Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Antal Zsolt

2022

Szegedi Tudományegyetem
Informatikai Intézet

Levelek és betegségeik felismerése TensorFlow-al

Szakdolgozat

Készítette:
Antal Zsolt
programtervező
informatikus szakos
hallgató

Témavezető:
Dr. Varga László Gábor
egyetemi adjunktus

Szeged
2022

Feladatkiírás

A jelentkező feladata annak vizsgálata, hogy konvolúciós mély neurális hálózatok betanításával milyen képfeldolgozási feladatok végezhetők el.
A feladat megoldásához egy kommerciálisan is elérhető konvolúciós neurális hálózat használatával kell feladatokat megoldani. A tanítás különböző típusú adatbázisokat használhat, amelyek között lehet orvosi adat, hagyományos kamera felvételek, légifelvételek. stb.

Tartalmi összefoglaló

- **Téma megnevezése:** Levek és azok betegségeinek felismerése TensorFlow-al
- **Megadott feladat megfogalmazása:** A dolgozat célja egy olyan Android applikáció készítése, ami képes alkalmazni az eszköz kameráját illetve az eszköz galériáját egy statikus kép bekérésére és erről a képről egy egyénileg előre betanított TensorFlow Lite modell segítségével képes beazonosítani róla levelek típusát illetve akár esetleges betegségüket. Ezen felül legyen képes a felettébb említett modell segítségével valós időben felismerni leveleket illetve betegségüket a kamera kép folyamatos kiértékelésével. Az applikációnak ki kell írnia becsléseinek százalékát is, illetve ha ez a százalék nem haladja meg a 92%-ot akkor írja ki, hogy nem beazonosítható a kép. Az applikáció minden esetben rajzolja ki a képet, amit használt az aktuális becslésére.
- **Megoldási mód:** Az applikáció Java-ban íródott Android Studio-ban, illetve alkalmaz Androidx könyvtárakat. A betanított mesterséges intelligencia modell a TensorFlow python könyvtárával íródott PyCharm-ban.
- **Elért eredmények:** A tanító python script átlagosan 2GB memóriát foglal le, mert a tanítás folyamán dinamikusan bufferel elemeket. A létrehozott TensorFlow modell körülbelül 98.5% pontossággal képes megbecsülni a dataset képeinek osztályát. A létrehozott applikáció képes egy TensorFlow Lite modell segítségével képekre becslési értékeket adni. Az applikáció ezen felül képes az eszköz alapértelmezett kameráját, galériáját valamint akár a kamerájának az előnézeti képeit használni kiértékelésre.

- **Kulcsszavak:** Android, TensorFlow, képfelismerés

Tartalomjegyzék

1. Bevezetés

A legtöbb modern mobiltelefon, illetve tablet már rendelkezik vagy gyárilag beépített képfelismerő mesterséges intelligenciával, ami az eszköz kamerája segítségével képes beazonosítani rengeteg hétköznapi eszközt, növényeket, járműveket és egyéb alapvető entitásokat. Ezek azonban széleskörű implementációk, amik nem feltétlen azt mondják meg egy növényről, hogy mi a fajtája, hanem, csak például annyit hogy az egy fa vagy bokor, vagy virág. Jelen diplomamunka célja egy olyan applikáció létrehozása, ami kép alapján képes felismerni növények leveleinek típusát és esetleges betegségét, de specializálódik pár haszonnövény leveleire, illetve ezek pár gyakori, levelükről beazonosítható betegségeikre. A képfelismerésért egy TensorFlow modell fog felelni, ami a már előre tanított MobileNet modellt veszi alapul, hogy kellően gyors és kicsi legyen a modell egy átlagos Android alapú eszköz számára. A modell becslési pontossága el kell érjen egy reális használhatóságot. Az applikáció használja az eszköz kameráját és galériáját a felhasználó tetszése szerűen. Az applikáció egy kép kiválasztása után kiírja a megbecsült levéltípust és betegségét.

2. Felhasznált eszközök

Ebben a fejezetben ismertetni fogom az applikáció, illetve a TensorFlow modell elkészítéséhez alkalmazott eszközöket, módszereket, valamint adatokat. Az applikáció fejlesztésének főbb lépései: Tanító dataset kiválasztása, annak esetleges standardizálása, betanított modell létrehozása python scriptel, Android applikáció fejlesztése.

2.1 Tanító dataset

Minden TensorFlow modell betanításához kell egy lehetőleg elégségesen nagyméretű dataset ami valamilyen módon osztályozva van. Mivel az applikáció elsődleges célja levéltípusok felismerése, így kizárólag leveleket tartalmazó dataset-re van szükségem. A számomra legideálisabb publikusan elérhető dataset egy 88000 képet tartalmazó 38 osztályba rendezett volt. Minden tartalmazott kép mérete 256*256 pixel, valamint kiterjesztésük JPG volt. Ez a dataset előre szét volt osztva tanító és ellenőrző szekciókra, valamint már előre tartalmazta a képek elforgatott, valamint tükrözött verzióit, így növelve a tartalmazott képek számát. Ezeket a legtöbb dataset nem szokta tartalmazni, mert le lehet őket könnyen generálni a TensorFlow tanítása közben, de a meglétük egy kevés feldolgozási időt tud spórolni, a foglalt tárhely megsokszorozásáért cserébe. Itt ez nem volt jelentős probléma, mert még így is a dataset teljes mérete kevesebb, mint 1,5 GB volt. A dataset hála az egységes formátumnak és felbontásnak teljesen standardizált, tehát standardizálással nem kell foglalkozni a Python scrit-en belül.

