експлицитно препрограмирани, него "уче" како треба да се понашају на основу тренинга. Зато је то најбитнија ствар код машинског учења, и она представља давање много парова (улаз/коректан излаз за задати улаз), где ми очекујемо од наше машине да научи како треба да се понаша у одређеној ситуацији.

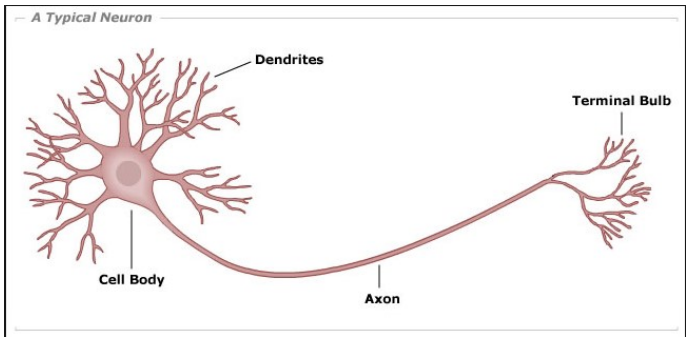
Имамо три типа "послова" које желимо да наша машина извршава:

- **Несупервизирано учење** – желимо да нашим улазним подацима доделимо карактеристике помоћу којих се они описују
- **Учење побољшавања** – желимо да наша мрежа враћа онај "потез"који максимизује неку унапред дефинисану награду
- **Супервизирано учење** – имамо парове (улаз/ излаз), и желимо да нађемо неку функцију која што је могуће боље апроксимира праву функцију која мапира улаз у излаз. *Ово је једина врста учења коју неуралне мреже директно могу да раде.*

2.2 Неурони у природи

Научници нису сигурни како наш мозак заправо функционише, али сва испитивања указују на то да наш мозак није препрограмиран да извршава одређене функције, заправо једини урођен алгоритам нашег мозга је да уме да учи. Главни задатак ове целокупне гране програмирања је да покуша да "схвати" тај алгоритам и примењује га у разним ситуацијама.

Наш мозак се састоји од мноштва испреплетаних неурона (реда величине стотине милијарди), исповезиваних тако да је су излази једних неурона повезани са улазима



Слика 2.1: Пример неурона

других.

Као што видимо са слике, један неурон се састоји од:

- **Дендрита** – који служе да бисмо добили информацију од других делова нервног система(намеће се да ће ово бити улаз)
- **Аксона и наставака аксона** – који служе да бисмо информацију коју наш неурон "израчуна" послали у следеће неуроне на путу
- **Тела неурона** – које служи да бисмо добијене информације на неки начин обрадили и "послали"даље

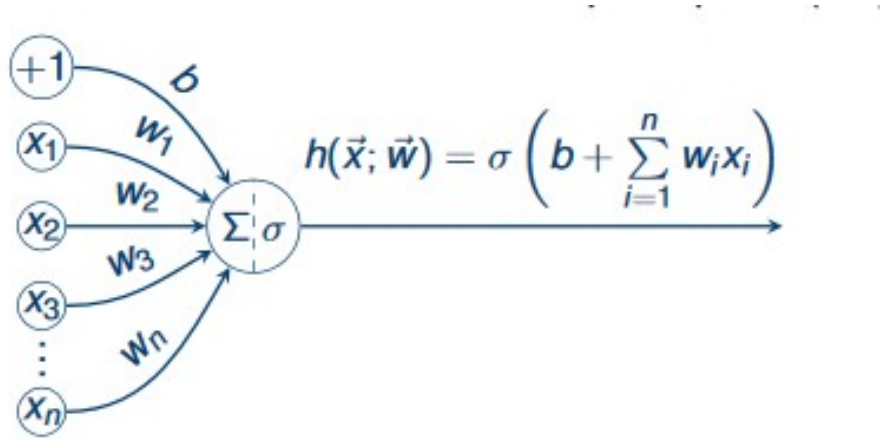
Јасно је да је наш задатак овде да исповезујемо наше неуроне на начин на који нама то одговара, и да направимо функцију за неурон која из улазних информација рачуна информацију коју треба проследити даљим неуронима.

2.3 Компјутерска репрезентација неурона

Све што смо до сада звали информацијама ћемо прогласити реалним бројевима, јер је то једини тип података са којим наша машина може да ради. Нека наш неурон има N улаза, и то ћемо означити низом бројева X_i. Једна од најлакших (а и најучинковитијих) стратегија како рачунамо резултат је тако што правимо линеарну комбинацију свих улаза, евентуално јој додамо неку константну вредност, и онда применимо активациону функцију на резултат који смо добили. Мало формалније:

У овој репрезентацији, W_i ћемо звати коефицијентима нашег неурона, број b ћемо звати константном неурона, и активациону функцију ћемо означавати са σ. Јасно је, да при оваквој имплементацији неурона, тренирање неурона се своди на рачунање најбољих могућих коефицијената за дати неурон.

Једна јако битна ствар при дизајнирању неурона је избор активационе функције. Једна од главних примена активационе функције је то да наш резултат не буде пука



Слика 2.2: Компјутерска репрезентација неурона

линеарна комбинација наших улаза, јер се многи проблеми једноставно не могу описати линеарно.

2.3.1 Избор активационих функција

Чести избори активационих функција су:

- $\sigma(x) = x$ – *Identity* функција, не користи се тако често јер се онда не постиже губљење линеарности
- $\sigma(x) = \max(0, x)$ – *ReLU* функција, једна од најчешће коришћених функција
- $\sigma(x) = \tanh(x), \sigma(x) = \frac{1}{1+exp(-x)}$ – Сигмоидне функције, друга од ових је тзв. *Logistic* функција

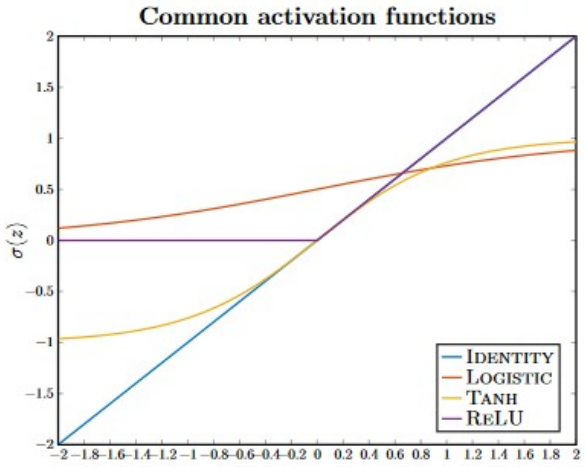
2.4 Изградња неуралне мреже

Ако имамо готове неуроне, лако је направити неуралну мрежу – само користимо излазе претходних неурона као улаз за следеће.

То можемо урадити на два начина:

- **Напредујуће** – Тако да наш добијени граф нема циклусе
- **Рекурентно** – Тако да наш добијени граф има циклусе

При изради овог програма, опредељујем се за напредујуће. Један од лепших начина да направимо наш граф је да га изделимо у слојеве, тако да су излази сваког слоја



Слика 2.3: Компарација активационих функција

заправо улази следећег. Овим добијамо граф подељен у слојеве, где ћемо први слој звати *улазни слој*, а последњи *излазни слој*. Слојеве који нису ни први, ни последњи ћемо звати *дубоким слојевима*.

Још једна од згодних ствари које можемо урадити је то да излаз неког неурона преусмеримо ка свим неуронима следећег слоја (нема разлога зашто би се неурони неког слоја испрва разликовали). Овим добијамо *вишеслојну неуралну мрежу*, коју ћемо користити при изради *DeepDream*-а.

2.5 Шта се налази у слојевима?

Уколико имамо мрежу са пуно слојева, можда можемо да испратимо шта се дешава у првом, али готово је немогуће схватити и разумети шта се дешава у њеним дубоким слојевима. Сваки следећи слој садржи још комплекснију репрезентацију нашег улаза. Зато се дошло на идеју да улази буду слике, јер при раду са сликама лакше схватамо шта се дешава него са гомилом бројева. Модели са сликама имају најлакшу визуелизацију.

