Универзитет у Београду

Електротехнички факултет



Машинско препознавање карата

у оквиру стоне игре картама

Завршни рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| доц. др Марко Мишић | Александар Динчић 2018/0028 |

Београд, септембар 2022.

Садржај

[Садржај 2](#_Toc115305068)

[1. Увод 3](#_Toc115305069)

[2. Опис проблема 5](#_Toc115305070)

[2.1. О машинском учењу 5](#_Toc115305071)

[2.2. О неуралним мрежама 7](#_Toc115305072)

[2.3. О конволуционим неуралним мрежама 9](#_Toc115305073)

[2.4. Преглед изабране игре картама 12](#_Toc115305074)

[2.5. Анализа проблема 13](#_Toc115305075)

[3. Коришћени алати 15](#_Toc115305076)

[3.1. TensorFlow и Keras 15](#_Toc115305077)

[3.2. TensorFlow Object Detection API (TFOD API) 15](#_Toc115305078)

[3.3. Albumentations 15](#_Toc115305079)

[3.4. LabelImg 16](#_Toc115305080)

[3.5. Android Studio 17](#_Toc115305081)

[4. Проблем детекције карата 18](#_Toc115305082)

[4.1. Принципи рада неуралних мрежа за детекцију објеката 18](#_Toc115305083)

[4.2. Имплементација 20](#_Toc115305084)

[4.2.1. Прикупљање података 20](#_Toc115305085)

[4.2.2. Избор архитектуре 21](#_Toc115305086)

[4.2.3. Тренирање 22](#_Toc115305087)

[4.2.4. Интеграција у мобилну апликацију 25](#_Toc115305088)

[4.3. Евалуација решења 28](#_Toc115305089)

[5. Проблем идентификације карата 30](#_Toc115305090)

[5.1. Прикупљање података 30](#_Toc115305091)

[5.2. Принципи класификационих алгоритама машинског учења 30](#_Toc115305092)

[5.3. О сијамским мрежама 32](#_Toc115305093)

[5.4. Имплементација решења 34](#_Toc115305094)

[5.4.1. Припрема скупа података 34](#_Toc115305095)

[5.4.2. Изградњња архитектуре и тренирање 37](#_Toc115305096)

[5.4.3. Интеграција у мобилну апликацију 40](#_Toc115305097)

[5.5. Евалуација решења 41](#_Toc115305098)

[6. Закључак 43](#_Toc115305099)

[Литература 44](#_Toc115305100)

1. Увод

Компјутерска визија је област вештачке интелигенције која се бави изградњом система способних за обраду и разумевање визуелних података, попут слика и видео снимака. Примери проблема из домена компјутерске визије јесу детекција објеката, препознавање лица, и детекција покрета. Иако постоје алгоритми за решавање ових и сродних проблема, који се могу употребити у специфичним околностима, чест приступ решавању проблема у општим условима јесте коришћење техника машинског учења. Овај рад бави се применом машинског учења на специфичан проблем из домена компјутерске визије. Циљ је имплементација система за препознавање карата у изабраној стоној игри картама, намењеног за употребу на мобилним уређајима. Као игра картама за коју ће се решавати проблем изабрана је игра *Yu-Gi-Oh!*, због великог броја различитих карата у игри, као и због препознатљивих и разноликих визуелних одлика карата. Биће размотрене архитектуре коришћене за решавање сродних проблема, и донет закључак о њиховој применљивости на наш проблем и скуп података којим располажемо.

Рад је сачињен од шест поглавља. Уводно поглавље сачињаваће излагање о компјутерској визији, области којој припада наш проблем, опис самог проблема, као и кратко представљање сваког од поглавља рада.

Друго поглавље посвећено је теоријским основама техника машинског учења, са освртом на специфичне технике коришћене при решавању проблема компјутерске визије. Затим ћемо, у контексту коришћења ових техника, анализирати проблем који решавамо, и рашчланити га на целине.

У трећем поглављу биће укратко представљени значајни алати и библиотеке који су коришћени за израду решења.

Наредна два поглавља биће посвећена по једној од двеју целина дефинисаних у другом поглављу. Четврто поглавље бавиће се проблемом проналажења карата на слици, док ће се пето поглавље бавити проблемом идентификације карата добијених као резултат проналажења. Оба поглавља биће структурирана на сличан начин. Најпре ће бити представљени принципи често коришћених решења за дати потпроблем, након чега ћемо изабрати приступ који највише одговара нашим околностима. Након тога, биће изнети детаљи изградње модела, укључујући процес прикупљања података, имплементацију архитектуре, тренирање и интеграцију у мобилну апликацију. Поглавља завршавамо евалуацијом добијених модела, као и разматрањем могућности за њихово побољшање.

На крају, резимираћемо рад и изнети закључак о применљивости коришћених метода за решавање нашег и сличних проблема, на основу добијених резултата.

1. Опис проблема

У овом поглављу биће постављена теоријска подлога за технике машинског учења, као и генерални приступи решавању визуелних проблема у виду конволуционих неуралних мрежа. Затим ће бити дефинисан проблем из домена компјутерске визије који желимо да решимо, као и могући приступи решавању проблема коришћењем машинског учења.

* 1. О машинском учењу

Машинско учење представља област вештачке интелигенције која се бави изградњом алгоритама чији се рад ослања на скуп података релевантних за проблем који се решава. Изградњом статистичког модела базираног на датим подацима, алгоритам може да реши проблем, без експлицитног програмирања како да то чини. Овај приступ се примењује на задатаке за које је тешко дефинисати алгоритамски приступ извршавању.

Машинско учење се дели на надгледано (*supervised learning*), ненадгледано (*unsupervised learning*), полунадгледано (*semi-supervised learning*) и учење поткрепљивањем (*reinforced learning*). Пошто су за овај рад важне прве две категорије, оне ће бити објашњене.

Код надгледаног учења, скуп података дат је у виду парова – улазна вредност, и очекивана излазна вредност. Циљ надгледаног учења је створити модел који би, на основу скупа података, умео да одреди излаз за било коју улазну вредност. Проблеми који се на овај начин решавају јесу класификација и регресија. Класификација је проблем одређивања вредности (класе) из коначног скупа, за дати улазни податак. Регресија је проблем одређивања континуалне вредности за дати улазни податак.

Код ненадгледаног учења, скуп података дат је само у виду улазних вредности, без других ознака. Циљ је створити модел који би улазну вредност трансформисао у неку другу вредност, која би се користила за решавање неког практичног проблема. Примери проблема који се решавају ненагледаним учењем јесу кластеризација података, и проналажење аномалија.

Генерално, градивни блокови алгоритама машинског учења јесу следеће три компоненте:

* Фунцкија губитка
* Оптимизациони критеријум заснован на функцији губитка
* Оптимизациони алгоритам за решавање оптимизационог критеријума на основу датог скупа података

Ова три појма биће објашњени на примеру линеарне регресије. Линеарна регресија је регресивни алгоритам надгледаног учења. За улазни податак , дат у виду вектора својстава димензије *D*, циљ алгоритма је изградња модела који је линеарна комбинација својстава улазног податка

(2.1.1.)

где је вектор параметара димензије *D*, а реална константа.

У општем случају, функција губитка представља меру корисности изгубљену због погрешног предвиђања модела за неку улазну вредност. У случају линеарне регресије, за функцију губитка користи се функција квадрата грешке

(2.1.2.)

где је очекивана излазна вредност за улазну вредност .

Сви алгоритми у машинском учењу заснивају се на експлицитној или имплицитној оптимизацији неког критерујма. Уобичајено се ради о минимизацији или макзимизацији вредности неке оптимизационе функције. У случају линеарне регресије, циљ је пронаћи вредности и за које је дати израз минималан

(2.1.3.)

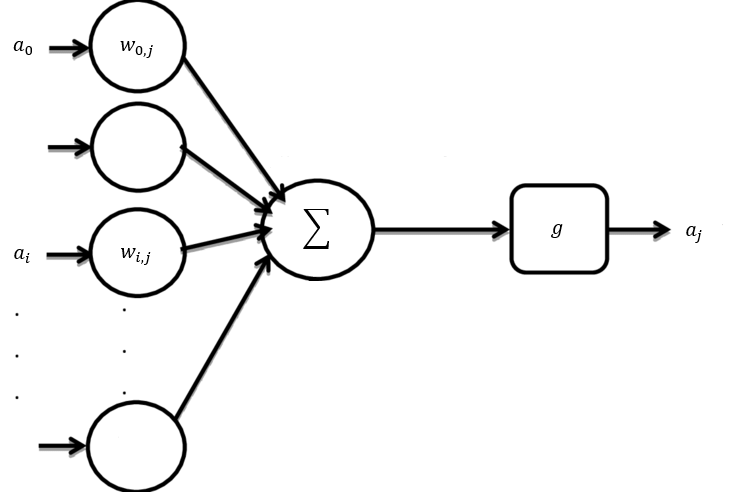
где је величина скупа података.

Задатак оптимизационог алгоритма је да, на основу коришћеног скупа података, пронађе решење оптимизационог критеријума. За диференцијабилне оптимизационе функције, најчешће коришћени оптимизациони алгоритми су градијентни спуст и стохастички градијентни спуст. Градијентни спуст је итеративан алгоритам који, почевши од насумичнх вредности променљивих оптимизационе фунцкије, у сваком кораку ажурира вредност сваке променљиве сразмерно парцијалном изводу функције по тој променљивој, у тачки у којој се алгоритам тренутно налази. На овај начин проналази се локални минимум фунцкије. Градијентни спуст се управо користи и код линеарне регресије.

Решење оптимизационог проблема алгоритма назива се статистички модел, или само модел. Процес налажења модела за дати скуп података назива се тренирање. Један пролаз алгоритма кроз цео скуп података за тренирање назива се епоха. [1]

* 1. О неуралним мрежама

Неуралне мреже су скуп алгоритама у машинском учењу чија је структура заснована на начину повезивања можданих ћелија. Састоје се из јединица (неурона) повезаних усмереним везама. Веза од неурона *i* до неурона *j* служи да се излаз (активација) неурона *i* у ознаци доведе на улаз неурона *j*. Свака веза има нумеричку тежину која одређује јачину и знак те везе. Сваки неурон такође поседује константан фактор на улазу, који се може моделовати улазном везом вредности једнаке 1, са тежином . Неурон се „активира“ када линеарна комбинација његових улаза пређе неки оштар или меки праг.



Слика 2.2.1. Визуелни приказ неурона [2]

Излаз неурона рачуна се применом активационе функције на пондерисани збир његових улаза

(2.2.1.)

где је активациона функција. То је обично нелинеарна функција, ручно изабрана пре почетка тренирања. Чест избор за активациону функцију јесу ReLU функција

(2.2.2.)

као и сигмоид функција

(2.2.3.)

Значај активационих функција је у увођењу нелинеарности у неуралну мрежу. Без њих, неурална мрежа би представљала линеарну функцију без обзира на димензије мреже, из разлога што је линеарна фунцкија линеарне функције и даље линеарна функција.

Неурони се везама повезују у мрежу. У зависности од начина повезивања, разликујемо мреже са простирањем унапред и рекурентне мреже. Мреже са простирањем унапред могу се представити усмереним ацикличним графом, односно, не постоје петље у везивању неурона. Код рекурентних мрежа, петље су дозвољене. Због зависности одзива мреже и од претходног улаза, може се рећи да рекуретне мреже имају краткорочну меморију. У овом раду користиће се мреже са простирањем унапред.