2.2 Python / TensorFlow

A Python egy általános célú, nagyon magas szintű programozási nyelv, többek között a funkcionális, az objektumorientált, az imperatív, és a procedurális programozási paradigmákat támogatja. A TensorFlow pedig egy ingyenesen elérhető open-source könyvtár gépi tanuláshoz és mesterséges intelligenciához. Kiválasztásuk fő szempontja az egyszerű felhasználás és a bőséges publikusan elérhető segéd információ illetve dokumentáció volt. Mivel a szakdolgozat célja egy egyénileg betanított mesterséges intelligencia felhasználása, így a tanításra egy Python script teljesen ideális, mivel a TensorFlow modellt csak egyszer kell betanítani és nem szükséges, valamint az Android eszközök erőforrásaiból adódó

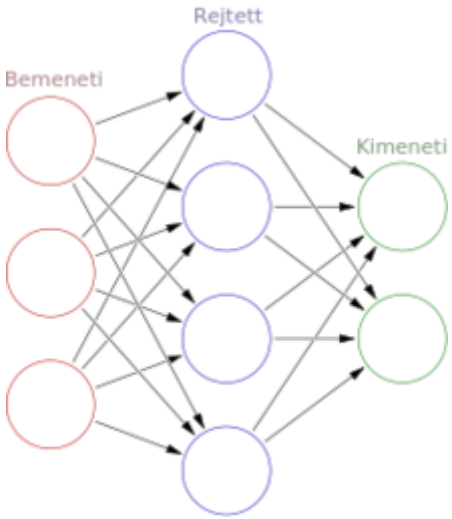
limitációk miatt nem is célszerű, hogy az Android applikáció tovább tanítsa a modellt valamilyen módon.

2.3 Java / Android

A Java egy általános célú, objektumorientált programozási nyelv amelyet a Sun Microsystems, majd később az Oracle fejlesztett. Az Android egy mobil operációs rendszer ami egy modifikált Linux kernelen és más open-source szoftveren alapszik és elsődlegesen érintő kijelzős készülékekre lett tervezve. Androidon a két legelterjedtebb programozási nyelv amiben applikációkat lehet írni, az a Java és a Kotlin. Az Android könyvtárai főként csak ezt a két nyelvet támogatják, tehát ezek közül kellett választanom és mivel Java-val jóval több tapasztalatom van így ezt választottam. Az applikáció egy jelentős részét az Androidx könyvtárak segítségével fejlesztettem. Ennek fő okai a relatíve egyszerű használatuk, és a rendkívül hasznos funkcióik voltak.

3. Neuron hálók

A mesterséges neuron hálók a biológiai neuron hálók működése inspirálta adatfeldolgozó rendszerek. Ezeket a hálókat node-okból építjük fel és az így kapott háló egy biológiai neuron hálóra hasonlít működésileg. Minden node-ot összekötünk más node-okkal és ezek az élek, hivatottak jelek vagy adatok átadására az egyik node-ból a másikba. A hálókat általában gráf szerű ábrákkal szoktuk vizualizálni, hogy könnyebben áttekinthetőek legyenek. A node-ok valamint szintekbe azaz layer-ekbe sorolhatóak és egy adott layer node-jai egymástól függetlenek. A legelső layer a bemenete és a legutolsó pedig a kimenete a halónak. Ezt a szerkezetet szemlélteti lentebb a 3-1. ábra.



3-1. ábra: egy neurál háló felépítésének szemléltetése

Egy kialakított hálóban általában súlyozva vannak mind az élek és mind a node-ok, hogy befolyásoljuk a tanítási folyamatot. Ezek a súlyozások hivatottak prioritásokat valamint akár elfogultságokat kialakítani a háló számára.

3.1 Tanítás

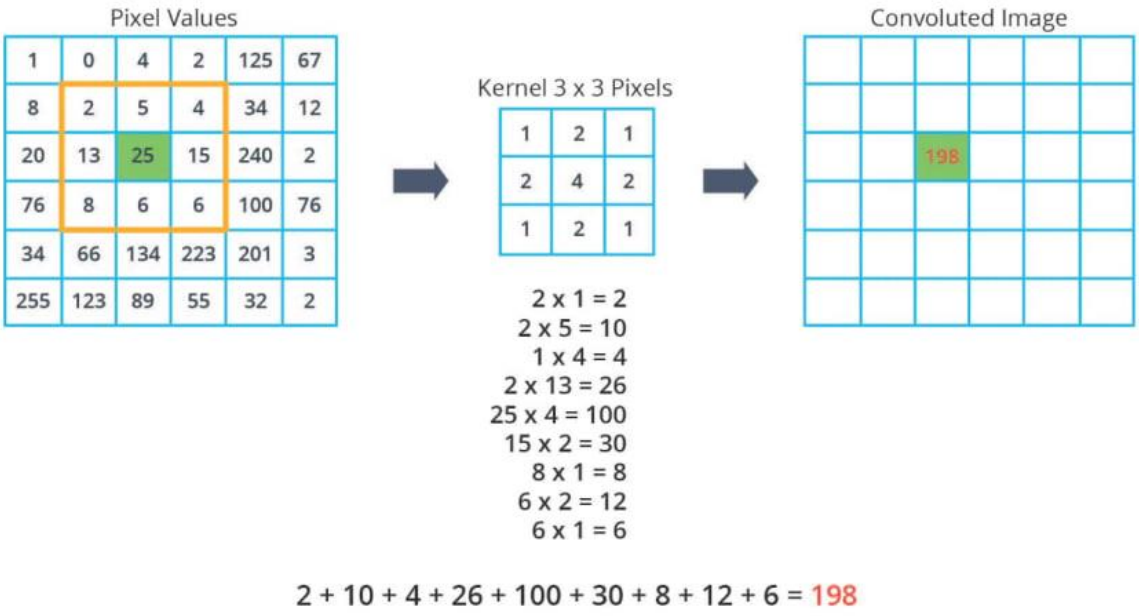
A neuron hálók példa adatok feldolgozásából tanulnak. Minden példa adatnak tartalmaznia kell az általa elvárt tökéletes kimenetet, ami alapján tudjuk majd számolni közte és a háló által generált kimenet közötti eltérést, ami a hiba vagy veszteség lesz. Miután a háló feldolgozta az egyszerre neki adott adatokat, az előre megszabott tanulási szabály és a feldolgozott példákból számolt veszteség alapján állít a súlyozásokon. Ezt a folyamatot többször elvégeztetjük a hálóval és ennek folyamán a háló mindig egyre jobban megpróbálja minimalizálni a kapott