Проблем са сликама је то што су слике јако меморијски захтевне. Уколико бисмо чували сваки пиксел слике као засебан улазни податак, то би резултирало са $H * W * D$ параметара по неурону у првом скривеном слоју, где су H, W, D ширина, висина, и дубина слике, респективно. Под дубином слике, подразумевам број параметара за описивање једног њеног пиксела (при обичним сликама, $D = 3$ (РГБ систем)).

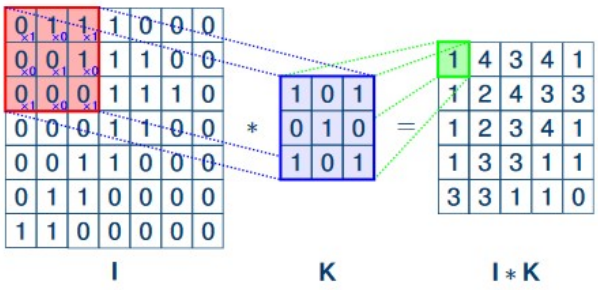
Због тих проблема, треба смањити димензије слике што је више могуће, покушавајући да сачувамо и искористимо њену просторну структуру.

2.6 Конволуцијски слој

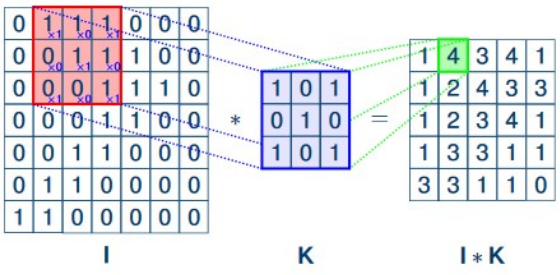
Дефинишимо D малих матрица (рецимо $3 * 3$) и назовимо их **кернелима**.

Преносимо кернел i преко i-тог слоја слике I, и суму производа елемемата кернела и елемента преко ког је сада тај елемент кернела записујемо на одређено место у решењу. То радимо за сваки од D слојева.
Оваквом операцијом, структура слике долази до изражаја - што су ближи пиксели, то више утичу један на други!

$$(I * K)_{x,y} = \sum_{i=1}^{H} \sum_{j=1}^{W} K_{i,j} * I_{x+i-1,y-j+1}$$



Слика 2.4: Пример конволуције 1

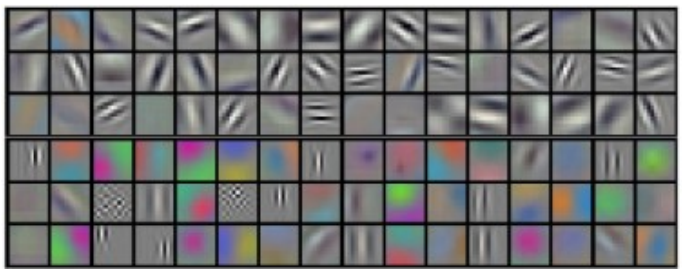


Слика 2.5: Пример конволуције 2

При конволуцији, могу се додати још неки параметри:

- **Дубина** – Број филтера, тј. број сетова кернела који траже одређену особину слике (утиче на величину решења)
- **Корак** – Величина за коју померамо наш кернел, што је већи то је величина резултата мања, подразумевано је 1
- **Допуна нулама** – Број који означава са колико редова/колона нула треба допунити нашу слику, да бисмо применили кернел на њу (користи се да би димензије слике остале исте, јер некад незнатно смањење нема поенте)

С обзиром на то да је величина кернела мала, корисне информације није могуће извући из кернела дубљих од оних у првом слоју. Испоставља се да је, у великој већини случајева, **први слој кернела заправо детекор ивица**.

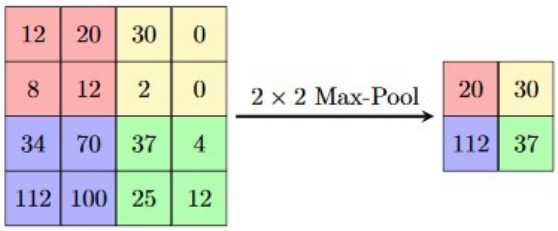


Слика 2.6: Први слој кернела као детектор ивица

2.7 Смањивање слике

У резултату конволуције, за дати кернел, велику вредност имаће она поља где је вероватноћа да је изражена баш та одређена особина велика. Зато, при смањивању слике, има смисла да памтимо максимуме у одређеним подручјима. Један од најпознатијих алгоритама за смањивање слике, који ћу користити, је 2×2-смањивање.

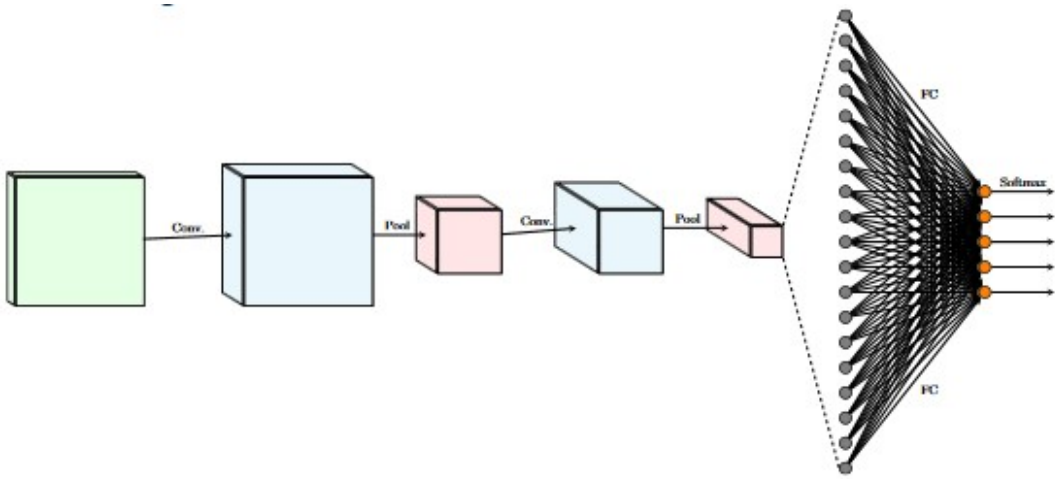
Од $2 * N \times 2 * M$ матрице, добије се $N \times M$ матрица, тако што се из сваке 2×2 подматрице узме максимум та 4 елемента.