Неурони у оквиру мреже су уобичајено распоређени у слојеве. Код мрежа са простирањем унапред, неурони примају улаз само од неурона из претходних слојева. Уколико на улаз сваког неурона у слоју долазе сви неурони из претходног слоја, ради се о потпуно повезаном слоју. Број слојева у мрежи, као и величина сваког појединачног слоја, су прозивољне вредности и њихов избор је одлука коју треба донети при пројектовању. [3]

Diagram

Description automatically generated

Слика 2.2.2. Пример неуралне мреже [4]

Слојеви мреже који се налазе између улазног и излазног слоја називају се скривени слојеви. Мрежа на слици 2.2.2. садржи 2 скривена слоја. Пораст димензија мреже води и до пораста параметара које треба оптимизовати, што чини време тренирања дужим а резултујући модел комплекснијим. Као и остали алгоритми учења, неуралне мреже се за изградњу модела ослањају на принципе из потпоглавља 2.1. За оптимизациони алгоритам најчешће се користе варијанте стохастичког градијентног спуста, као што је Adam. Ажурирање параметара скривених слојева при тренирању могуће је захваљујући принципу повратног простирања. Овај принцип ослања се на правило рачунања извода сложене функције. За сложену функцију извод се рачуна према формули

(2.2.4.)

Осим тога, излаз неуралне мреже може се приказати као низ угњеждених функција, где свака функција одговара једном слоју. Захваљујућу овоме, повратно простирање рачуна парцијалне изводе функције кренувши од последњег слоја ка првом, користећи резултате из претходно посећених слојева за рачунање парцијалног извода у тренутном. Ажурирање параметара на основу добијених парцијалних извода ради се као и раније. [1]

* 1. О конволуционим неуралним мрежама

Број параметара у неуралној мрежи убрзано расте са порастом њених димензија. Из овог разлога, слике се лоше скалирају са традиционалним неуралним мрежама. Као улазни податак, слике су обично представљене као тродимензионалне (висина, ширина, број канала боја), где прве две димензије често умеју да буду троцифрених величина. Због тога, ако се користе потпуно повезани слојеви, већ у првом слоју после улазног слоја неурони имају огроман број параметара. Превелики број параметара у мрежи може негативно да утиче не само на време тренирања, већ и на перформансе модела. Стога се, као решење за проблеме где су улазни подаци слике, користе конволуционе неуралне мреже.

Као и традиционалне неуралне мреже, конволуционе неуралне мреже су изграђене од неурона распоређених у слојеве. Разлика је у начину повезивања неурона са улазним подацима, где излаз из сваког слоја задржава тродимензионалност улазног податка. Конволуционе неуралне мреже најчешће чине следећи слојеви:

* Конволуциони слојеви
* Pooling слојеви
* Потпуно повезани слојеви

Конволуциони слојеви састоје се од скупа тродимензионалних филтара. Број филтара, као и њихова висина и ширина, одлучује се приликом пројектовања, док је дубина одређена дубином улазних података. При тренирању, параметри за које се траже оптималне вредности су елементи сваког од филтара, уз још по један, константан параметар уз сваки филтар. Приликом одређивања излазне вредности, филтар клизи пролази кроз улазни податак по висини и ширини – слева надесно, одозго надоле. У свакој позицији, рачуна се векторски производ елемената улаза на тренутној позицији и елемената филтра у тренутној позицији, додаје се константна вредност, и одводи на излаз. На овај начин, на излазу филтра добија се дводимензионална матрица величине *N* – *F* + 1, где је *N* димензија улаза, а *F* димензија филтра.

A diagram of a house

Description automatically generated with low confidence

Слика 2.3.1. Пример рада филтра у конволуционом слоју [1]

Излази свих филтара једног слоја представљају излаз тог слоја. Дакле, излаз конволуционог слоја је тродимензионалан, ширине и висине једнаке ширини и висини излаза филтара, и дубине једнакој броју филтара.

Циљ филтара је да, у малом региону улаза одређеним величином и тренутном позицијом, детектују визуелне шаблоне, као што су ивица објекта или појава боје. На основу открића претходних слојева, каснији конволуциони слојеви откривају својства више комплексности. Број параметара у конволуционом слоју са *N* филтара величине (*a*,*b*,*c*) јесте

(2.3.1.)

Узевши у обзир да се за димензије филтара обично узимају мали бројеви, број параметара у конволуционом слоју је драстично мањи од броја параметара у поптуно повезаном слоју.

Конволуциони слојеви могу бити параметризовани прескоком (*stride*) – величином помераја филтара при пролазу кроз улазни податак. На тај начин смањује се величина излаза. Још један чест параметар јесте попуњавање (*padding*) – ширина обода којим се улазни податак проширује пре проласка филтара. Обод може бити попуњен нулама, или пресликавањем улазног податка као у огледалу.

Сврха Pooling слојева јесте смањење димензија података при протоку кроз мрежу. Слично конволуционим слојевима, одликује их величина филтра, као и прескок. Међутим, разликују се по томе што не поседују параметре, већ филтри примењују неку једноставну функцију над вредностима са улаза. Најчешће коришћена функција је проналажење максимума у тренутном региону. Филтар примењује функцију на сваки ниво дубине улазног податка. Тако за улазни податак величине (*a*,*b*,*c*), Pooling слој величине *N* и прескока *S* даје излаз величине

(2.3.2.)

Table

Description automatically generated

Слика 2.3.2. Пример рада Max pooling слоја [5]

Најчешће се користи Max pooling слој величине 2 и прескока 2, који доводи до смањења величине података од 75%. [5]

Поптуно повезани слојеви су већ обрађени у потпоглављу 2.2. У конволуционим неуралним мрежама појављују се као последњи слојеви, како би се излаз мреже прилагодио проблему који се решава. На пример, код проблема класификације слика, потребно је доделити једну од *N* класа слици на улазу. У том случају се као последњи слој мреже може искористити потпуно повезани слој од *N* неурона, где је излаз сваког неурона вероватноћа да слика припада датој класи.

* 1. Преглед изабране игре картама

Као игра за коју решавамо проблем препознавања карата изабран је *Yu-Gi-Oh!*. У тренутку израде овог рада, у овој игри постоји 12122 различитих карата.

A picture containing text

Description automatically generated

Слика 2.4.1. Пример карата из игре *Yu-Gi-Oh!* [6]

Заједнички визуелни елементи на свакој од карата у овој игри јесу:

* Име карте на врху
* Илустрација у средини
* Текстуална област са подацима о карти на дну
* Јединствени идентификатор у доњем левом ћошку

Карта се јединствено може одредити помоћу имена, илустрације, или идентификатора. Одлучили смо да при препознавању карте посматрамо само њену илустрацију. Не желимо се ослањати на текстуалне податке на карти, како би препознавање могло да ради независно од језика на којем је карта написана. Наш скуп података поседује само слике карата на енглеском. Са друге стране, идентификатор је превише малих димензија на карти, те врло лако може бити нејасан на фотографији. Такође, пошто се налази у ћошку карте, склон је оштећењу, те лако може постати неупотребљив за наше потребе. Стога, илустрација је изабрана као најповерљивији извор информација о карти.

* 1. Анализа проблема

Улазни податак нашег система треба да буде слика, док излазни податак представљају информације о карти/картама на слици. Овај проблем се може раставити на два потпроблема

* Детекција карата на слици
* Класификација детектованих карата

Један од приступа решавања првог потпроблема јесте оставити у одговорност кориснику да лоцира карту. То се може учинити приказивањем правоугаоног региона преко приказа камере за време фотографисања карата. Од корисника би се очекивало да у овај правоугаоник постави карту. Један од недостатака оваквог приступа јесте ограничавање детекције на само једну карту по слици. Други начин за решавање овог проблема јесте изградња неуралне мреже која решава проблем детекције објеката на слици. Она би, за унету слику на улазу, као излаз дала координате темена граничних правоугаоника који окружују карте на слици. На основу добијених граничних правоугаоника, са слике можемо исећи области са картама и користити их као улаз у следећи потпроблем. Овај приступ лишава корисника одговорности лоцирања карата, и омогућава проналажење већег броја карата на слици. Одлучили смо се за овај начин детекције.

Други потпроблем биће решен изградњом неуралне мреже за препознавање карте на основу унете слике илустрације карте. Биће размотрена два погледа на овај проблем. Један од њих јесте посматрати га као проблем класификације слика. У том случају, неурална мрежа би на излазу имала по једну вредност за сваку класу, која би представљала вероватноћу да слика припада тој класи. Пошто је свака карта одвојена класа, у овом случају постојало би укупно 12122 класа. Класа са највећом вероватноћом представљаће излаз система. Са друге стране, проблем се може посматрати и као одређивање сличности двeју слика. Излаз мреже би онда била нумеричка вредност, или вектор вредности. Сличне слике имаће сличне излазе из мреже. Карта која, по изабраној метрици, има најближу излазну вредност слици коју смо унели представљаће излаз система. Оба ова приступа биће размотрена у наставку рада.

Пошто су циљна платформа мобилни уређаји, преносивост и брз одзив система су приоритет. За потребе тестирања биће развијена прототипска апликација за уређаје са оперативним системом *Android*. Она ће омогућити препознавање карата са фотографије добијене камером мобилног уређаја у реалном времену. Тестирање над постојећим сликама из фајл система биће урађено на рачунару.

1. Коришћени алати

У овом поглављу биће дат кратак приказ свих алата и библиотека коришћених за имплементацију решења. За изградњу и тренирање свих модела коришћен је програмски језик *Python*, док је прототипска апликација имплементирана у програмском језику *Java*.

* 1. TensorFlow и Keras

*TensorFlow* је софтверска библиотека намењена за машинско учење и вештачку интелигенцију. Пружа више нивоа апстракције за обављање радњи у вези прикупљања и процесирања података, изградње, тренирања, евалуације и покретања модела, као и њихове интеграције у веб и мобилне апликације. Доступна је за већи број програмских језика, укључујући *Python*, *C++*, *Java*, и *JavaScript*. *Keras* је библиотека која пружа API високог нивоа за рад са неуралним мрежама. Користи *TensorFlow* као подлогу.

A picture containing text, clipart, sign

Description automatically generated

Слика 3.1.1. Логои библиотека TensorFlow и Keras [7][8]

* 1. TensorFlow Object Detection API (TFOD API)

*TensorFlow* нуди бесплатан API отвореног кода, као надоградњу на саму библиотеку, намењен раду са проблемима детекције објеката. Пружа интерфејс за једноставно прављење сопствених архитектура, као и коришћење постојећих познатих архитектура, доступних имплементиране за овај API, и њихову модификацију за сопствене потребе помоћу конфигурационих фајлова.

* 1. Albumentations

*Albumentations* је библиотека за програмски језик *Python*, која пружа функционалности за аугментацију података (*data augmentation*) код скупова података са сликама. Ова техника се користи за повећање величине скупа података, генерисањем варијација постојећих података у скупу. У овом случају, библиотека нуди велики број трансформација над сликама, укључујући просторне трансформације и трансформације над појединачним пикселима. Могућа је интеграција са библиотекама за машинско учење, попут библиотеке *TensorFlow*, што би омогућило генерисање трансформација у меморији, приликом тренирања. Због проблема некомпатибилних верзија, ова могућност није коришћена, већ смо скуп података проширили пре тренирања, и сачували га на диску.