veszteséget és az általa generált kimenettel egyre jobban közelebb kerülni a példákban szereplő tökéletes kimenethez. Ezt az önmagát ismétlő folyamatot valamilyen kritériumhoz kötve megállíthatjuk. Az ilyen jellegű tanítás során a hálók nem kapnak előre utasításokat vagy specifikus kritériumokat, ami alapján generálják a kimenetük, hanem a tanulási folyamat közben saját maguknak alakítanak ki azokat. Tehát ha például egy kutyát szeretnénk, hogy felismerjen egy háló képek alapján, nem mondjuk meg neki, hogy egy kutya hogy néz ki, csak képeket adunk neki, amik be vannak neki valamilyen módon „feliratozva" hogy kutya van-e a képen vagy nem. Ez a „felirat" lesz a kép osztálya.

3.2 Képfelismerés

Képfelismerésre a legelterjedtebb könyvtár mára a TensorFlow lett. Egy képfelismerésre létrehozott TensorFlow neuron hálóban általában nem hozunk létre manuálisan egyesével node-okat, hanem, layer-en ként definiáljuk a háló szerkezetét, majd a TensorFlow dinamikusan generálja ez alapján a node-okat. Minden layer egy bizonyos lépésért felel a kép lebontásában majd az így kapott adatok összegzésében vagy transzformálásában. Egy képfelismerésre létrehozott neuron háló bemenete standard kell, hogy legyen, ami annyit tesz, hogy minden bemeneti kép felbontása, aspektusa, valamint színcsatornáinak száma meg kell, hogy egyezzen. Ezért vagy meg kell bizonyosodnunk róla, hogy a használni kívánt dataset ennek megfelel, vagy pedig erről gondoskodó metódusokat kell írnunk. Pár szimpla transzformációhoz, mint az átméretezéshez, találhatunk a TensorFlow könyvtáron belül külön erre hivatott layer-eket amik valójában nem tanítható részei a neuron hálóknak, de a kezdeti standardizálást könnyítik. Ezek hátránya, hogy minden egyes adatbeolvasás esetén ideiglenesen normalizálódik csak az adat, tehát minden egyes betöltésnél ezt el kell végezni, még akkor is, ha ugyanazon tanítási folyamat alatt többször ugyanazt a képet kell betölteni, mivel a normalizált verzió nem kerül mentésre. Előnye pedig az, hogy így megmarad az összes eredeti adatunk, eredeti felbontásukkal, színcsatornáikkal és aspektusukkal anélkül, hogy egy normalizált verziót kellene eltárolnunk véglegesen a tárhelyen. A tanítható layer-ek általában konvolúciókat végeznek más-más paraméterekkel. A konvolúció képek esetén mikor veszünk egy általunk előre kialakított kernelt, mint például egy 3x3-as négyzetet, majd minden mezőjére meghatározunk egy műveletet, ezután pedig a kernelt ráhelyezzük egyesével a bemeneti kép minden pixelére, és kiértékeljük a kernelbe eső pixeleket a kernel adott helyén lévő műveletekkel, és végül az így kapott

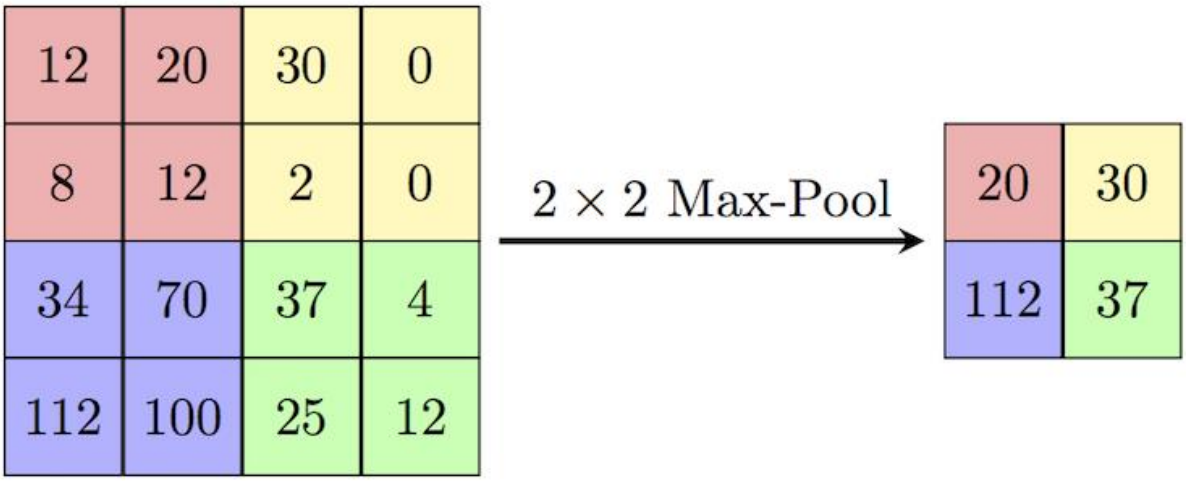
értékeket összeadjuk és ez lesz az új értéke az eredetileg kiválasztott pixelnek. Fontos megemlíteni, hogy konvolúciók során a kép által elfoglalt tárhely általában lényegesen növekszik.