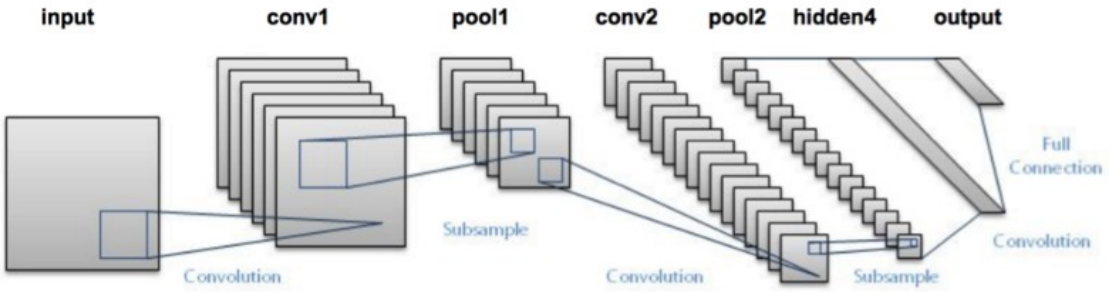
Слика 2.7: 2x2 смањивање

2.8 Комбиновање конволуција и смањивања

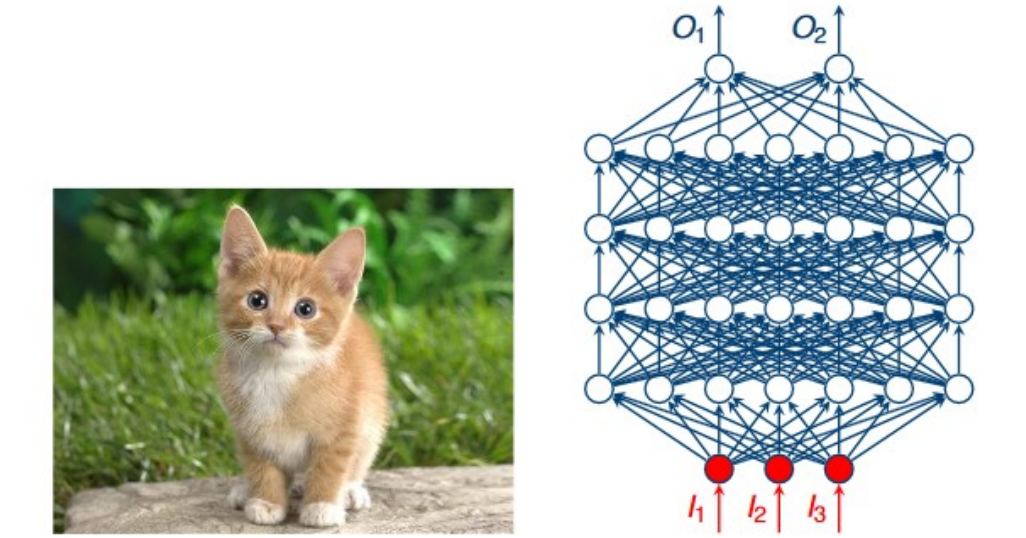
Једно од правила које се испоставило да даје најбоље резултате: **Треба повећавати дубину (број конволуција) како се ширина и висина слике смањују.**



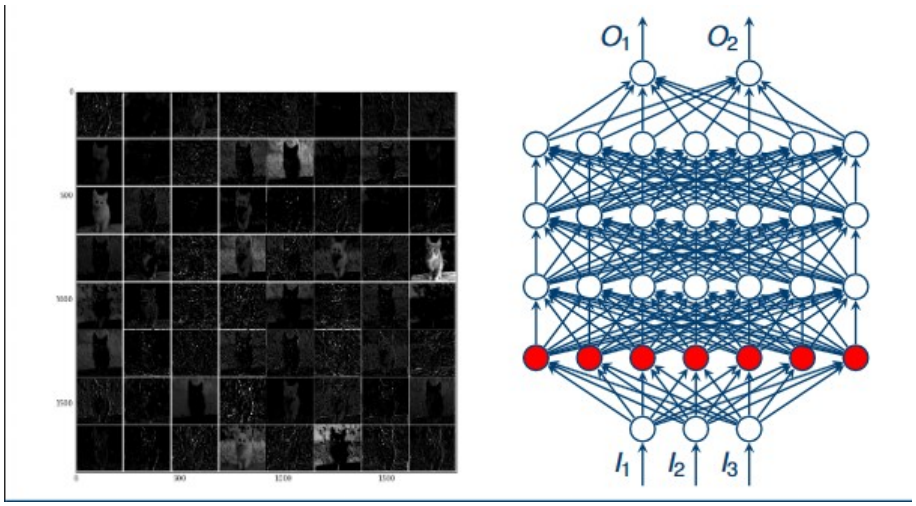
Слика 2.8: Комбиновање



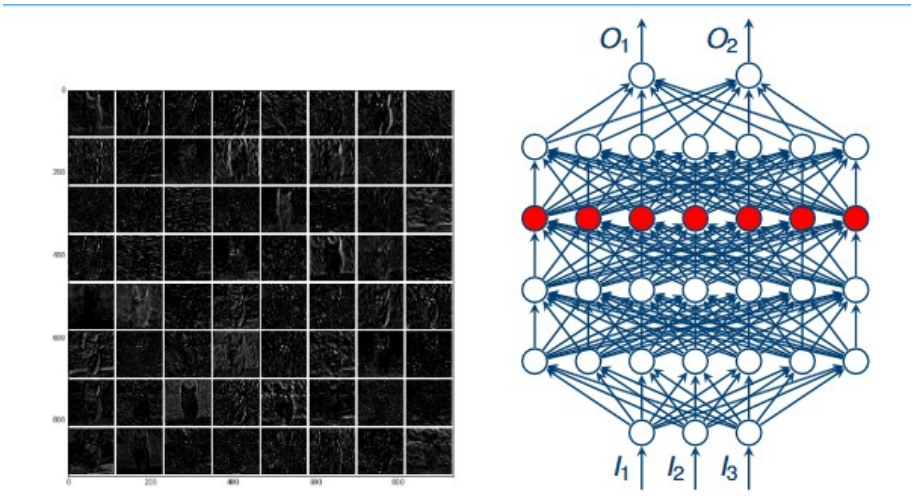
Слика 2.9: Комбиновање



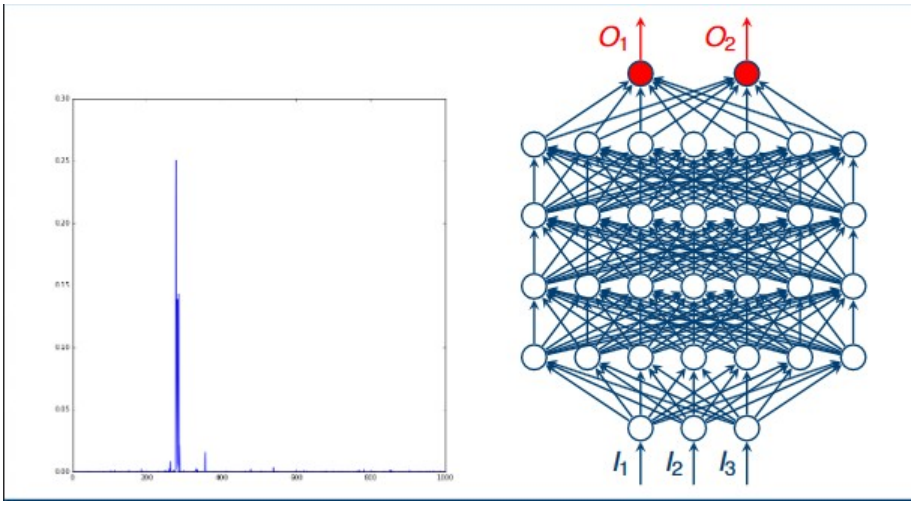
Слика 2.10: Улазни слој



Слика 2.11: Први слој(понаша се као детектор ивица)



Слика 2.12: Неки дубоки слој



Слика 2.13: Излазни слој

2.9 Диференцијабилност

Једна од најважнијих особина која се користи при учењу наше мреже да даје резултат који нама треба, и без које не би било могуће радити овакве ствари, јесте **диференцијабилност**. Доказаћу диференцијабилност функција које ће нам бити потребна у даљем раду.

2.9.1 Диференцијабилност неурона

Пошто је наш аргумент за активациону функцију линеарна комбинација улазног вектора и вектора коефицијената плус константна, тривијално је диференцијабилно и по улазу и по коефицијентима и по константи.

Свака од наведених активационих функција је диференцијабилна[1].

Пошто је излаз неурона композиција двеју горенаведених функција (које су обе диференцијалне), и цео неурон је тривијално диференцијабилан по сваком параметру.

2.9.2 Диференцијабилност везивања неурона

Пошто ми користимо излаз једног неурона као улаз следећег, опет имамо композицију две неуронске функције, за које смо већ доказали да су диференцијабилне, па ће и спојени неурони бити диференцијабилни.