Слика 3.3.1. Пример трансформација над картом

* 1. LabelImg

*LabelImg* је бесплатан алат написан у језику *Python*, намењен за анотацију слика. Кориснику се омогућава означавање граничних правоугаоника око објеката, као и класа којима припадају, путем графичког интерфејса. Анотације се потом чувају као засебне датотеке, у једном од више понуђених формата коришћених од стране познатих сликовних скупова података и модела, као што су *Pascal VOC* или *YOLO*.

Text

Description automatically generated with low confidence

Слика 3.4.1. Приказ алата LabelImg

* 1. Android Studio

*Android Studio* је званично интегрисано развојно окружење за израду апликација намењених *Android* уређајима. Као основу користи *IntelliJ*, окружење за развој апликација у језицима за *JVM*, као што су *Java* и *Kotlin.* Окружење пружа олакшано дизајнирање корисничког интерфејса графичким или програмским путем, проверу перформанси, безбедности и приступачности апликације, као и тестирање апликације на свим конфигурацијама и типовима уређаја захваљујући уграђеном емулатору *Android* система.

A screenshot of a computer

Description automatically generated with medium confidence

Слика 3.5.1. Приказ окружења Android Studio [9]

1. Проблем детекције карата

У овом поглављу биће разматрано решавање проблема детекције карата на датој слици. Најпре ће бити успостављени теоријски принципи рада неуралних мрежа за решавање овог проблема, са освртом на две врсте мрежа и предности њиховог коришћења. Затим ће бити приказан детаљан поступак изградње модела за изабрану архитектуру. На крају ће бити приказани резултати евалуације добијеног модела, као и области у којима се модел може побољшати.

* 1. Принципи рада неуралних мрежа за детекцију објеката

Проналажење координата граничног правоугаоника око траженог објекта представља регресивни проблем. Најчешће коришћене функције губитка код овог проблема јесу варијанте фунцкије *Intersect over Union* (*IoU*). За гранични правоугаоник *A*, који представља предикцију модела, и тачан гранични правоугаоник *B*, који је дат у скупу података за тренирање, ова фунцкија рачуна количник пресека површина *A* и *B* са унијом површина истих.

Diagram

Description automatically generated with low confidence

Слика 4.1.1. IoU функција [10]

Како се, са повећањем степена преклапања два правоугаоника, пресек повећава, а унија смањује, циљ је максимизовати вредност ове функције. [11]

Проналажење класе којој припада објекат представља класификациони проблем. Принципи рада неуралних мрежа које решавају овај проблем биће представљени у поглављу 4.

Поступак детекције објеката се, код решења базираних на техникама машинског учења, најчешће може рашчланити на два корака:

1. Прављење генерализованих предикција постојања објеката на слици
2. Класификација објеката из претходног корака, као и регресија њихових граничних правоугаоника

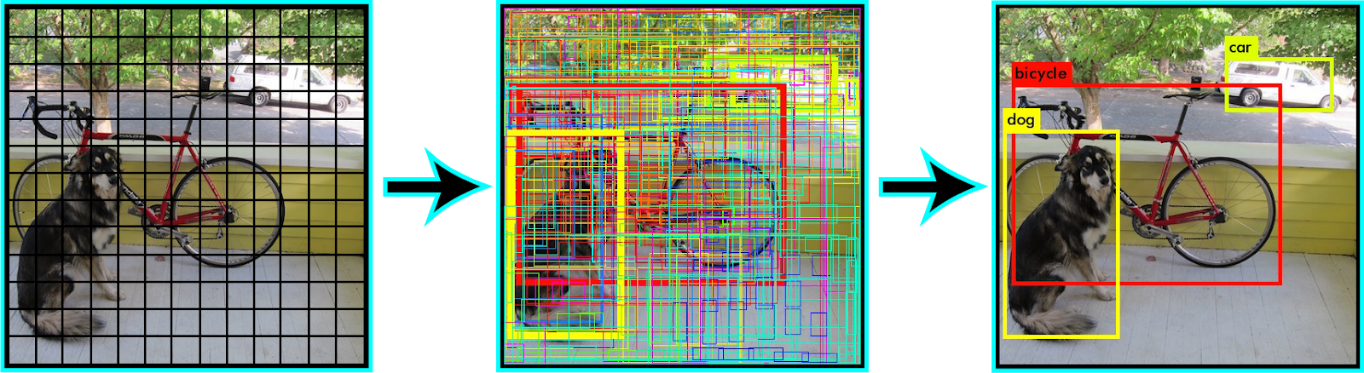
У зависности од начина имплементације ова два корака, разликујемо детекцију објеката у два корака (two-stage object detection), и детекцију објеката у једном кораку (one-stage object detection). Код детекције објеката у два корака, систем можемо раздвојити на две целине, од којих је свака задужена за један од поменутих корака. Сврха прве целине јесте издвајање региона од интереса са слике, који представљају кандидате за детектоване објекте, не узимајући у обзир класу самог објекта. Ово се може извести алгоритмима из области компјутерске визије као што је селективна претрага, или коришћењем засебне неуралне мреже. Добијени региони прослеђују се другој целини, која врши класификацију објекта обухваћеног регионом, као и регресију његовог граничног правоугаоника. Ова целина се имплементира коришћењем једне или више неуралних мрежа. Због веће комплексности имплементације, детектори објеката у два корака су углавном спорији од детектора у једном кораку, али постижу већу тачност. Пример архитектура за детекцију објеката у два корака јесу *Region-Based Convolutional Network* (*R-CNN*), и његове надоградње. [12]

Diagram

Description automatically generated

Слика 4.1.2. Приказ рада R-CNN [13]

Са друге стране, код детекције објеката у једном кораку, не постоји одвојена целина система за проналажење региона од интереса. Уместо тога, класификација и регресија раде се целовито. Улазна слика дели се у мрежу, где свака ћелија добијене мреже представља генерализовану предикцију. Ово се може постићи конволуционом мрежом, чији би излаз биле мапе својстава исте ширине и висине као и мрежа која дели слику. Комбиновањем елемената мапа својстава, за сваку појединачну ћелију мреже могу се одредити релевантне нумеричке вредности попут вероватноће постојања објекта у ћелији, вероватноће припадности тог објеката свакој од класа, као и координате граничног правоугаоника. Пошто на овај начин могу настати редундантне детекције, треба изабрати рачин за њихово отклањање. Одбацивање предикција са ниском вероватноћом, као и задржавање највероватније предикције међу оним са великим преклапањем, може бити један од решења. Иако мање тачности у поређењу са детекцијом у два корака, архитектуре за детекцију објеката у једном кораку су једноставније и брже, што их чини повољним за коришћење у апликацијама у реалном времену. Пример архитектура за детекцију објеката у једном кораку јесу *Single Shot Multibox Detector* (*SSD*), као и *You Only Look Once* (*YOLO*).[14]



Слика 4.1.3. Приказ рада YOLO[10]

* 1. Имплементација

Ово потпоглавље намењено је опису имплементације модела за детекцију карата. Најпре ће бити описан поступак креирања скупова података за тренирање и тестирање, након чега ће бити приказан начин на који су искоришћене специјализоване библиотеке, намењене проблемима детекције објеката, у изради нашег решења.

* + 1. Прикупљање података

Пошто нисмо пронашли постојећи скуп података за детекцију *Yu-Gi-Oh!* карата на Интернету, морали смо прибећи креирању сопственог. За ову сврху написали смо *Python* скрипту, која прикупља слике карата са популарних веб сајтова за куповину и продају *eBay* и *KupujemProdajem*. Из скупа прикупљених слика, изабрано је 350, које су потом ручно анотиране коришћењем алата *LabelImg*. Анотације су чуване у *Pascal VOC* формату. То је формат чувања анотација у виду *XML* фајлова који, поред општих података о називу и путањи фајла са сликом и димензијама слике, за сваки гранични правоугаоник чувају његову класу, као и два темена: горње лево и доње десно, које једнозначно одређују координате правогаоника.

<annotation>

<folder>**img\_scraped\_labeled\_sq**</folder>

<filename>**img\_1001.jpg**</filename>

<path>**D:\Faks\diplomski\jugio\data\img\_scraped\_labeled\_sq\img\_1001.jpg**</path>

<source>

<database>**Unknown**</database>

</source>

<size>

<width>**450**</width>

<height>**600**</height>

<depth>**3**</depth>

</size>

<segmented>**0**</segmented>

<object>

<name>**card**</name>

<pose>**Unspecified**</pose>

<truncated>**0**</truncated>

<difficult>**0**</difficult>

<bndbox>

<xmin>**126**</xmin>

<ymin>**161**</ymin>

<xmax>**301**</xmax>

<ymax>**336**</ymax>

</bndbox>

</object>

</annotation>

Код 4.2.1.1. Нотација генерисана на слици 3.4.1.

* + 1. Избор архитектуре

Анотирање великог броја слика за коришћење у скупу података за тренирање детектора објеката је временски захтеван посао. Истовремено, тренутна количина од 350 анотираних слика је недовољна да бисмо од нуле изградили модел који би пружио задовољавајуће перформансе. Како бисмо што брже добили прихватљиве резултате, искористићемо технику учења трансфером (transfer learning). Учење трансфером подразумева коришћење постојећег модела, намењеног решавању једног проблема, као подлогу за решавање другог проблема.

*TensorFlow* нуди колекцију модела за детекцију објеката намењених за коришћење са *TensorFlow Object Detection API,* тренираних над скупом података *COCO 2017*. Овај скуп података садржи 330 хиљада слика са 1,5 милиона означених објеката из 80 категорија. За све моделе у понуди, доступан је табеларан приказ брзине одзива у милисекундама, као и тачност над *COCO* скупом података у процентима. Ове две вредности су обрнуто пропорционалне. Пошто је модел намењен за коришћење на мобилним уређајима, брзина одзива је изабрана за значајнију метрику. Определили смо се за коришћење модела *SSD MobileNet V2 FPNLite 320x320* као основу за изградњу нашег модела. *SSD, MobileNet V2,* и *FPNLite* су називи мрежа које својим везивањем сачињавају архитектуру овог модела, док се *320x320* односи на величину слика које се прихватају на улазу.

* + 1. Тренирање

Тренирање, евалуација, и извоз модела се могу извршити позивањем *Python* скрипти које нуди *TFOD API*, уз прослеђивање одговарајућих аргумената. Пре него што је то могуће учинити, неопходно је изгенерисати следеће фајлове:

* Листа могућих класа објеката
* Анотације слика, сакупљених у један фајл
* Конфигурација модела

Сви ови фајлови сачувани су у виду *Protocol Buffers* порука. *Protocol Buffers* је формат порука намењен за преношење и складиштење структурираних података. Овај формат прописује језик за дефинисање структуре порука, у виду *.proto* фајлова. Коришћењем бесплатног званичног алата за генерисање кода, из датог *.proto* фајла са дефиницијом структуре порука може се добити кôд на изабраном подржаном језику, који садржи класе за генерисање, серијализацију и десеријализацију порука у том формату. Поруке се могу чувати у бинарном или текстуалном формату. Формат самих порука сличан је формату *JSON*.