3-2. ábra: egy konvolúciós layer működésének egyszerű szemléltetése

A fenti 3-2. ábrán bal oldalt található az eredeti bemeneti kép, középütt a kernel és jobb oldalt a kimeneti kép. Középütt látható a levezetett számítása az eredeti képen 25-ös értékű pixelnek. Továbbá gyakran alkalmazottak még a pooling layer-ek is, amik általában egy kisebb dimenziókkal rendelkező kimenetet adnak, mint a bemenetük. Ezek a képből konvolúciók által kiszedett értékes adatokat priorizálják és a kevésbé hasznos pixeleket dobják el. Ez a rengeteg felesleges művelet elkerülése miatt hasznos, mivel egy sok layer-el rendelkező neuron háló rengeteg műveletet végez el és konvolúciós layer-ek akár több dimenziós kimenettel rendelkezhetnek, mint a bemenetük, ezáltal akár nagyságrendekkel növelve az átmenetileg tárolni szükséges adatokat és ezek számát célszerű mielőbb csökkenteni. Egy példa erre, ha egy konvolúciós layer 10x10 pixeles képből csinál 20x20-as képet, majd ha ugyanazokkal a paraméterekkel rakunk erre még egy konvolúciós layert, akkor ez 40x40 pixel lesz és így sokszorozódik a kimeneti kép mérete, ha nincs rajta valamilyenfajta polling layer.

Polling layerekből több féle létezik, a lényegük annyi, hogy az adatok mennyiségét csökkentsük úgy, hogy a számunkra fontos adatokból minél többet megőrizzünk.



3-3. ábra: a 2x2-es Max-Pooling szemléltetése

A fenti 3-3. ábrán látható egy 2x2-es max-pooling layer működése. Itt ráillesztjük a kimenet pixeleit a bemeneti képre, és a maximum értéket vesszük minden kimeneti pixelre a ráeső bemeneti pixelek közül. A modell a tanulás folyamán úgy fogja kialakítani a paramétereit a konvolúciós layereknek, hogy a pooling layerekkel a számára legfontosabb adatokat megőrizze.

4. A modell kiválasztása

A neurál háló modelljére két alternatívát próbáltam ki, majd elemeztem ezek előnyeit és hátrányait. Az egyik egy szimpla pár layeres szekvenciális modell, míg a másik a Google által fejlesztett és előre betanított MobileNet modell saját dataset-emre való modifikálása.

4.1 Szimpla szekvenciális modell előnyei és hátrányai

```
model = Sequential([
    layers.experimental.preprocessing.Rescaling(1./255, input_shape=(IMG_SIZE, IMG_SIZE, 3)),
    layers.Conv2D(FILTERS, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(FILTERS * 2, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(FILTERS * 4, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(FILTERS * 8, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(FILTERS * 16, activation='relu'),
    layers.Dense(num_classes)
])
```

4-1. ábra: a saját szimpla szekvenciális modellem felépítése

A fenti 4-1. ábrán látható, az egyszerű szekvenciális modellem. Az első layer egy standardizálásért felelős layer, ami a képet az IMG_SIZE*IMG_SIZE felbontásra állítja. Ez azért volt hasznos, mert már eleinte látszott, hogy rendkívül sok memóriát igényel a tanítási folyamat ezzel a modellel, és ezzel tudtam könnyedén tesztelni más-más felbontásokat. Minél kisebbre állítom ezt, annál kevesebb a memória igénye, viszont általában a 10 epoc után elért becslési pontossága annál rosszabb. Ezután következik felváltva 4 konvolúciós valamint 4 max-polling layer majd pedig jön egy flatten majd 2 dense layer. A konvolúciós, valamint az első dense layerben a FILTERS változó segítségével tudtam könnyen nagyságrendekkel állítani a filterek méretét. A saját szimpla szekvenciális modellem fő előnye számomra a könnyű alakíthatóság, valamint a valamivel nagyobb becslési pontosság. A fő hátrányai pedig a MobileNet-hez képest rendkívül nagy erőforrás igénye, lassabb tanulási fázisa, valamint jóval nagyobb exportált modell mérete. Mellékes haszna viszont, hogy mivel az egész modellt könnyedén lehet modifikálni és átlátni, így nagyon jó volt a TensorFlow egyes metódusainak a tesztelésére.