Напомена: *Приметимо да је за диференцијабилност мреже небитан начин на који су неурони везивани, па је мрежа диференцијабилна, каква год њена структура била.*

2.9.3 Диференцијабилност смањивања

Иако смањивање, као сама функција, не делује диференцијабилно, ту функцију можемо написати као:

$$pool(\vec{x}) = \begin{cases} x_1 & \forall i \neq 1, x_1 > x_i \\ x_2 & \forall i \neq 2, x_2 > x_i \\ \dots & \\ x_n & \forall i \neq n, x_n > x_i \end{cases}$$

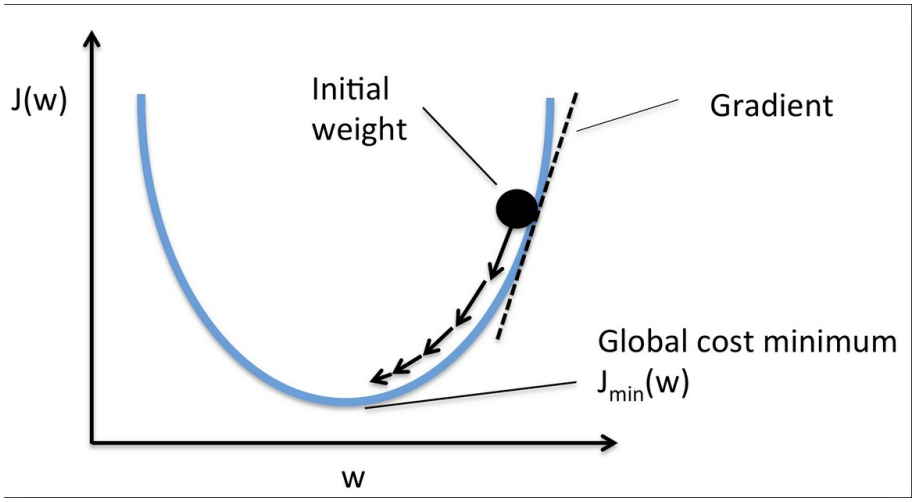
Дакле, када овако напишемо, за сваки вектор $\vec{x}$ функција је тривијално диференцијабилна[2].

[1] Осим ReLU у нули, што је мање битно

[2] Осим када максимум није јединствен, али онда се може увести правило изузетка.

2.10 Градијентни спуст

Дефинишимо диференцијабилну **функцију губитка** $\mathcal{L}(\vec{x};\vec{w})$. Она ће, на неки начин, бити мера колико се наш резултат разликује од правог резултата (објашњено шта се ради супервизираним учењем). Ми можемо истренирати нашу мрежу да минимизује нашу функцију губитка, тако што се, у сваком кораку, прати негативан смер извода функције губитка.



Слика 2.14: Приказ функционисања градијентног спуста (само са једним параметром ради лакшег сналажења)

Дакле, када се определимо за целокупну архитектуру наше мреже, одабир функције губитка је оно шта одлучује за шта ће наша мрежа бити тренирана.

2.10.1 Избор функције губитка

Навешћу неколико широко коришћених функција губитка:

- **Квадратна грешка** - Најпростија фунцкија губитка, параметар за функцију губитка је вектор коефицијената $\vec{w}$, а за улазне параметре (x_i, y_i) (x_i је улаз, а y_i је очекивани излаз) функција се рачуна као

$$\mathcal{L}(\vec{w}) = (h(\vec{x_i};\vec{w}) - y_i))^2$$

- **Квадратна грешка + тежина** - Приметимо да при квадратној грешци, нама ни на шта не утиче да ли су коефицијенти велики бројеви или не. Велики бројеви, обично, нису пожељни, па нашој функцији губитка можемо додати још један фактор:

$$\mathcal{L}(\vec{w}) = (h(\vec{x_i};\vec{w}) - y_i))^2 + \lambda||\vec{x}||^2$$

Овде, параметар λ је параметар "компромиса" између два сабирка који чине функцију густине, и променом параметра можемо мењати понашање читаве мреже.

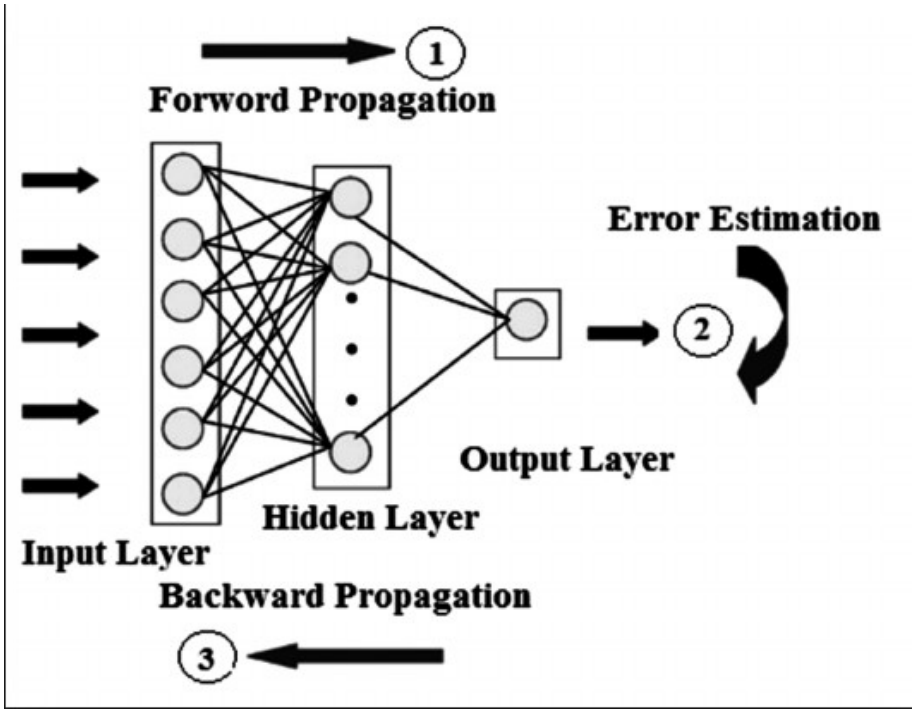
2.10.2 Пропагација уназад – (*Backpropagation*)

Пропагација уназад је јако честа метода тренирања неуралних мрежа, која се користи заједно са неком оптимизационом методом (у нашем случају, са градијентним спустом. Уско је повезана са Гаус–Њутновом методом.)

Следећи алгоритам треба извршавати све док наша мрежа не достигне жељену тачност (класификује све улазе тачно).

Алгоритам 1 Алгоритам тренирања мреже користећи пропагацију уназад

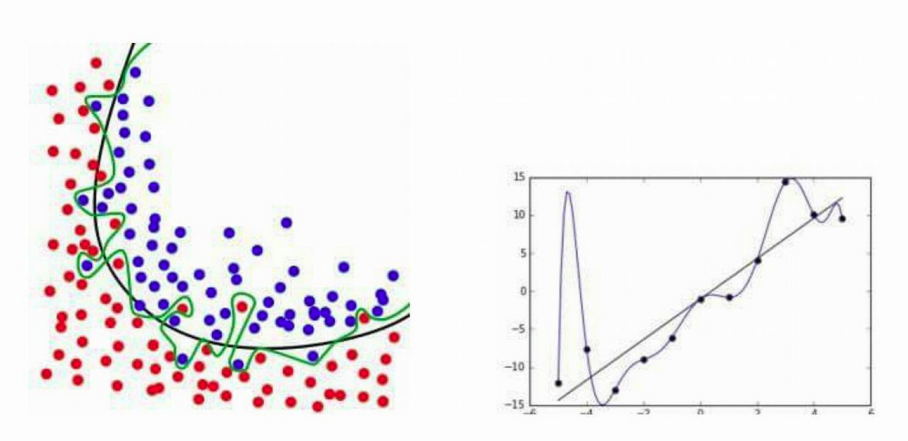
1: *иницијализовати све коефицијенте у мрежи (мали, насумични бројеви)*
2: **for each** ex in training-examples **do**
3: *prediction* ← `neural-net-output(ex)`
4: *value* ← `real-output(ex)`.
5: **for each** layer *lay* going backwards from last hidden to input **do**
6: *compute* Δw_j for all the weights in *lay*//using backpropagation
7: update network weights