Слика 4.2.3.1. Дијаграм рада *Protocol Buffers (слику ћу превести на српски накнадно)* [15]

Фајл који представља листу могућих класа објеката треба да буде у текстуалном *Protocol Buffers* формату, где је свака класа престављена као *item* објекат, са пољем са именом класе *name*, и са јединственим целобројним идентификатором *id*. У нашем случају, постоји само једна класа – класа карте.

item {

name:'card'

id:1

}

Код 4.2.3.1. Приказ генерисаног фајла са листом класа објеката

Фајл са анотацијама слика за цео скуп података написан је у *TFRecord* формату. Овај формат представља речник, чији су кључеви типа *string,* док су вредности *Protocol Buffers* поруке, чији су могући типови дефинисани у класи *tensorflow.train.Feature*, као и у истоименом *.proto* фајлу. За попуњавање овог фајла подацима које захтева *TFOD API*, искоришћена је постојећа *Python* скрипта, која као аргументе захтева локацију слика, локацију анотација, и локацију претходно изгенерисаног фајла са листом класа, а као излаз даје генерисан *TFRecord* фајл. Од 350 анотираних слика, одлучили смо да 335 сачињавају скуп података за тренирање, док преосталих 35 сачињавају скуп података за тестирање. То такође значи да смо у овом кораку изгенерисали два *TFRecord* фајла. Скуп података за тестирање се неће користити за тренирање, већ за евалуацију перформанси модела по завршеном тренирању.

Пошто желимо да користимо постојећи модел за детекцију објеката, користићемо његов конфигурациони фајл као почетну тачку, и модификовати га према нашим потребама. *TFOD API* пружа апстракције за рад са конфигурационим фајловима, као и одговарајуће *.proto* фајлове са очекиванам структурама сегмената конфигурациног фајла.

**import** os

**import** tensorflow **as** tf

**from** object\_detection**.**utils **import** config\_util

**from** object\_detection**.**protos **import** pipeline\_pb2

**from** google**.**protobuf **import** text\_format

#Ucitava konfiguraciju iz konfiguracionog fajla postojeceg modela

config **=** config\_util**.**get\_configs\_from\_pipeline\_file**(**files**[**'pipelineConfig'**])**

#Generisanje konfiguracije za treniranje, i prepisivanje svih polja iz #postojeceg modela

pipeline\_config **=** pipeline\_pb2**.**TrainEvalPipelineConfig**()**

**with** tf**.**io**.**gfile**.**GFile**(**files**[**'pipelineConfig'**],** "r"**)** **as** f**:**

proto\_str **=** f**.**read**()**

text\_format**.**Merge**(**proto\_str**,** pipeline\_config**)**

#Broj klasa

pipeline\_config**.**model**.**ssd**.**num\_classes **=** 1

#Velicina serija u kojim ce se skup podataka ucitavati

pipeline\_config**.**train\_config**.**batch\_size **=** 4

#Putanja do fajla sa parametrima modela, koji ce se koristiti kao pocetni

pipeline\_config**.**train\_config**.**fine\_tune\_checkpoint **=** os**.**path**.**join**(**paths**[**'pretrainedModelPath'**],** pretrainedModelName**,** 'checkpoint'**,** 'ckpt-0'**)**

#Jedna od tri moguce vrednosti za ovu opciju, koristi se kada postojeci model #treniramo nad novim skupom podataka

pipeline\_config**.**train\_config**.**fine\_tune\_checkpoint\_type **=** "detection"

#Podesavanje trening skupa

#Putanja do fajla sa listom klasa

pipeline\_config**.**train\_input\_reader**.**label\_map\_path**=** files**[**'labelMap'**]**

#Putanja do TFRecord fajla sa podacima o skupu podataka

pipeline\_config**.**train\_input\_reader**.**tf\_record\_input\_reader**.**input\_path**[:]** **=** **[**os**.**path**.**join**(**paths**[**'annotationPath'**],** 'train.record'**)]**

#Podesavanje test skupa

#Putanja do fajla sa listom klasa

pipeline\_config**.**eval\_input\_reader**[**0**].**label\_map\_path **=** files**[**'labelMap'**]**

#Putanja do TFRecord fajla sa podacima o skupu podataka

pipeline\_config**.**eval\_input\_reader**[**0**].**tf\_record\_input\_reader**.**input\_path**[:]** **=** **[**os**.**path**.**join**(**paths**[**'annotationPath'**],** 'test.record'**)]**

#Upisivanje dobijene konfiguracije u fajl

config\_text **=** text\_format**.**MessageToString**(**pipeline\_config**)**

**with** tf**.**io**.**gfile**.**GFile**(**files**[**'pipelineConfig'**],** "wb"**)** **as** f**:**

f**.**write**(**config\_text**)**

Код 4.2.3.2. Генерисање конфигурационог фајла

Након генерисања свих потребних фајлова, може се започети тренирање модела. Ово се постиже позивањем *Python* скрипте *model\_main\_tf2.py* из *TFOD API*. Као аргументи, прослеђују јој се путања до конфигурационог фајла, као и локацију на којој ће се периодично чувати параметри модела у тој етапи тренирања, као и коначни параметри и извештај о тренирању по завршетку. Евалуација модела извршава се коришћењем исте скрипте, са додатком аргумента који представља путању до сачуваних параметара модела.

Да бисмо изграђени модел користили на мобилном уређају, неопходно је конвертовати га у *TFLite* модел. *TFLite* је лака верзија библиотеке *TensorFlow* за покретање на мобилним уређајима и микроконтролерима. Поступак конверзије састоји се из два корака. Први корак јесте конверзија у међумодел, који је формата компатибилног са стандарном процедуром за конверзију у *TFLite* модел. Ово се постиже коришћењем *Python* скрипте *export\_tflite\_graph\_tf2.py*, доступне у оквиру *TFOD API*, и давањем путањи до конфигурације модела и његових параметара као аргументе. Добијени међумодел можемо конвертовати *TensorFlow* командом *tflite\_convert*, која за дати међумодел генерише тражени модел у виду једног *.tflite* фајла.

* + 1. Интеграција у мобилну апликацију

Замишљен рад детекције објеката у апликацији јесте приказ камере преко целог екрана, где се гранични правоугаоници исцртавају за детектоване карте у реалном времену. То значи да ћемо сваку слику коју камера ухвати, и коју приказујемо на екрану, довести на улаз наше мреже, и исцртавати граничне правоугаонике на основу њеног излаза.

За решавање специфичних проблема машинског учења везаних за слике, текст, или звук, *TensorFlow* нуди библиотеке за коришћење у апликацијама за *Android*, под називом *TensorFlow Lite Task.* За проблеме компјутерске визије, укључујући проблем детекције објеката, намењена је библиотека *TensorFlow Lite Task Vision*. Пре њеног коришћења, мора се иницијализовати позивом статичке методе *initialize* класе *TfLiteVision*. Уколико је иницијализација успешна, могу се користити класе из ове библиотеке. Главну апстракцију за моделе за детекцију објеката представља класа *ObjectDetector*. Објекат ове класе креира се давањем *.tflite* фајла са моделом, као и специфицирањем опција, као што су максималан број детекција по слици, или доња граница вероватноће за коју се детекција прихвата. Добијени објекат може да врши предикције за задату слику, при чему враћа листу објеката класе *Detection*. Ова класа енкапсулира координате граничног правоугаоника детекције, у виду горњег левог и доњег десног темена, као и вероватноћу за сваку класу објеката.

//Kreiranje detektora

ObjectDetector**.**ObjectDetectorOptions**.**Builder objOptionsBuilder **=** ObjectDetector**.**ObjectDetectorOptions**.**builder**()**

**.**setScoreThreshold**(**DETECTION\_THRESHOLD**).**setMaxResults**(**DETECTION\_MAX\_RESULTS**);**

BaseOptions**.**Builder baseOptionsBuilder **=** BaseOptions**.**builder**().**setNumThreads**(**DETECTION\_THREADS**);**

**try{**

baseOptionsBuilder**.**useGpu**();**

**}catch** **(**Exception e**){**

//Ne moze se koristiti GPU

**}**

objOptionsBuilder**.**setBaseOptions**(**baseOptionsBuilder**.**build**());**

**try{**

objDetector **=** ObjectDetector**.**createFromFileAndOptions**(**context**,** DETECTION\_MODEL\_NAME**,** objOptionsBuilder**.**build**());**

**}** **catch** **(**Exception e**){**

//Neuspesno kreiranje detektora

**}**

//Koriscenje detektora

//Rotiranje slike, radi se zbog rotacije mobilnog uredjaja

ImageProcessor imgProcessor **=** **new** ImageProcessor**.**Builder**()**

**.**add**(new** Rot90Op**(-**rotation **/** 90**)).**build**();**

TensorImage tensorImg **=** imgProcessor**.**process**(**TensorImage**.**fromBitmap**(**img**));**

List**<**Detection**>** detections **=** objDetector**.**detect**(**tensorImg**);**

Код 4.2.4.1. Креирање и коришћење објекта класе ObjectDetector

Да би се улаз са камере приказао на екрану, користићемо компоненту графичког интерфејса *PreviewView*, намењену управо за ову примену. Ову компоненту неопходно је повезати на изабрану камеру уређаја. При покретању апликације, мора се тражити дозвола од корисника за коришћење камере, у супротном неће бити приказа на екрану.

Preview preview **=** **new** Preview**.**Builder**()**

**.**setTargetAspectRatio**(**AspectRatio**.**RATIO\_4\_3**)**

**.**setTargetRotation**(**binding**.**cameraView**.**getDisplay**().**getRotation**())**

**.**build**();**

CameraSelector cameraSelector **=** **new** CameraSelector**.**Builder**()**

**.**requireLensFacing**(**CameraSelector**.**LENS\_FACING\_BACK**)**

**.**build**();**

cameraProvider**.**unbindAll**();**

camera **=** cameraProvider**.**bindToLifecycle**(** **this,** cameraSelector**,** preview**);**

//binding.cameraView je komponenta PreviewView

preview**.**setSurfaceProvider**(**binding**.**cameraView**.**getSurfaceProvider**());**

Код 4.2.4.2. Повезивање компоненте PreviewView са камером

За извршавање произвољног кода над улазном сликом камере, користи се класа *ImageAnalyzer*. Објекту ове класе поставља се ламбда функција, која прима слику као аргумент, којом се специфицира понашање над улазном сликом. Повезивање овог објекта са камером врши се додавањем још једног аргумента у позив *cameraProvider.bindToLifecycle* из кода 4.2.4.2. Овом класом прослеђиваћемо улаз са камере раније креираном детектору објеката.