4.2 MobileNet modell előnyei és hátrányai

A MobileNet modell előre be van tanítva egyszerű dolgokra, mint például kocsik, fák, kutyák, macskák felismerésére. Ebből adódóan, itt egy más megközelítéssel kell a modellt kialakítani, mivel itt nem alapjaiból építem fel a modellt, hanem importálom a már betanított MobileNet modellt, amit aztán modifikálok a saját dataset-emre hogy „tovább taníthassam" a saját felhasználásomra. Ez annyit tesz, hogy az utolsó pár adat sűrítésért és kiértékelésért felelős layert inaktiválni kell, vagy el kell őket dobni, és a helyükre bevezetni a saját layer-jeimet amik ugyanezt a célt szolgálják, csak a saját felhasználásomra igazítva azt. Miután ez megvan pár epoch tanítás után a súlyozások javarészt korrigálódnak az általam használt dataset osztályai becslésére.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

4-2. ábra: a MobileNet modell felépítése illetve a layerek ki- és bemeneti dimenziói

A fenti 4-2. ábrán láthatóak a MobileNet modell layer-jei, valamint dimenziói. Mint látszik, a pixel szám folyamatosan csökken, de a 3. dimenziójuk mérete folyamatosan bővül, valamint a filterek mérete folyamatosan nől. Ezekből csak az utolsó 5 az, ami nem kell, tehát azok helyére kell felvenni a saját kiértékelő layer-jeimet. A MobileNet modell számomra nyújtott fő előnyei a rendkívül jó optimalizáltsága volt, ami lehetővé tette, hogy akár relatíve gyenge Android

eszközökön is tudjon gyorsan futni a kiexportált modellje, valamint meglehetősen gyorsan fut le rajta tanítás során egy-egy epoc. Mivel a modell előre be van tanítva elég sok egyszerű dolog felismerésére, így valószínűleg ennek is köszönhető, hogy tanítás során már az első pár epoch is nagyon jó eredményeket produkált, de feltehető, hogy ebből kifolyólag a modellnek lesz némi elfogultsága bizonyos minták illetve alakzatokra, amikre eredetileg figyelt a modellt. Hátránya számomra, hogy nehezebben átlátható a modell kialakítása, valamint esetlegesen, ha még több layert szeretnék megváltoztatni a modell belsejében, azt számottevően bonyolultabb megvalósítani, mint a saját szimpla szekvenciális modellem esetében.

4.3 Választott modell

A modellek előnyeit és hátrányait figyelembe véve az Adnroid applikációt a MobileNet modell segítségével valósítom meg, azonban a saját szimpla szekvenciális modellemet is használom, hogy dataset betöltési technikákat és hasonló általánosan használt beépített TensorFlow metódusokat teszteljek.

5. Tanító dataset előkészítése

Egy TensorFlow modell tanításához egy dataset-nek meg kell felelnie pár előfeltételnek, vagy úgy kell előkészítenünk a belőle felhasznált adatot, hogy a modell számára szolgáltatott input már megfeleljen ezeknek. Ezen feltételek például: standard kép méret és ebből következően képarány, standard kép formátum valamint minden kép valamilyen formában legyen osztályozva. Értelem szerűen, ha a képek felbontásában, képarányában, vagy színskálájában van különbség, akkor ugyanazon layerek más-más dimenziójú kimeneti képeket adnának, ami összeférhetetlen lesz.

5.1 Dataset növelés

Egy modellt minél több adattal tanítunk, annál kisebb az esélye, hogy talál valami olyan egyezést a képek között, ami számunkra valójában irreleváns. Például, ha van sok képünk olyan almafa levelekről amiknél, mindnél a háttérben van valami piros, akkor jelentős esélye, van, hogy ezt a tanítás folyamán a modell észre veszi. Majd ha egy olyan képet adunk be később becslésre a betanított modellnek, amin van a háttérben valami piros, akkor nagy valószínűséggel almafa levélnek becsülheti azt, mivel a modell ténylegesen csak közös vonásokat hivatott kiszűrni a képek között és ez alapján fog dönteni. A datasetek előkészítése során egy gyakran használt szokás a képek módosítása tükrözéssel és/vagy elforgatással, sőt akár színskála változtatással, ha csak az alakzatokra akarunk tanítani egy modellt. Ennek a lényege, hogy a tanításhoz felhasználható adat mennyisége megsokszorozható anélkül, hogy a modell hajalmos legyen kizárólagosan egy pár specifikus képet keresni majd a használat során. Az általam felhasznált datasetben azonban előre megtalálhatóak a képek tükrözött illetve elforgatott verziói, így ezt a lépést ki kell hagyni, mert csak redundáns adatokat szülne. Levelek felismerése esetén a szín fontos lehet, tehát a színskálát nem szabad változtatni itt. Mivel a rendelkezésemre álló dataset mérete csak körülbelül 88000 kép, amiben benne van az összes elforgatott és tükrözött verzió is, így a szimplán fekete-fehér képek alapján nem ideális. Szimplán tesztelésre viszont hasznos lehet, mivel a neurál hálónak ténylegesen csak az adatok harmadát kell feldolgoznia, ezzel drasztikusan csökkentve a tanítási folyamat alatt a memória használatot, valamint a tanítás idejét és egy elégséges indikátora lehet annak, hogy az épp tesztelt paraméterekkel hogy teljesítene a modell, ha minden színcsatornát feldolgoznánk. Fontos, hogy egy képből ne

készítsünk túl sok variációt, mert ekkor a modell azt a specifikus képet fogja keresni, és túlzott elfogultsága lesz azonos típusú képekkel szemben.

5.2 Standard kép méret és formátum

Mivel a dataset-ben található képek mérete mind 256*256 és mind JPG ezért ez már felhasználható a modell tanítására és nem kell vele különösebben foglalkozni, amíg nem használunk máshonnan képeket a modell tanítására. A 256*256 felbontás viszont relatíve nagynak számít itt. Egy egyszerű neurál hálónak nem feltétlenül kell ilyen nagy felbontás, mivel a képen nem apró részletekre vagyunk kíváncsiak, hanem főleg alakzatokra és színekre.