Слика 2.15: Алгоритам **backpropagation**-а

2.10.3 Могуће грешке – *overfitting* и *underfitting*

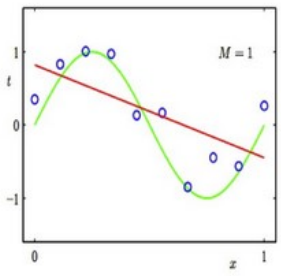
При *overfitting*-у, наш модел приказује "насумичну грешку тј. функцију са много шумова уместо реалне функције. До *overfitting*-а може доћи када је наш модел превише компликован, или кад има превише параметара у поређењу са бројем параметара који су неопходни да би се описала циљана релација. Наравно, модел у ком је дошло до *overfitting*-а има лоше предиктивне способности, као што се може видети и са следећих примера:



Слика 2.16: Приказ *overfitting*-а

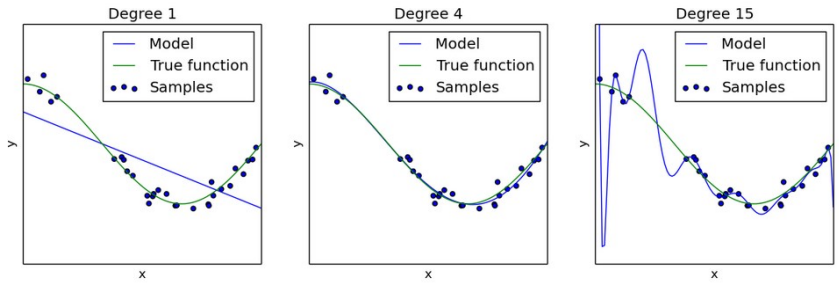
Као што видимо, иако зелена, тј. плава функција можда дају тачније одговоре на досадашњим тренинг примерима, у оба примера је црна линија бољи одабир за функцију јер има доста већу вероватноћу да тачније реши проблем за предстојеће улазе. У другом примеру, иако плава функција савршено репрезентује скуп тачака до сад, линеарна црна функција даваће боље резултате убудуће. И интерполацијом би се добило нешто што више личи на црну, него на плаву линију.

При *underfitting*-у, такође долази до лоших предиктивних способности, али из обрнутог разлога – покушавамо да уклопимо комплексну функцију у прост модел.



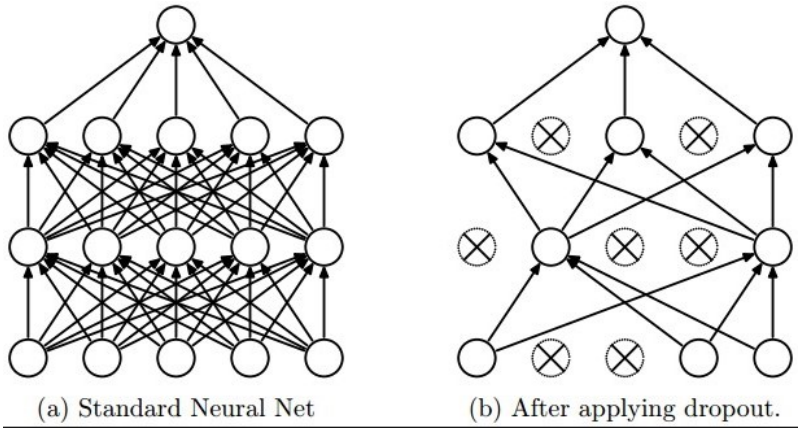
Слика 2.17: Пример *underfitting*-а, покушавање моделирања нелинеарне функције (рецимо синуса) линеарном функцијом.

Јасно је да су и *overfitting* и *underfitting* лоши и могу довести до нежељених резултата, па је јако битно проценити сложеност функције, коју желимо да апроксимирамо што је могуће боље.



Слика 2.18: Примери за *overfitting* и *underfitting* – грешке су и више него очигледне

Постоји доста начина како избећи ова два проблема, а ја ћу у својој имплементацији користити **dropout**. Параметар **dropout**-а је вероватноћа p. Сваки чвор наше мреже (сваки неурон), се са вероватноћом $1 - p$ избацује из мреже пре покретања сваког тренинг примера, и на крају извршавања, сви ти неурони се враћају у мрежу са непромењеним коефицијентима. За улазне неуроне, или се не примењује, или је вероватноћа да буде избачен доста мала, док је p за неуроне дубоких слојева углавном 0.5.

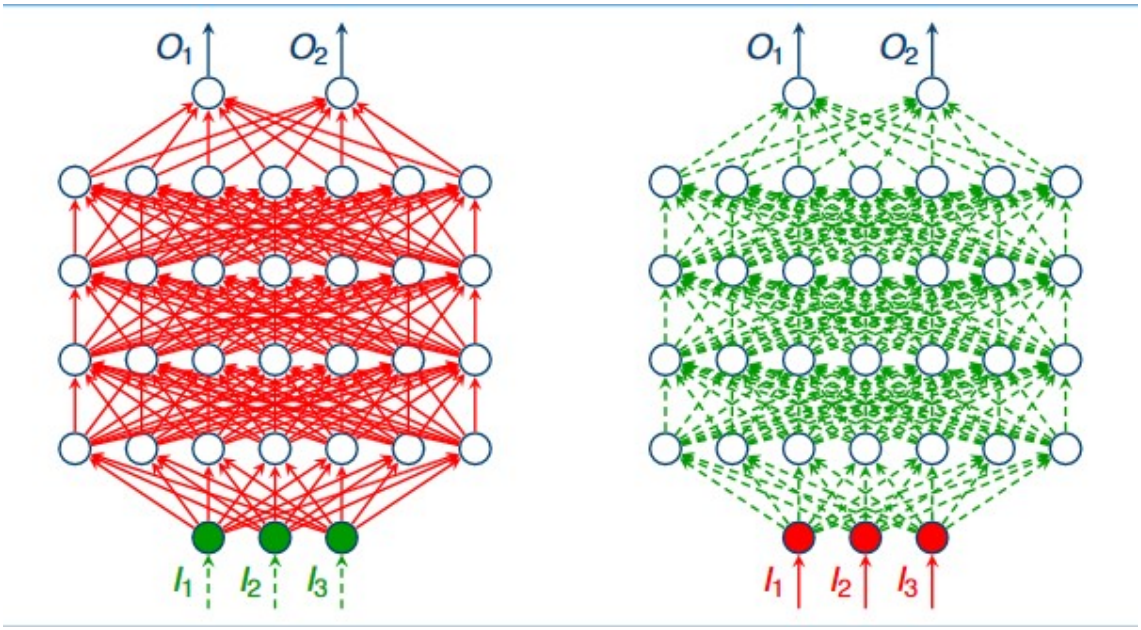


Слика 2.19: Пример **dropout**-а

Желимо да нађемо средњу вредност свих 2^N мрежа (подскупова оригиналне мреже), како бисмо смањили могућност горенаведених проблема, али ово је немогуће у случајевима када је N велико. Зато насумично избацујемо и убацујемо чворове, па је очекивана вредност баш та средња вредност. Такође, што дуже тренирамо мрежу, то ће одступање од тачне вредности бити мање.

Приметимо да смо у свим функцијама губитка до сад параметризовали само коефицијенте, док смо улаз и излаз све време чували као константе из тренинг података.