За исцртавање граничних квадрата, креираћемо своју графичку компоненту, извођењем из класе *View*, која представља основну класу за компоненте интерфејса у *Android* апликацијама. Као поље ове класе додаћемо листу детекција, коју ћемо постављати након сваке предикције детектора објеката. За исцртавање на основу детекција, преклопићемо постојећу методу за цртање:

@Override

public void draw**(**Canvas canvas**)** **{**

**super.**draw**(**canvas**);**

**for** **(**Detection detection **:** detections**)** **{**

RectF boundingBox **=** detection**.**getBoundingBox**();**

float top **=** boundingBox**.**top **\*** scaleFactor**;**

float bottom **=** boundingBox**.**bottom **\*** scaleFactor**;**

float left **=** boundingBox**.**left **\*** scaleFactor**;**

float right **=** boundingBox**.**right **\*** scaleFactor**;**

RectF drawableRect **=** **new** RectF**(**left**,** top**,** right**,** bottom**);**

canvas**.**drawRect**(**drawableRect**,** boxPaint**);**

**}**

**}**

Код 4.2.4.3. Исцртавање граничних правоугаоника

* 1. Евалуација решења

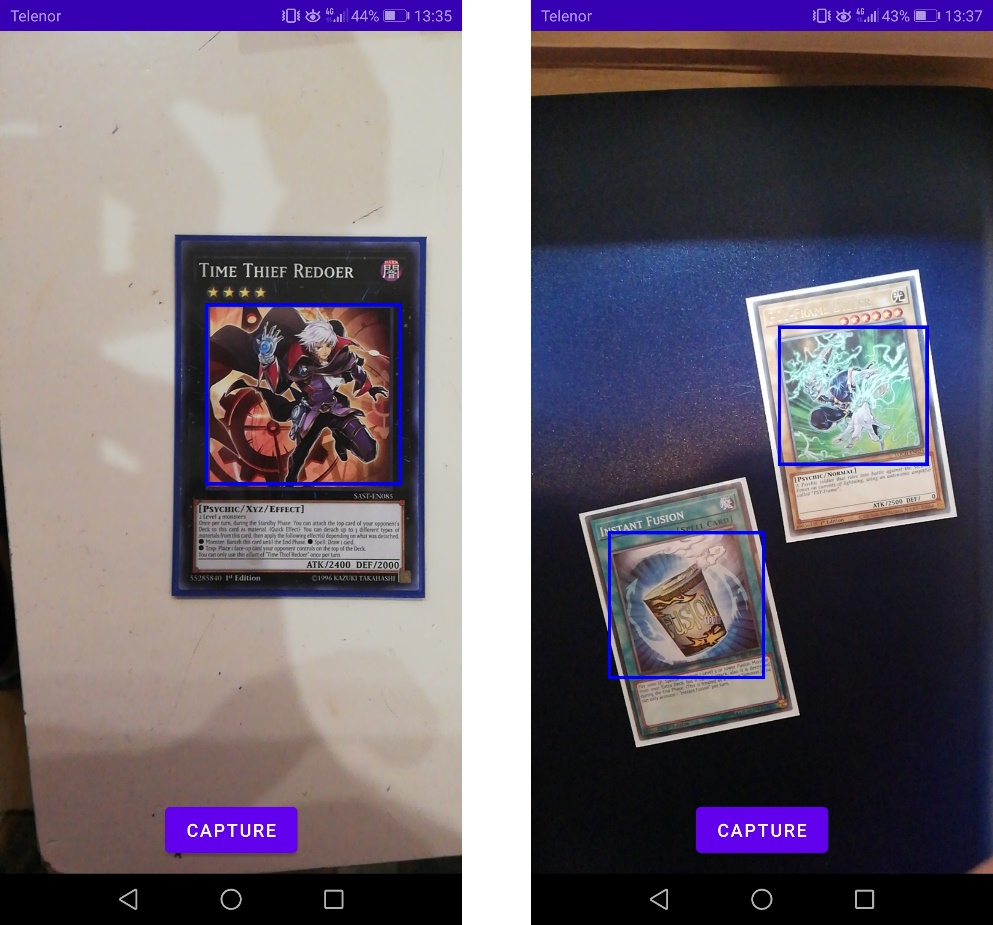
Две значајне метрике при евалуацији модела за детекцију објеката јесу прецизност (*precision*) и одзив (*recall*). Нека је дат број објеката коју су исправно детектовани (*true positive, TP*), број лажних детекција тамо где објеката нема (*false positive, FP*), број исправних непостојања детекција тамо где објеката нема (*true negative, TN*), као и број објеката који нису детектовани (*false negative, FN*). Прецизност представља удео детекција које су исправне

(4.3.1.)

Одзив представља удео објеката који су исправно детектовани [16]

(4.3.2.)

При евалуацији над скупом података за тестирање од 35 анотираних слика, наш модел за детекцију постиже прецизност од 0,8574, и одзив од 0,4135. Овај резултат се може побољшати проширењем скупа података за тренирање, продужавањам трајања тренирања, као и коришћењем неког од других понуђених модела са бољим метрикама.



Слика 4.3.1. Приказ рада детекције карата у оквиру апликације

При покретању прототипске апликације на мобилном телефону *Huaweii Mate 10 Lite*, модел показује време одзива од око 340ms. За доњи праг вероватноће детекције постављен на 0,5 при стварању детектора, систем детектује све карте над којима је тестиран, осим када су ротиране под углом од око 45°. Са друге стране, постоји приметан број лажних детекција. Ово је за потребе прототипске апликације у реду, пошто се лажне детекције могу игнорисати при разматрању резултата, док би немогућност детекције правих карата представљало велики проблем. Стога је ова итерација модела за детекцију објеката прихваћена за даље коришћење.

1. Проблем идентификације карата

У овом поглављу биће дискутовано решавање проблема идентификације карата на основу исечених илустрација добијених у претходном кораку. Пошто приступ решавању овог проблема у великој мери зависи од количине података коју поседујемо, најпре ће бити речи о процесу креирања скупа података за овај проблем, након чега ће бити разматрана могућа решења. За одабрани приступ решавању, биће приказани детаљи имплементације, као и евалуација добијеног решења, уз могућа побољшања.

* 1. Прикупљање података

За прикупљање информација о картама и њихових слика коришћен је јавно доступан API веб странице *YGOProDeck*. Слањем захтева за дохватање свих карата у игри, као одговор добили смо *JSON* поруку са комплетним информацијама. Једна од информација коју поседује свака карта јесте линк до дигиталне слике карте. Пример изгледа слика добијених на овај начин може се видети на слици 2.4.1. За преузимање свих слика итерирањем кроз карте написали смо једноставну *Python* скрипту. Резултујући скуп података састоји се од једне слике за сваку карту.

* 1. Принципи класификационих алгоритама машинског учења

Основни алгоритам за решавање класификационих проблема машинским учењем назива се логистичка регресија. Излаз модела логистичке регресије представља вероватноћу припадности улаза некој од могућих класа. У случају да се ради о бинарној класификацији, где постоје две могуће класе, решење се моделира сигмоид функцијом, која је у контексту коришћења за активацију неурона разматрана у потпоглављу 2.2.

(5.2.1.)

где представља улазни вектор својстава димензије *D*, вектор параметара димензије *D*, а реална константа. На овај начин, произвољне вредности из скупа реалних бројева мапирају се у реалну вредност из опсега (0,1), која представља вероватноћу да улаз припада једној од двеју класа. Пошто разматрамо проблеме где улаз припада само једној од могућих класа, на основу вероватноће припадности једној класи добијене на излазу модела, сазнаје се и вероватноћа припадности другој класи, једнака . За функцију губитка, у проблемима логистичке регресије користи се категоричка унакрсна ентропија (*Categorical Cross Entropy*)

(5.2.2.)

где представља број класа, представља дистрибуцију вероватноћа припадности свакој од класа за дати улаз, коју је генерисао модел, док представља очекивану припадност класама за дати улаз, где елемент који се односи на класу којој улаз припада има вредност 1, док остали елементи имају вредност 0. За случај бинарне класификације, функција се може једноставније написати

(5.2.3.)

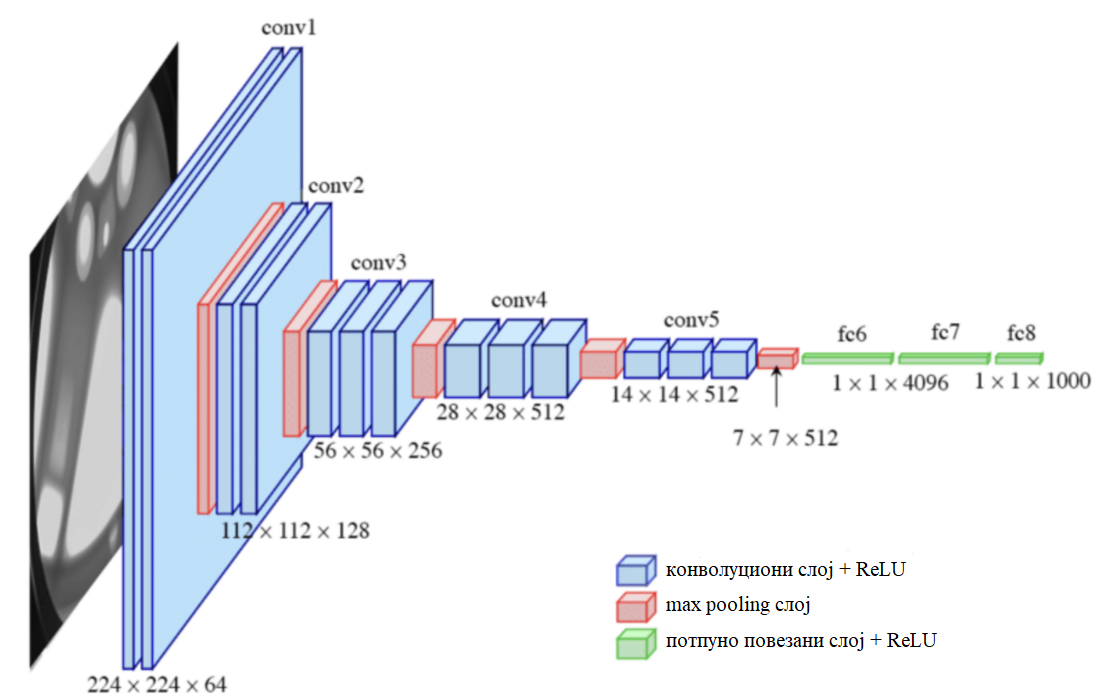
Оптимизациони критеријум који треба минимизовати јесте збир функције губитка за сваки податак из скупа података за тренирање. Типични оптимизациони алгоритми коришћени за решавање овог проблема јесу варијанте градијентног спуста. [1]

Уколико решавамо проблем вишеструке класификације, где постоји више од две могуће класе, функција која се користи за израчунавање вероватноћа припадности улаза за сваку од класа назива се *softmax* функција. Ова функција као улазну вредност прихвата вектор реалних бројева, димензије једнаке броју класа, а као излаз даје вектор реалних бројева из опсега (0,1), истих димензија, који представља дистрибуцију вероватноћа припадности улаза за сваку од класа. Свака вредност излазног вектора представља вероватноћу припадности за једну класу. Збир свих елемената излазног вектора једнак је 1. За улазни вектор величине , eлемент излазног вектора рачуна се по следећој формули:

(5.2.4.)