5.3 Szegmentálás

A TensorFlow modellek tanítása 2 fázisban zajlódik. Az első az maga a tanítási fázis, majd utána jön egy validálási fázis. Egy epoch-nak nevezzük, mikor ez a 2 fázis lezajlik egyszer a tanítási folyamat során. A tanítási fázisban próbál ténylegesen tanulni a modell a súlyozásainak a változtatásaival. A validálási fázisban pedig ellenőrzést végez egy kisebb steril mintán, amit nem használ fel tanulásra, csak a tanítási fázisban változtatott súlyozások korrigálására, ezáltal egy pontosabb értéket kapva, hogy mi az epoch végén a modell vesztesége illetve pontossága. A két fázis részére ketté kell választani a dataset-et tanítási és validálási részre olyan arányban, ahogy azt szeretnénk majd használni. Ezt általában a tanításra írt programunkban vagy scriptünkben szoktuk megtenni, de az általam használt dataset előre szét van választva erre körülbelül 80%-20% arányban, ami az átlagosan használt arány szokott lenni, valamint ezeken kívül van egy pár darab kép, ami manuális tesztelésre van elkülönítve. Az előre manuálisan szétválasztott tanítási és validálási dataset előnye, és egyben hátránya, hogy a validálásra elkülönített dataset részre soha nem fog lefutni a tanítási fázis. Ez azt jelenti, hogy így 20%-al kevesebb adatunk van tanításra. Viszont azt is jelenti, hogy mivel ezeket a képeket soha nem látja tanulás közben, így ha elfogultsága lenne a tanítási dataset rész pár specifikus képére, az egyből látszik az által, hogy az egymás utáni epoch-ok egyre jobb becslési százalékot produkálnak a tanítási folyamat során, és egyre rosszabbat vagy stagnáló becslési százalékot érnek el a validálási folyamat során.

5.4 Osztályzás

A szétválasztott tanítási illetve validálási mappán belül minden osztálynak saját mappája van kialakítva és ezen belül találhatóak a hozzá tartozó képek. A képfájlok nevei random generáltak, tehát a modell tanítása során a mappák nevei alapján kell majd hivatkoznunk a képek osztályára, amit a TensorFlow egyik saját metódusa el tud végezni.

6. TensorFlow modell

A TensorFlow modell tanításáért egy python script felel. Ez a python-nak hála lehet egészen nagy objektum orientált program akár, de az én esetemben egy egyszerű és rövid script volt a legpraktikusabb. Ennek számos oka van. Például, hogy a script számos modifikációt és akár teljes újra írást kapott, valamint a modell egyszerűségéből adódóan nem egy hosszú programról lett volna szó, ha mégis egy külön álló programot írok rá. Ezen felül a modell tanítása során rendkívül sok TensorFlow metódus egyszerű kipróbálására adott ez lehetőséget, valamint a tanítást csak egyszer kell elvégezni és a tanítási folyamat elégségesen rövid ahhoz, hogy esteleges korrigálásokért az egész folyamatot újra lefuttassuk.

6.1 Képek betöltése

Egy TensorFlow modell tanításakor az első áthidalandó probléma az adatok betöltése. Mivel a tanítás folyamán egy képből legenerált ideiglenes adat az eredeti kép méretének hatványozottja ezért egy jelentős limitáló faktor a folyamatot végző számítógép memóriájának mérete. A leggyorsabb tanítási folyamatot persze az eredményezi, ha lehetőségünk van az egész dataset-et betölteni egyszerre, ám ez még az általam felhasznált 1.5 GB-os dataset esetén is több mint 20 GB memóriát igényel egy szimpla 5-6 layer-es szekvenciális neuron háló esetén is. Ez azért van, mert az első pár layer konvolúciós layereket tartalmaz és minden számukra adott értékből több értéket generálhatnak és ezek mennyiségét a polling layerek csökkentik le, ám az első pár layer során általában még nem olyan mértékben, mint amilyen mértékben növeljük őket konvolúciós layerekkel. Ezzel szemben, ha csak az épp felhasználni kívánt batch-eket töltjük be, a felhasznált memória mennyisége az előző példa töredékére csökken, az esetemben például kevesebb, mint 2 GB-ra. Ennek a jelentős memória spórolásnak viszont hátránya is van. Ha nem töltünk be mindent egyszerre, akkor egyrészt rengeteg idő megy el azzal, hogy a betöltésért felelős algoritmus a memóriát kezelje, azt beossza, illetve ürítse a tanítás alatt folyamatosan. Ez esetben a limitáló faktor nem csak a videó kártya gyorsasága lesz, hanem az a sebesség, amivel minden szükséges batch-et be tudjuk tölteni buffer-be tanításra, illetve fel tudjuk szabadítani a memóriát a már nem használtaktól. Ezt a folyamatot a TensorFlow saját metódusa végzi, de érdemes számon tartani, hogy megéri-e ezt a módszert használni, mert ha rendelkezésre áll elég memória a teljes dataset betöltésére, akkor a tanítási folyamat gyorsasága akár a sokszorosa is

lehet, viszont az erre szükséges memória nem lineárisan nő a tanító dataset méretével. A betöltés-re való felkészítést a TensorFlow ImageDataGenerator metódus végzi, ami szükség esetén átméretezi a képet a standard 256*256-ra, valamint 64-essével batch-ekbe rendezi őket, valamint összekeveri a képek sorrendjét, hogy a modell ne csak a sorrendet tanulja be. Az esetleges átméretezésre azért lehet szükség, mert ha manuálisan felvennék új képeket a dataset-be akkor azokat esetlegesen standardizálni kell, hogy az elfogadható legyen a tanításra. Az előkészített batchek két változón elérhetőek, amiken keresztül a tanítás során hivatkozok rájuk. Ezek a „train_batches” és a „valid_batches” változók lesznek, amiket aztán majd a TensorFlow tanító metódusa fog használni.