Шта би било када бисмо коефицијенте чували као константе, а покушавали да мењамо улаз, са истом идејом минимизације функције губитка?[3]



Слика 2.20: Идеја окретања проблема

[3]Ово смемо урадити јер је функција губитка диференцијална и по улазу

3

Имплементација

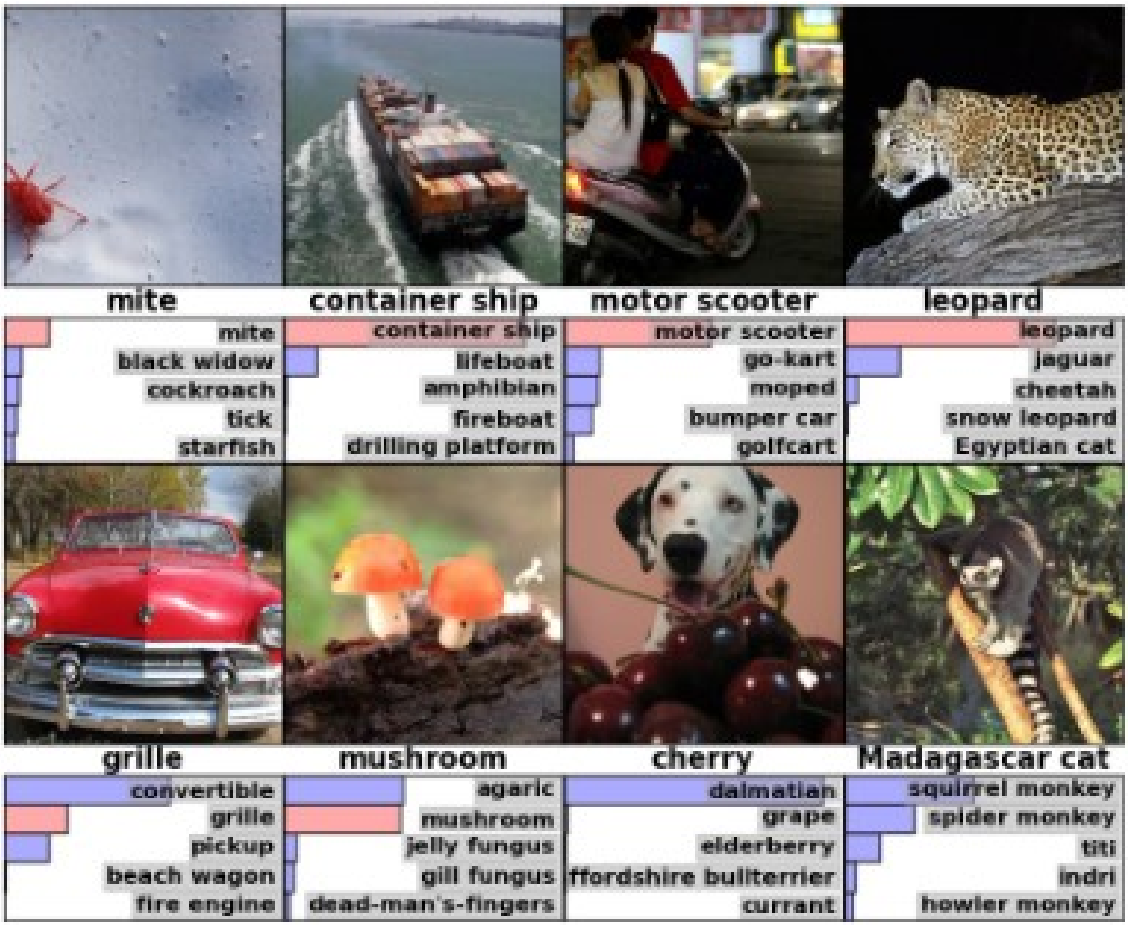
Пошто смо дошли на идеју да желимо да наше коефицијенте чувамо као константу, те коефицијенте морамо некако унапред израчунати. За тренирање сам користио *ImageNet*, а као модел мреже сам узео VGG16.

3.1 Тренирање мреже – ImageNet

ImageNet је велика база података направљена за коришћење у софтверу који се бави визуелним препознавањем објеката. Сада има више стотина хиљада различитих предмета са више од 500 слика по појму, у просеку. Многи програми користе баш ову базу за тренирање својих мрежа.

ImageNet сваке године од 2010. организује годишње такмичење у развоју софтвера (ILSVRC - *ImageNet Large Scale Visual Recognition Challenge*). На том такмичењу се мноштво тимова такмичи, покушавајући да остваре што бољи резултат у детектовању визуелних карактеристика слике.

Поред класификације слика у класе, што је вероватно први задатак овог такмичења, треба детектовати и разне друге карактеристике слика. Примери су: локализација и детекција објеката, тражење граница објеката, обрада видео снимака и детектовање објеката на њима, итд..



Слика 3.1: Пример класификације

Постоји 1000 класа у које слике треба разврстати, и посебно је битно то што постоје многе међу собом сличне класе (има преко 100 раса паса), док постоје и многе које су тотално различите (људи, воће, моторна возила). Због оволике различитости, али и сличности појединих класа, овај проблем класификације је екстремно тежак.

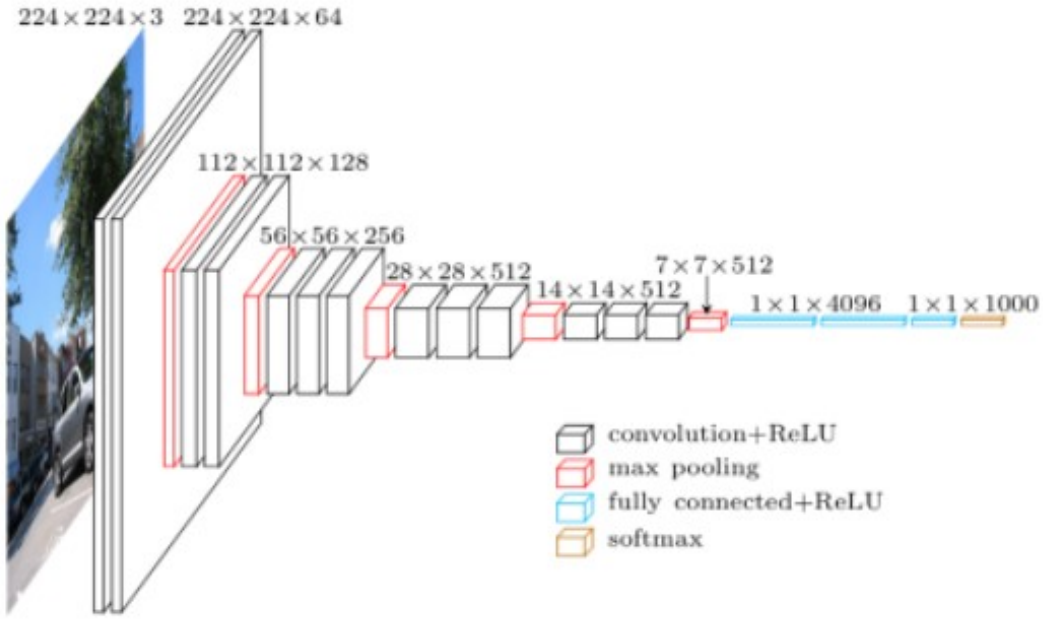
Праву револуцију при решавању овог проблема донео је **VGG** (Visual Geometry Group) са Универзитета у Оксфорду 2012. године, када је први пут употребио конволуцијску неуралну мрежу. Те године, однео је победу са више од 10% предности у осталу на све остале тимове. Од тада па до данас, сваке године се користе конволуцијске мреже. Просечан резултат човека на овом тесту је 94%, и једна од првих великих победа дубоког учења над човеком се десила 2015. године, када је победник овог такмичења престигао човечји резултат.

Коефицијенти које ћу узети за мрежу јесу коефицијенти коришћени од стране једног од победничких тимова на овом такмичењу, а затим и објављени.

Унапред тренирани модели већ постоје у свим *deep learning* библиотекама. У мојој имплементацији ћу користити *Keras* библиотеку.

3.2 Архитектура мреже – VGG16

VGG16 је модел конволуцијске неуралне мреже која је коришћена од стране тима **VGG Team** на **ILSVRC** такмичењу 2014. године.



Слика 3.2: VGG16 мрежа

Све у овој схеми је већ објашњено, конволуцијски слој, слој смањивања, и дубоки слојеви неуралне мреже, једино што је остало необјашњено је *softmax* функција.