Избор функције губитка, оптимизационог критеријума као и оптимизационог алгоритма идентичан је као у случају бинарне класификације. [17]

Најчешћи приступ изградњи неуралних мрежа за класификационе проблеме јесте коришћење потпуно повезаног слоја као последњег слоја, са бројем неурона слоја једнаком броју класа, и са *softmax* функцијом као активационом функцијом слоја. Остатак топологије мреже прилагођава се специфичном проблему који се решава, као и типу улазних података. Пошто су код проблема класификације слика улазни податак слике, остатак неуралне мреже биће реализован у виду конволуционе неуралне мреже. Примери популарних архитектура неуралних мрежа са класификацију слика јесу *VGGNet*, *AlexNet*, и *ResNet*. [18]

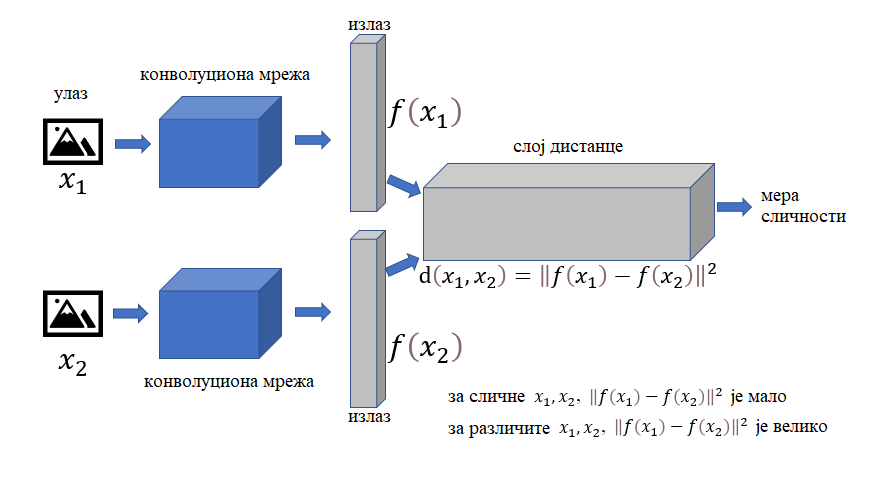


Слика 5.2.1. Визуелизација архитектуре VGG16 [19]

Како би модел за класификацију слика показивао добре перформансе, неопходно је да скуп података за тренирање садржи велики број примерака за сваку од класа. Како наш скуп података садржи само једну слику за сваку од изузетно великог броја класа, овај метод решавања нашег проблема је слабо применљив. Упркос томе, покушано је тренирање модела архитектуре *VGG16*, где је скуп података за тренирање сачињен од 24 варијанти сваке карте, генерисаних аугментацијом података, али се овај метод убрзо показао неприхватљивим, те нећемо детаљно разматрати његову имплементацију. Закључено је да морамо пронаћи другачији приступ решавању проблема.

* 1. О сијамским мрежама

Сијамска мрежа (*Siamese network*) представља архитектуру неуралне мреже која се састоји из пара мрежа „близнакиња“, истоветних топологија и дељених параметара, на које се доводи различити улаз. На излаз ових двеју мрежа надовезује се слој за мерење метрике раздаљине њихових излаза. Циљ је добити модел који даје малу раздаљину за сличне улазе, док је раздаљина за различите улазе велика. Мрежа је симетрична, тако да се заменом места улаза добија иста раздаљина. [20]



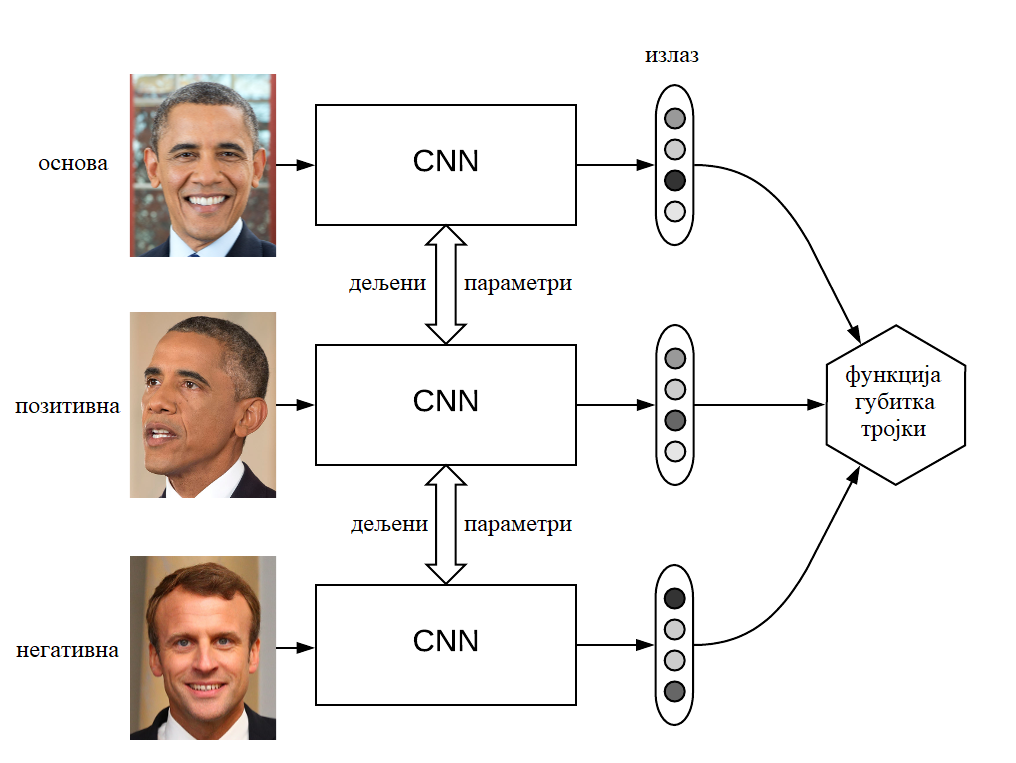
Слика 5.3.1. Приказ рада сијамске мреже [21]

Оваква архитектура користи се за решавање проблема где је неопходно добити меру сличности два улаза, као што је препознавање лица, верификација потписа, или детектовање дупликата. [1]

Приликом тренирања сијамске мреже, као функција губитка користи се функција губитка тројки (*triplet loss function*). За ову функцију неопходно је обезбедити три улаза – један који служи као основа (*anchor input*), један позитивни улаз (*positive input*) који је сличан основи, и један негативан улаз (*negative input*) који се разликује од основе. За парове улаза (*anchor*, *positive*) и (*anchor*, *negative*) рачуна се дистанца коришћењем сијамске мреже која се тренира. Функција губитка тројки представља разлику ове две дистанце

(5.3.1.)

где су основа, позитивни и негативни улаз, редом, је функција дистанце, док је константна вредност изабрана приликом пројектовања. Да би се функција минимизовала, неопходно је да дистанца између сличних улаза буде што мања, док дистанца између различитих улаза треба да буде што већа. Оптимизациони критеријум представља просек функције губитка за скуп података за тренирање, док је оптимизациони алгоритам, као и раније, једна од варијанти градијентног спуста. [22]



Слика 5.3.2. Приказ рада функције губитка тројки [23]

Сијамске мреже погодне су за коришћење у случајевима где постоји велики број класа, док скуп података садржи мали број примерака по класи. Пошто ово одговара нашем случају, определили смо се за коришћење ове архитектуре за изградњу модела за идентификацију карата.

* 1. Имплементација решења

Ово потпоглавље посвећено је детаљима имплементације модела сијамске мреже. Биће приказан поступак креирања скупа података, имплементације мреже, тренирања и интеграције у мобилну апликацију, за итерацију модела која је дала најбоље резултате приликом тестирања.

* + 1. Припрема скупа података

Пошто се на улазу мреже очекује илустрација са карте, за скуп података искористићемо илустрације са преузетих слика карата, које ћемо програмски исећи. Ово можемо учинити захваљујући чињеници да илустрације на свакој карти заузимају једну од двеју могућих површина. О којој врсти илустрације се ради може се утвдити провером информација о карти, пошто карте са широм илустрацијом у свом типу поседују кључну реч *Pendulum*. Пример обе врсте илустрације може се видети на слици 2.4.1.

За коришћење функције губитка тројки неопходна је више од једне слике по класи, како би се могли створити слични парови. Стога смо најпре генерисали 5 варијанти сваке карте, ротацијом и искошењем за мали угао, модификацијом светлине, контраста и јачине боја на слици, као и замућењем. Све ове трансформације раде се са насумичном вероватноћом, и слику модификују за насумичну вредност. Трансформације се врше пре сечења, како би ротирана илустрација била окружена реалним ободом карте, уместо алгоритамски генерисаним попуњавањем. За генерисање трансформисаних карата коришћена је библиотека *Albumentations*. Приликом ротације и искошења карте, мења се и позиција илустрације, те сечење не можемо више радити на фиксној, очекиваној позицији. Решење овог проблема лежи у компатибилности библиотеке *Albumentations* са скуповима података за детекцију објеката. Уколико се уз слику проследе координате граничних правоугаоника објеката, библиотека ће, уз трансформисану слику, вратити и модификоване координате граничних правоугаоника, који окружују објекте на трансформисаној слици. Зато ћемо, уз слику карте, проследити координате правоугаоника око илустрације, и за сечење користити враћени, модификовани правоугаоник.

skewAugment **=** A**.**Compose**([**

A**.**Affine**(**rotate**=[-**5**,**5**],** shear**=[-**5**,**5**],** p**=**1**),**

**],** bbox\_params**=**A**.**BboxParams**(**format**=**'pascal\_voc'**,** label\_fields**=[**'class\_labels'**]))**

**def** skewImage**(**imageObj**,** bbox**):**

skewedImageData **=** skewAugment**(**image**=**np**.**array**(**imageObj**),** bboxes**=[**bbox**],** class\_labels**=[**'card'**])**

dstImage **=** Image**.**fromarray**(**skewedImageData**[**'image'**])**

**return** dstImage**,** skewedImageData**[**'bboxes'**][**0**]**

augment **=** A**.**Compose**([**

A**.**RandomBrightnessContrast**(**p**=**0.5**,** brightness\_limit**=[-**0.2**,**0.3**],** contrast\_limit**=[-**0.4**,**0.4**]),**

A**.**HueSaturationValue**(**hue\_shift\_limit**=[-**10**,**10**],** sat\_shift\_limit**=[-**20**,** 20**],** val\_shift\_limit**=[-**15**,**15**],** p**=**0.5**),**

A**.**GaussianBlur**(**p**=**0.3**),**

A**.**MedianBlur**(**blur\_limit**=**3**,** p**=**0.3**),**

**])**

**def** augmentImage**(**imageObj**):**

augmentImageData **=** augment**(**image**=**np**.**array**(**imageObj**))**

**return** Image**.**fromarray**(**augmentImageData**[**'image'**])**

Код 5.4.1.1. Функције за трансформацију карата

Скуп података за тренирање треба да се састоји од тројки, на начин који је објашњен у потпоглављу 5.3. при разматрању функције губитка тројки. Због уштеде на времену тренирања, нисмо правили парове сличних слика користећи све варијације једне карте, већ смо као основу фиксирали немодификовану илустрацију карте. док смо као позитивну користили сваку од варијација карте. За негативну слику, насумично је биран примерак друге карте. На овај начин, добили смо скуп података који се састоји од 5 тројки за сваку карту.

image\_count **=** len**(**imageClasses**)**

**for** imageClass **in** imageClasses**:**

imageClassFolder **=** os**.**path**.**join**(**imagesPath**,** imageClass**)**

images **=** sorted**(**os**.**listdir**(**imageClassFolder**))**

anchorImagePath **=** os**.**path**.**join**(**imageClassFolder**,** images**[**0**])**

**for** i **in** range**(**1**,** len**(**images**)):**

positiveImagePath **=** os**.**path**.**join**(**imageClassFolder**,** images**[**i**])**

negativeImageClass **=** rng**.**choice**(**imageClasses**)**

**while** negativeImageClass **==** imageClass**:**

negativeImageClass **=** rng**.**choice**(**imageClasses**)**

negativeImageClassFolder **=** os**.**path**.**join**(**imagesPath**,** negativeImageClass**)**

negativeImagePath **=** os**.**path**.**join**(**negativeImageClassFolder**,** rng**.**choice**(**os**.**listdir**(**negativeImageClassFolder**)))**

anchorImages**.**append**(**anchorImagePath**)**

positiveImages**.**append**(**positiveImagePath**)**

negativeImages**.**append**(**negativeImagePath**)**

Код 5.4.1.2. Креирање скупа података

Пре довођења на улаз мреже, слику је неопходно прилагодити димензије очекиваним димензијама улаза. Ово ћемо укључити у препроцесирање, процес који се врши пре довођења улаза на мрежу, како би се улаз прилагодио мрежи и проблему. Осим промене величине, променили смо слику у формат нијанси сиве, као и применили насумично сечење, где задржавамо сегмент слике који чини 85% оригинала. Ово је вид аугментације података који смо успели да интегришемо у препроцесирање, где се при сваком препроцесирању исте слике одсеца нови, насумични сегмент. На овај начин повећавамо количину информација скупа података при тренирању.