6.2 Callback pontok

Egy modell tanítása sok esetben rendkívül hosszú időt igénybe vehet, valamint a számítógép erőforrásait intenzíven igénybe veszi, szóval egy jó biztonsági lépés saját magunknak, ha minden epoch végén létrehozunk egy callback pontot, amire aztán vissza ugorhatunk további tanításra akár más módszerrel, mint azt az előtte történt epoch-ok alatt tettük. Ezzel rendkívül sok időt tudok megspórolni, valamint ez által ki tudtam próbálni, hogy milyen eredményt produkál a modell, ha kettő vagy akár több féle kis részletekben eltérő neuron hálóval tanítom a modellt. Mivel a modell felettébb egyszerű célt szolgál, így ez nem produkált lényeges különbséget a becslési pontosságban, viszont sokkal bonyolultabb és időigényesebb ezzel a módszerrel tanítani. Ezen okoknál fogva, tehát a callback-ek csak biztonsági mentés célt szolgáltak számomra. A megvalósítási módjuk pedig a következő: Egy checkpoint_path nevű változóra kijelölök egy mappa elérési útját, amibe majd mentem ezeket, majd tovább adom a ModelCheckpoint metódusnak, ami a callback pontok létrehozásáért és mentéséért felelős.. Argumentumként beállítom neki, hogy csak a modell súlyozásait kell mentenie, az épp tanításra betöltött dataset-et nem, mert felesleges az esetemben, mivel az tanító dataset-em statikus, azaz nem változik vagy gyarapszik, tehát ezek valamint az ezek által legenerálódott adatok redundánsak lennének. Valamint verbose változóval beállítom neki, hogy a tanítás alatt konzolban írja, ha épp mentett egyet. Ez utóbbi funkció hasznos, ha nem követem aktívan a tanítás menetét, hanem csak utólagosan átolvasom, hogy a tanítás menete hogy alakult, mert látszik, hogy mely pontokon mentett, illetve, ha probléma adódna fel ezzel kapcsolatosan, az is látszódik a konzolon.

6.3 MobileNet testreszabása

A modell utolsó pár layere a konvolúciók által kihozott információk összegzéséért valamint az eredetileg betanított osztályokra való becslésért felelnek. Mivel nekem csak a modell fő szerkezete kell nem pedig a teljes betanított modell, az utolsó 5 layert át írom nem taníthatóra, ezzel inaktívvá téve őket és az ezek előtti layer kimenetét átadom a saját Dense layeremnek. Ez teszi lehetővé, hogy alkalmazhassam a modell szerkezetét. Tehát a mobile nevű változóba betöltöm a MobileNet modellt, majd egy x nevű változóra kivezetem a modell hátulról számolt hatodik layer output-ját. Egy predictions néven létrehozom a saját utolsó Dense layerem aminek át van adva x mint bemenet, majd egy model nevű változóban létrehozok egy keret modellt ami az inputját átadja a betöltött MobileNet modellnek, és output-nak pedig a saját Dense layerem output-ját veszi. Tehát a bemeneti kép először bekerül a MobileNet modellbe, majd az utolsó layertől visszafele számolva 6. layer kimenetét átadjuk egy sima Dense layernek. Utóbbi Dense layer kimenet pedig a teljes modell kimenete lesz.

6.4 Modell tanítás

A modell tanítása előtt a compile metódussal hozzárendelem az Adam optimizer-t, valamint a CategoricalCrossentropy loss-t és a becslési pontosságot, mint metrikát. Ezek felelnek azért, hogy a modell milyen elven tanuljon majd, a becslési pontosság metrika pedig célszerű a tanítás monitorozásához, hogy lássuk körülbelül mikor célszerű abbahagyni. Ezután jön a tényleges tanítás. A modell fit metódusával elindítjuk a tanulást és hozzárendeljük a már előkészített tanító és validáló batcheket. Megadjuk neki továbbá az epoch-ok számát, ami azt jelenti, hogy hányszor fog végigmenni az egész dataset-en, ami jelen esetben 10 lesz, valamint megadjuk a verbose változóval, hogy a tanítás alatt konzolban legyen egy folyamat csíkunk minden epoch-ra és mellette jelenjen meg a statisztikája az adott epoch-nak, ami jelen esetben a loss és a becslési pontosság értéke lesz.

6.5 Tanítási statisztika

A tanítás mivel elég sokáig is eltarthat célszerű valami automatikus diagramot rajzoltatnunk a lényeges metrikákról, hogy könnyen észre vegyük, hol vannak azok a pontok ahol célszerű beavatkozni. A számomra lényeges metrikák itt a becslési pontosság és a loss volt, de mivel

minden epoch tanítási és validálási fázisának van saját értéke mind a két metrikára, így két külön diagramon rendeztem el a pontosságot és a loss-t, valamint diagramon ként két külön görbét húztam a már említett 2 fázis értékeinek.