3.2.1 *Softmax*

Softmax функција (позната и под називом **нормализована експоненцијална функција**) је функција која као улаз узима вектор од N произвољних реалних бројева $\vec{v}$, и враћа вектор $\sigma(\vec{v})$ од N бројева у интервалу $[0, 1]$ чија је сума 1. У теорији вероватноће, ова функција се користи као расподела вероватноће када постоји N могућих исхода.

$$\sigma(\vec{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{N} e^{v_j}}$$

3.3 DeepDream

Као што смо рекли, конволуциони слој ће имати велике вредности уколико наш улаз има изражене баш те карактеристике које тај слој испитује. Дакле, уколико ми држимо наше коефицијенте константним, а желимо да минимизујемо функцију губитка, наша слика ће се мењати тако да карактеристике тог слој буду све израженије и израженије.

Пример: Уколико наш слој служи за детекцију човека на слици (један слој не може детектовати тако сложене карактеристике, тако да није тако просто; узето је примера ради), итерације *DeepDream*-а ће свему што имало подсећа на човека додавати човечје карактеристике.

Као што видимо, одабиром слојева које желимо да максимизујемо мењамо како ће улазна слика бити обрађивана. Нека је S скуп слојева чије вредности желимо да максимизујемо. Имамо:

$$\mathcal{L}_{layers}(\vec{x}) = -\sum_{i \in S} ||\mathcal{A}_i(\vec{x}; \vec{w})||^2$$

где је $\mathcal{A}$ нека функција слоја, рецимо сума свих вредности у слоју. Наравно, предзнак минус је због тога што ми градијентним спустом минимизујемо вредност функције губитка, а ову функцију желимо да максимизујемо.

За слојеве, обично се бирају или само плићи или само дубљи слојеви, али може се бирати и нека мешавина. Више о резултатима за другачији одабир слојеве у поглављу Евалуација.

Приметимо да, уколико бисмо дозволили да слика буде јако светла, могли максимизовати $\mathcal{L}_{layers}$, али бисмо тиме добили слику без икаквог значења. Зато, треба некако да "казнимо" слике са много светлих пиксела. Уведимо нову функцију губитка (познату под именом L_2 функција губитка):

$$\mathcal{L}_{L_2}(\vec{x}) = ||\vec{x}||^2$$

Приметимо да нигде нисмо пазили на то да ли је слика континуална. Ако желимо да буде визуелно коректно, врло је пожељно да то не буде насумично набацана гомила пиксела. Зато уводимо континуалну функцију губитка:

$$\mathcal{L}_{continuity} = \sum_{i=1}^{N}\sum_{j=2}^{M}(x_{i,j} - x_{i,j-1})^2 + \sum_{i=2}^{N}\sum_{j=1}^{M}(x_{i,j} - x_{i-1,j})^2$$

, где су N и M ширина и висина слике. Дакле, узимамо разлику у вредностима између

свака два суседна пиксела и рачунамо суму квадрата тих разлика.

Коначно, имамо:

$$\mathcal{L}_{deepdream} = \mathcal{L}_{layers} + \lambda \mathcal{L}_{L_2} + \kappa \mathcal{L}_{continuity}$$

Као и пре, λ и κ су коефицијенти који представљају компропис између ових функција губитка (што је коефицијент испред функције већи, то нам је битнија карактеристика коју та функција одређује). Занимљиво је то, да се малим променама ових коефицијената могу догодити драстичне промене у резултату.

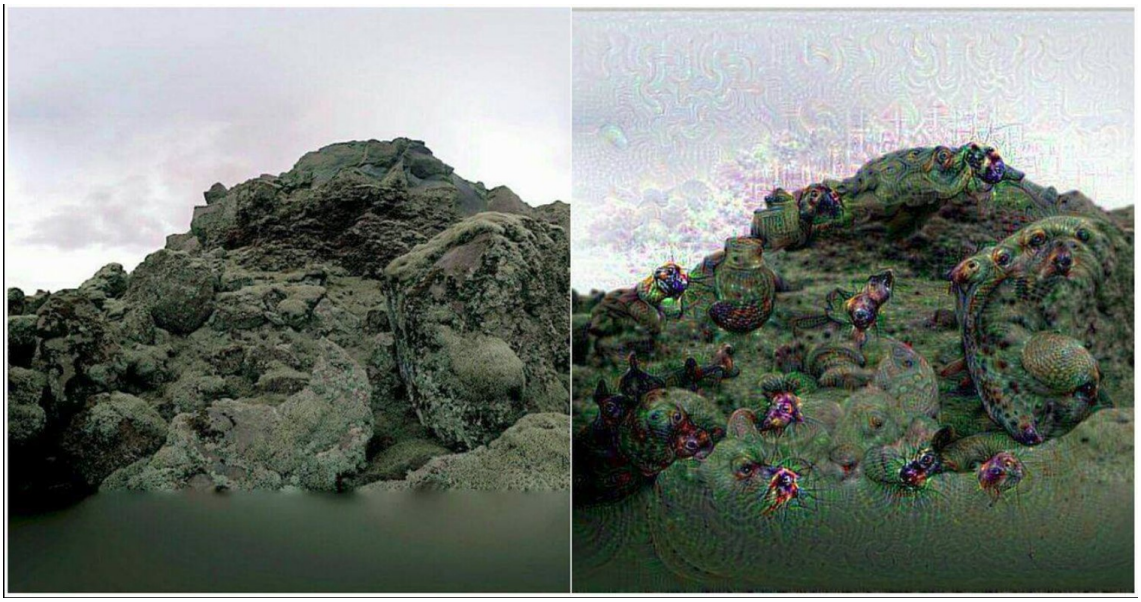
4

Евалуација

Направио сам своју андроид апликацију, која учитава слику, покреће *Python* скрипту која је на серверу, обрађује слику, и приказује је. Ево неколико резултата обраде слика помоћу моје апликације:

Напомена: Пошто, због материјалних разлога, нисам био у стању да закупим много јак сервер, фиксирао сам величину слике на 500 × 500 × 3, и процесирање једне такве слике траје око десет минута. Такође, број итерација ми је само 5, што је мање од оптималног броја којим се достиже жељени ниво халуцинације. Зато ћу, такође, приложити пример који нисам сâм направио, али који лепо показује разлику у сликама која се направила малим изменама параметара.