A picture containing text, old, different

Description automatically generated

Слика 5.4.1.1. Пример тројки из добијеног скупа података

* + 1. Изградњња архитектуре и тренирање

Као и приликом изградње модела за детекцију карата, употребићемо технику учења трансфером да бисмо убрзали процес тренирања и што брже постигли прихватљиве резултате. Библиотека *Keras* за ове потребе нуди моделе архитектура *ResNet*, трениране над скупом података *ImageNet*, који садржи око 14 милиона слика подељених у око 20 хиљада класа. Определили смо се за архитектуру *ResNet* са 50 слојева, познату под именом *ResNet50*. Пошто је ово архитектура за решавање проблема класификације слика, одбацићемо потпуно повезане слојеве којима се мрежа завршава, и који су прилагођени проблему класификације над скупом података *ImageNet*. Уместо њих, надовезаћемо сопствене потпуно повезане слојеве, од којих ће последњи бити величине 256, што ће представљати величину излазног вектора својстава слике.

base\_cnn **=** resnet**.**ResNet50**(**

weights**=**"imagenet"**,** input\_shape**=(**200**,**200**,**3**),** include\_top**=False**

**)**

flatten **=** layers**.**Flatten**()(**base\_cnn**.**output**)**

dense1 **=** layers**.**Dense**(**512**,** activation**=**"relu"**)(**flatten**)**

dense1 **=** layers**.**BatchNormalization**()(**dense1**)**

dense2 **=** layers**.**Dense**(**256**,** activation**=**"relu"**)(**dense1**)**

dense2 **=** layers**.**BatchNormalization**()(**dense2**)**

output **=** layers**.**Dense**(**256**)(**dense2**)**

embedding **=** Model**(**base\_cnn**.**input**,** output**)**

Код 5.4.2.1. Креирање основне неуралне мреже

Добијена мрежа представљаће основу наше сијамске мреже, односно то ће бити мрежа коју ћемо у два примерка, односно три за време тренирања, повезати на слој за одређивање дистанце.

Пре имплементације слоја дистанце, неопходно је изабрати метрику дистанце коју ћемо користити. Три најчешће коришћене метрике за меру сличности два вектора реалних бројева јесу:

* еуклидска дистанца
* косинус угла
* скаларни производ

Одлучили смо да користимо еуклидску дистанцу као метрику.

**class** **DistanceLayer(**layers**.**Layer**):**

**def** \_\_init\_\_**(**self**,** **\*\***kwargs**):**

super**().**\_\_init\_\_**(\*\***kwargs**)**

**def** call**(**self**,** anchor**,** positive**,** negative**):**

ap\_distance **=** tf**.**reduce\_sum**(**tf**.**square**(**anchor **-** positive**),** **-**1**)**

an\_distance **=** tf**.**reduce\_sum**(**tf**.**square**(**anchor **-** negative**),** **-**1**)**

**return** **(**ap\_distance**,** an\_distance**)**

anchor\_input **=** layers**.**Input**(**name**=**"anchor"**,** shape**=(**200**,** 200**,** 3**))**

positive\_input **=** layers**.**Input**(**name**=**"positive"**,** shape**=(**200**,** 200**,** 3**))**

negative\_input **=** layers**.**Input**(**name**=**"negative"**,** shape**=(**200**,** 200**,** 3**))**

distances **=** DistanceLayer**()(**

embedding**(**resnet**.**preprocess\_input**(**anchor\_input**)),**

embedding**(**resnet**.**preprocess\_input**(**positive\_input**)),**

embedding**(**resnet**.**preprocess\_input**(**negative\_input**)),**

**)**

siamese\_network **=** Model**(**

inputs**=[**anchor\_input**,** positive\_input**,** negative\_input**],** outputs**=**distances

**)**

Код 5.4.2.2. Креирање и везивање слоја дистанце

Последњи корак имплементације мреже је дефинисање функције губитка и оптимизационог критеријума. Ово ћемо учинити извођењем сопствене класе модела, у којој ћемо специфицирати поступке за тренирање и валидацију који ће користити функцију губитка коју смо написали. За добијање оптимизационе функције која представља аритметичку средину, користићемо објекат класе *keras.metrics.Mean*, у којој ћемо акумулирати вредност фунцкије губитка у сваком кораку тренинга и валидације, док ћемо тренутну вредност средине вратити као тренутну вредност оптимизационог критеријума. Стање овог објекта ресетује се након сваке епохе.

**class** **SiameseModel(**Model**):**

**def** \_\_init\_\_**(**self**,** siamese\_network**,** margin**=**0.5**):**

super**(**SiameseModel**,** self**).**\_\_init\_\_**()**

self**.**siamese\_network **=** siamese\_network

self**.**margin **=** margin

self**.**loss\_tracker **=** metrics**.**Mean**(**name**=**"loss"**)**

**def** call**(**self**,** inputs**):**

**return** self**.**siamese\_network**(**inputs**)**

**def** train\_step**(**self**,** data**):**

**with** tf**.**GradientTape**()** **as** tape**:**

loss **=** self**.**\_compute\_loss**(**data**)**

gradients **=** tape**.**gradient**(**loss**,** self**.**siamese\_network**.**trainable\_weights**)**

self**.**optimizer**.**apply\_gradients**(**

zip**(**gradients**,** self**.**siamese\_network**.**trainable\_weights**)**

**)**

self**.**loss\_tracker**.**update\_state**(**loss**)**

**return** **{**"loss"**:** self**.**loss\_tracker**.**result**()}**

**def** test\_step**(**self**,** data**):**

loss **=** self**.**\_compute\_loss**(**data**)**

self**.**loss\_tracker**.**update\_state**(**loss**)**

**return** **{**"loss"**:** self**.**loss\_tracker**.**result**()}**

**def** \_compute\_loss**(**self**,** data**):**

ap\_distance**,** an\_distance **=** self**.**siamese\_network**(**data**)**

loss **=** ap\_distance **-** an\_distance

loss **=** tf**.**maximum**(**loss **+** self**.**margin**,** 0.0**)**

**return** loss

@property

**def** metrics**(**self**):**

**return** **[**self**.**loss\_tracker**]**

Код 5.4.2.3. Имплементација модела који користи функцију губитка тројки

Као оптимизациони алгоритам модела, користићемо оптимизатор *Adam*. Модел је успешно трениран 19 епоха, док су за параметре модела који ће се користити изабрани они који су добијени у 17. епохи због најбоље вредности оптимизационе функције.

Да би се, користећи добијени модел, идентификовала карта на улазу, неопходно је пронаћи дистанцу излазног вектора својстава карте на улазу од излазног вектора сваке карте коју поседујемо у скупу података. Карта са најмањом дистанцом од улаза представљаће предикцију модела. Како су вредности излазног вектора за карте из скупа података константне, а њихово рачунање при свакој предикцији би трајало предуго, сачуваћемо излазне векторе у речник, чији је кључ идентификатор карте за коју је излазни вектор израчунат. Овај речник извешћемо у фајл *JSON* формата, ради увоза у мобилну апликацију. За предикцију је сада довољно довести само улазну слику на улаз неуралне мреже, и добијени резултат поредити са вредностима из речника.

**def** predict**(**imagePath**):**

imageObj **=** preprocess\_image**(**imagePath**)**

imageObj **=** np**.**expand\_dims**(**imageObj**,** axis**=**0**)**

imageObj **=** tf**.**constant**(**imageObj**)**

val **=** embedding**(**resnet**.**preprocess\_input**(**imageObj**)).**numpy**().**tolist**()[**0**]**

current\_max **=** **-**1

current\_card **=** **None**

**for** cardId**,** cardValues **in** embeddingValues**.**items**():**

**for** cardVal **in** cardValues**:**

similarity **=** norm**(**np**.**subtract**(**val**,** cardVal**))**

**if** current\_card **is** **None** **or** similarity **<** current\_max**:**

current\_max **=** similarity

current\_card **=** cardId

**return** current\_card**,** current\_max

Код 5.4.2.4. Функција за предикцију карте

У речнику нисмо чували резултате неуралне мреже за генерисане варијанте карата из скупа података, већ само за немодификоване илустрације, како време прављења предикције не би било предуго.