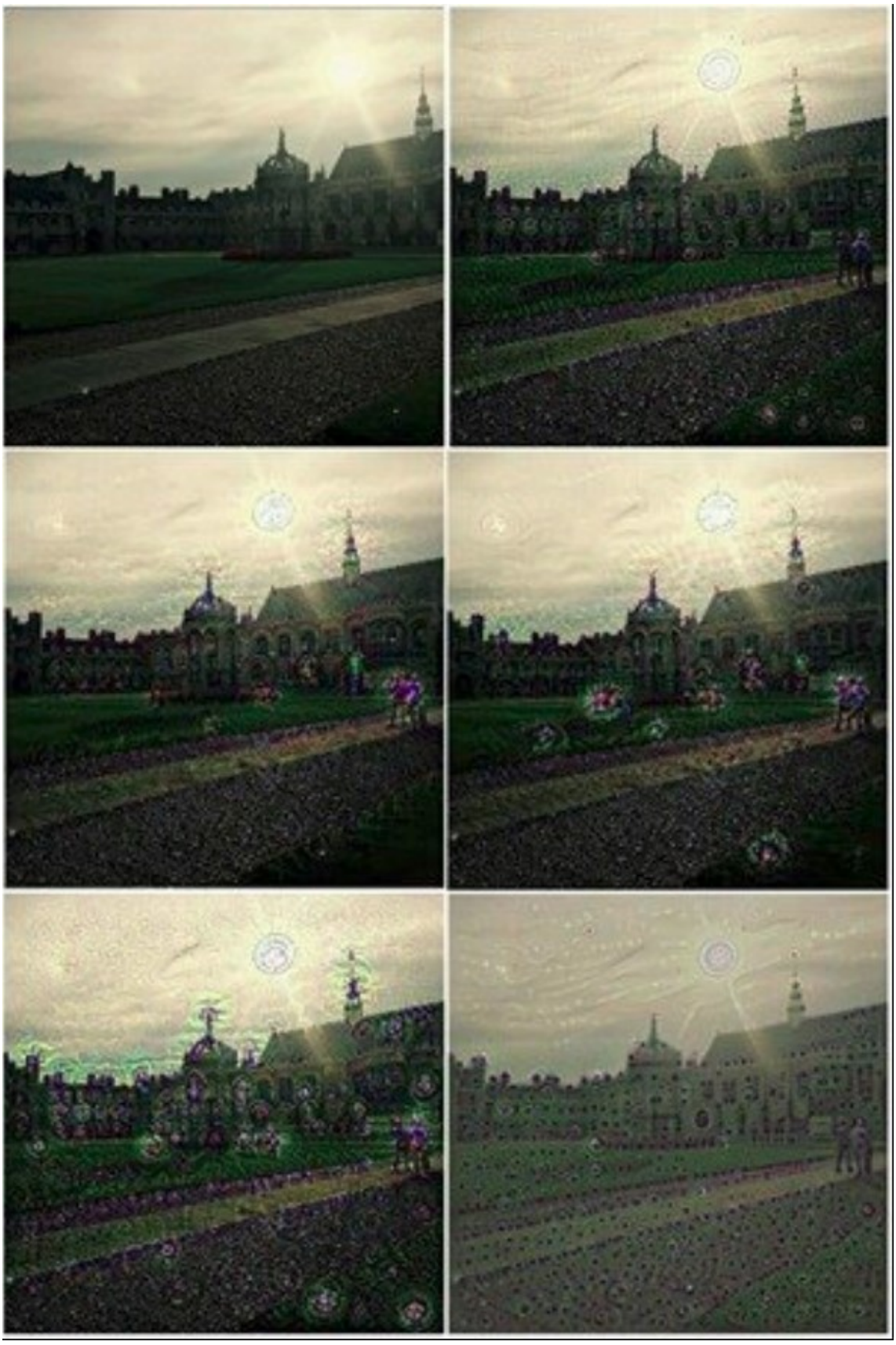
Слика 4.1: Оригинална слика и слика обрађена мојим програмом



Слика 4.2: Оригинална слика и слика обрађена мојим програмом

```
from keras.models import Model
from keras.layers import Input, Convolution2D, MaxPooling2D, Dense, Dropout, Flatten
from keras.layers.normalization import BatchNormalization
from keras.utils import np_utils

inp = Input(shape=(3, 227, 227))
inp_norm = BatchNormalization(axis=1)(inp)
conv_1 = Convolution2D(96, 11, 11, subsample=(4,4), activation='relu', name='conv_1')(inp_norm)
conv_1 = BatchNormalization(axis=1)(conv_1)
pool_1 = MaxPooling2D((3, 3), strides=(2,2))(conv_1)
conv_2 = Convolution2D(256, 5, 5, border_mode='same', activation='relu', name='conv_2')(pool_1)
conv_2 = BatchNormalization(axis=1)(conv_2)
pool_2 = MaxPooling2D((3, 3), strides=(2,2))(conv_2)
conv_3 = Convolution2D(384, 3, 3, border_mode='same', activation='relu', name='conv_3')(pool_2)
conv_3 = BatchNormalization(axis=1)(conv_3)
conv_4 = Convolution2D(384, 3, 3, border_mode='same', activation='relu', name='conv_4')(conv_3)
conv_4 = BatchNormalization(axis=1)(conv_4)
conv_5 = Convolution2D(256, 3, 3, border_mode='same', activation='relu', name='conv_5')(conv_4)
conv_5 = BatchNormalization(axis=1)(conv_5)
pool_3 = MaxPooling2D((3, 3), strides=(2,2))(conv_5)
flat = Flatten()(pool_3)

layer_1 = Dense(4096, activation='relu', name='layer_1')(flat)
layer_1 = BatchNormalization(axis=1)(layer_1)
layer_1 = Dropout(0.5)(layer_1)
layer_2 = Dense(4096, activation='relu', name='layer_2')(layer_1)
layer_2 = BatchNormalization(axis=1)(layer_2)
layer_2 = Dropout(0.5)(layer_2)

out = Dense(1000, activation='softmax', name='layer_3')(layer_2)

model = Model(input=inp, output=out)
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

```

Слика 4.3: Исечак из **Python** програма, код модела VGG16



Слика 4.4: Необрађена слика, као и 5 слика добијене након извршавања **DeepDream**-а са различитим параметрима

5

Закључак

5.1 Резултат

Овај пројекат је био успешан – идеја је успешно имплементирана, и андроид апликација је у потпуности функционална. Покренута је на доста примера, и за сваки је дала резултат у разумном времену и без преоптерећивања меморије. Због недостатка средстава, не производи се резултат какав је можда очекиван (слике нису довољно трипозне као оне које се могу наћи на интернету). Када је покретање извршавано користећи **GPU**(graphics processing unit) постизани су много бољи резултати, за доста краће време извршавања.

5.2 Научено штиво

Поред тога што сам морао темељно да савладам све теоријско знање потребно за писање овог рада, највећи проблем при изради рада је био хардверске природе – оптимизација алгоритма, тако да ради у разумном времену с обзиром на ограничене хардверске ресурсе. Иако је било прилично мучно наћи средину између ефекта трипозности и конзумирања ресурса, мислим да сам на крају успео да пронађем нешто чиме сам задовољан.

Такође, јако ме је заинтересовала функција губитка и праћење како се њеном ситном променом мења целокупни ефекат који се одвија на слици. Захваљујући овом раду, наставићу да се бавим темама уско повезаним са овом у будућности.

На самом крају, желео бих да се захвалим:

- **Петру Величковићу** – ментору овог рада, на изузетној посвећености и помоћи при савладавању потребног штива, као и на времену одвојеном да ми упути корисне и конструктивне критике и коментаре
- **Јелени Хаџи-Пурић** – професорки информатике, на неизмерној подршци током мог читавог школовања у свим аспектима
- **Жељку Лежаји и Мијодрагу Ђуришићу** – професорима информатике, на помоћи у савладавању основа нових знања из информатике
- **Својим родитељима** – на подршци током читавог школовања, а посебно мом оцу који ми је помогао да од малена развијем афинитете према математици и информатици
- **Катарини, Кости, Збирету,** као и свима осталима који су помогли при изради овог рада

Литература

[1] Deep learning, Yann LeCun, Yoshua Bengio and Geoffrey Hinton, Nature, 2015

[2] Learning representations by back-propagating errors, David Rumelhart, Geoffrey Hinton, and Ronald Williams, Nature, 1986

[3] Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Karen Simonyan, Andrea Vedaldi, Andrew Zisserman, 2013

[4] A Beginner's Guide To Understanding Convolutional Neural Networks, Adit Deshpande, 2016

[5] ImageNet Classification with Deep Convolutional Neural Networks, Alex Krizhevsky, Ilya Sutskever, Geoffey Hinton

[6] Convolutional Neural Networks for Visual Recognition, Stanford CS class CS231n, 2017

[7] Understanding and Visualizing Convolutional Neural Networks , Stanford CS class CS231n, 2017

[8] Neural Networks Part 1: Setting up the Architecture, Stanford CS class CS231n, 2017

[9] Neural Networks Part 2: Setting up the Data and the Loss, Stanford CS class CS231n, 2017

[10] Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation, Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, Rob Fergus

[11] Convolutional Kernel Networks, Julien Mairal, Piotr Koniusz, Zaid Harchaoui, Cordelia Schmid

[12] Do Deep Nets Really Need to be Deep?, Jimmy Ba, Rich Caruana