* + 1. Интеграција у мобилну апликацију

Као што је био случај са интеграцијом модела за детекцију, модел за идентификацију интегрисаћемо његовим извожењем у *TFLite* формат модела. За прављење предикција неопходна је само мрежа за рачунање вектора својстава, тако да ћемо извести само њу, без удвајања и слоја дистанце. За ову мрежу није нам неопходна додатна библиотека специјализована за одређену врсту проблема, као што је био случај са детекцијом објеката, већ ћемо само користити библиотеку *TFLite*. Класа *Interpreter* из ове библиотеке представља апстракцију модела. Објекат ове класе креаира се давањем путање до фајла са моделом. Након креирања, модел се покреће коришћењем методе *run*, којој се даје референца на низ са улазним подацима, као и референца на низ у којем ће модел сачувати резултат. Притиском на дугмe у апликацији, врши се поновна детекција објеката над тренутним приказом камере. Ово се ради да би се избегао случај где приказ камере и исцртани правоугаоници нису синхронизовани услед наглог покрета. Детектоване илустрације карата исецају се и дају моделу на предикцију. Резултати предикције приказују се на посебном прозору.

public void identify**(**DetectionResult result**)** **{**

Bitmap preprocessedBitmap **=** Bitmap**.**createScaledBitmap**(**result**.**bitmap**,** IMAGE\_WIDTH**,** IMAGE\_HEIGHT**,** **true);**

preprocessedBitmap **=** toGrayscale**(**preprocessedBitmap**);**

float**[][][][]** imgValues **=** **new** float**[**1**][**IMAGE\_WIDTH**][**IMAGE\_HEIGHT**][**3**];**

imgValues**[**0**]** **=** preprocessInputCaffe**(**preprocessedBitmap**);**

float**[][]** output **=** **new** float**[**1**][**256**];**

interpreter**.**run**(**imgValues**,** output**);**

String nearestCardId **=** findNearestCard**(**output**[**0**]);**

**if(**nearestCardId **!=** **null)** **{**

String nearestCardName **=** cardNames**.**get**(**nearestCardId**);**

result**.**cardName **=** nearestCardName**;**

**}** **else{**

result**.**cardName **=** "Error"**;**

**}**

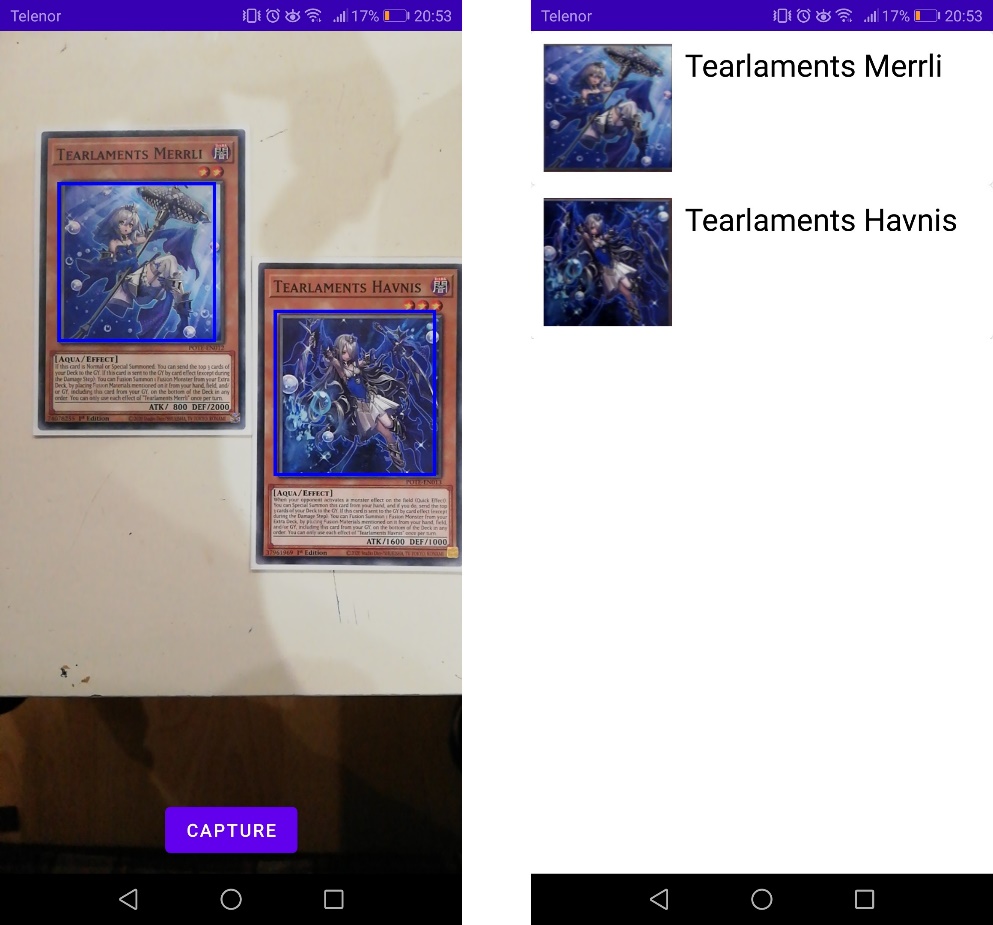
**}**

Код 5.4.3.1. Фунцкија за идентификацију карте

Као и приликом тренирања, најпре је потребно извршити препроцесирање слике, променом њених димензија, претварањем у нијансе сиве, као и претварањем у *Caffe* формат, који мрежа *ResNet* из *Keras* библиотеке очекује на улазу. Последња трансформација еквивалентна је позиву *resnet.preprocess\_input* из кодова 5.4.2.2. и 5.4.2.4. Кôд за проналажење најсличније карте је функционално еквивалентан коду 5.4.2.4. те неће бити приказан. Имена карата добијена су из *JSON* фајла са комплетним подацима карата.

* 1. Евалуација решења

Скуп података за тестирање сачињен је од 1000 насумично изабраних карата, над којим су извршене трансормације налик оним које су рађене приликом креирања скупа података за тренирање. Над овим скупом, модел показује тачност од 47.2%. Приликом покретања у оквиру мобилне апликације, време одзива модела је око 3s. Тестирање у оквиру апликације извршено је над око 100 правих карата, где је модел показао солидне резултате.



Слика 5.5.1. Приказ рада идентификације карата у оквиру апликације

Као што је био случај и код модела за детекцију, време тренирања у епохама је било кратко. Стога се резултати могу побољшати повећањем времена тренирања. Перформансе модела најбоље су приликом идентификације усправних карата под равномерном, умереном светлошћу. Побољшање перформанси над картама у различитим положајима и околинама постигло би се проширењем скупа података примерима веће разноликости. Један од начина за то би била модификација коришћених техника аугментације података, где би се трансформације укључиле у препроцесирање скупа података, а интензитет рађених трансформација повећао. Приликом тестирања стечен је утисак да идентификација даје боље резултате над картама чије су илустрације једноставније. Ово може сугерисати да је потребно искористити комплекснију архитектуру мреже, која би била способна да препозна комплексније облике на слици. Разматрана је изградња једне итерације модела коришћењем архитектуре *ResNet101*, варијанте *ResNet* са 101 слојем. Нови модел није показао значајно боље резултате од постојећег, што се поново може објаснити кратким периодом тренирања, на које би комплекснији модел био осетљивији. Услед ограниченог времена, није било могуће донети значајнију евалуацију примене ове архитектуре на наш проблем.

1. Закључак

У овом раду разматране су технике машинског учења коришћене у решавању проблема из домена компјутерске визије, и могућност њихове примене за изградњу система за препознавање карата у оквиру стоне игре картама *Yu-Gi-Oh!*, намењеног мобилним уређајима. Постављене су теоријске основе машинског учења и неуралних мрежа, као и принципи рада специфичних архитектура за решавање проблема детекције објеката, класификацију слика, и мерења сличности. Дискутован је избор коришћених архитектура, прилагођен нашем проблему и расположивом скупу података. Разматрано је коришћење сијамских мрежа за решавање проблема препознавања у случајевима оскудних расположивих података. Приказан је значај коришћења технике учења трансфером, са циљем побољшања времена тренирања и иницијалних перформанси модела, као и коришћење технике аугментације података за проширење малих скупова података, који је у нашем случају садржао само једну слику по карти. Описан је комплетан поступак изградње модела и његове интеграције у *Android* мобилну апликацију, коришћењем специјализованих библиотека, алата и окружења. На крају, евалуиране су перформансе модела, програмским тестирањем над скупом слика на рачунару, као и тестирањем у оквиру мобилне апликације над правим картама, и дати могући начини да се модели побољшају.

На основу резултата добијених тестирањем имплементираног система, закључујемо да су коришћене методе из области машинског учења применљиве на решавање датог проблема. Иако тренутна итерација решења не постиже ниво резултата који би били неопходни за оправдано коришћење у стварној апликацији, нису неопходне значајне промене у приступу решавању, нарочито пошто се велики део недостатака у имплементацији може објаснити недостатком времена и хардверских ресурса. Применом побољшања решења, предложених у оквиру рада, могао би се изградити модел задовољавајућих перформанси. Овакво решење може се применити на друге стоне игре картама, као и на било који проблем препознавања са великим бројем класа и ограниченим бројем примерака у скупу података.

Литература

1. Andriy Burkov, The Hundred-page Machine Learning Book, Andriy Burkov, 2019.
2. Joseph Awoamim Yacim, Douw Boshoff, „Impact of Artificial Neural Networks Training Algorithms on Accurate Prediction of Property Values“, *Journal of Real Estate Research*, 40(3), 375-418, 2018.
3. Stuart Russell, Peter Norvig, Artificial Intelligence: A Modern Approach 3rd Edition, Pearson, 2009.
4. Alex K, An Introduction to Neural Networks, <https://medium.com/swlh/an-introduction-to-neural-networks-de70cb4305f9>, 22.09.2022.
5. CS321n Convolutional Neural Networks for Visual Recognition, <https://cs231n.github.io/convolutional-networks>, 23.09.2022.
6. Yu-Gi-Oh! Card Database – YGOProDeck, <https://ygoprodeck.com/card-database>, 23.09.2022.
7. tensorflow/tensorflow: An Open Source Machine Learning Framework for Everyone, <https://github.com/tensorflow/tensorflow>, 26.09.2022.
8. Keras: the Python deep learning API, <https://keras.io/>, 26.09.2022.
9. Download Android Studio & App Tools – Android Developers, <https://developer.android.com/studio>, 26.09.2022.
10. YOLO: Real-Time Object Detection Explained, <https://www.v7labs.com/blog/yolo-object-detection>, 24.09.2022.
11. Intersection over Union (IoU) – Hasty.ai, <https://hasty.ai/docs/mp-wiki/metrics/iou-intersection-over-union>, 24.09.2022.
12. Object Detection in 2022: The Definitive Guide – viso.ai, <https://viso.ai/deep-learning/object-detection>, 24.09.2022.
13. Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, „Rich feature hierarchies for accurate object detection and semantic segmentation“, *IEEE Conference on Computer Vision and Pattern Recognition, 2014*.
14. Jeremy Jordan, An overview of object detection: one stage methods, <https://www.jeremyjordan.me/object-detection-one-stage/>, 25.09.2022.
15. Overview | Protocol Buffers | Google Developers, <https://developers.google.com/protocol-buffers/docs/overview>, 25.09.2022.
16. Classification: Precision and Recall | Machine Learning | Google Developers, <https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall>, 26.09.2022.
17. Gabriel Furnieles, Sigmoid and SoftMax Function in 5 minutes, <https://towardsdatascience.com/sigmoid-and-softmax-functions-in-5-minutes-f516c80ea1f9>, 26.09.2022.
18. Derrick Mwiti, Image Classification with Convolutional Neural Networks (CNNs), <https://www.kdnuggets.com/2022/05/image-classification-convolutional-neural-networks-cnns.html>, 26.09.2022.
19. Max Ferguson, Ronay Ak, Yung-Tsun Tina Lee, Kincho H. Law, „Automatic localization of casting defects with convolutional neural networks“, *IEEE international conference on big data*, 2017.
20. Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, „Siamese Neural Networks for One-shot Image Recognition“, *ICML deep learning workshop*, 2015.
21. Renu Khandelwal, One-Shot Learning With Siamese Network, <https://medium.com/swlh/one-shot-learning-with-siamese-network-1c7404c35fda>, 27.09.2022.
22. Florian Schroff, Dmitry Kalenichenko, James Philibin, „FaceNet: A unified embedding for face recognition and clustering“, *IEEE Conference on Computer Vision and Pattern Recognition, 2015.*
23. Oliver Moindrot, Triplet Loss and Online Triplet Mining in TensorFlow, <https://omoindrot.github.io/triplet-loss>, 27.09.2022.