6-1. ábra: a python segítségével generált statisztikai diagram a tanítási folyamatról

A fenti 6-1. ábrán látható a tanítás statisztikái epoch-on ként kiértékelve. A bal oldali részen láthatóak a tanítási és validálási fázisok pontosságai a függőleges tengelyen, és 1 jelzi a 100% pontosságot, 0.9 a 90% pontosságot. A jobb oldali részen pedig láthatóak a tanítási és validálási fázisok loss metrikái a függőleges tengelyen, ami minél kisebb, annál pontosabb modellt eredményez. A vízszintes tengely mindkét esetben az epoch számot jelöli ahol az értéket produkálta a tanítás során a modell, valamint mindkét részen a kék vonal jelöli a tanítási fázis, a sárga pedig a validálási fázis értékeit.

6.6 TensorFlow Lite modell

A tanítás során létrehozott modell már használható lenne asztali számítógépek számára, azonban egy Android-os eszköz erőforrásaihoz egy TensorFlow Lite modellé kell alakítanunk, hogy az előbb említett erőforrások ne legyenek túlzottan limitáló faktorok. Egy TensorFlow Lite modell sokkal kevesebb tárhelyet foglal, mint egy szokásos módszerrel kiexportált modell, valamint valamennyire le van egyszerűsítve így például egy szimpla Android alkalmazással

szinte bármilyen modern Android eszköz tudja használni és képest a modell becslés funkcióját használn. Egy converter névre meghívom a TFLiteConverter.from_keras_model-t és argumentumként adom neki a modellemet, amit már betanítottam, aztán meghívom a converter-nek a convert() metódusát és az így kapott lekonvertált modellt pedig kiíratom fájlba. Az így kapott fájl lesz a TensorFlow Lite modell, amit majd később felhasználhatok.

7. Android applikáció UI

Az Android applikáció két fő felületből áll, az elsődleges fő ablak, valamint, az élő kameraképet kiértékelő másodlagos ablak. Az applikáció az elsődleges ablakkal nyílik majd meg és innen a funkciók segítségével nyithatjuk meg az alapértelmezett kamera vagy galéria applikációt, ha csak egy statikus képet szeretnénk kiértékelni, vagy akár átnavigálhatunk a másodlagos ablakra, ahol pedig az élő kameraképet tudjuk kiértékelni.

7.1 Az elsődleges ablak

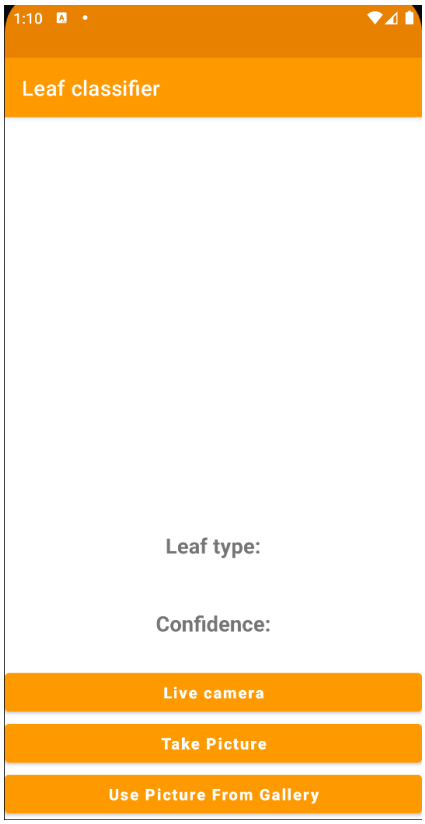
Az elsődleges ablakon RelativeLayout-ban nyolc elem van. A legfelső legnagyobb elem egy ImageView, ami addig fehér, mint a háttér, amíg az activity két fő metódusából egy képet nem kap. Ez alatt van négy TextView amik egymás aljához vannak csatolva, és ezek felülről haladva sorban: „Leaf type:” kiírás, egy üres kezdeti értékű „result” id-t birtokló, „Confidence:” kiírás és egy üres kezdeti értékű, „confidency” id-t birtokló. Az üres kezdeti értékűek felelnek majd a kapott kép osztályzásának eredményeinek kiíratásáért. A kijelző aljára pedig 3 gomb van elhelyezve, ezek pedig felülről haladva felelősek: az élő kameraképet kiértékelő activity-re váltásért, kamerával egy fotó készítéséért, amit aztán, mint input tud használni ez az ablak kiértékelésre, valamint galériából egy kép lekéréséért, amit szintén inputként tud használni ez az ablak kiértékelésre. A gombokon kívül minden elem a benne lévő érték méretéhez igazodik, és középre igazított, valamint egymás aljához vannak csatolva, hogy ne takarják ki egymást, ha más felbontáson nyitjuk meg az alkalmazást. A gombok az ablak aljától indulnak, és egymás tetejéhez vannak csatolva, és olyan szélesek amilyen széles az ablak, viszont a felirat bennük középre van igazítva.

7.2 Az élő kameraképet kiértékelő ablak

Az élő kameraképet kiértékelő ablakon hat elem van egy RelativeLayout-ban. Mivel a struktúrája szinte azonos az elsődleges ablakéval így a formázásért felelős paraméterek ugyanazok lesznek. A legfelső elem itt egy PreviewView, ami az Androidx.camera.view könyvtárból lett importálva. Ez a felület eleinte fekete, de amint az eszköz hátlapi kameráját elindítja a program, annak az előnézeti képe fog itt megjelenni. Ez alatt itt is szintén négy

TextView van, mint az elsődleges ablakon, ugyanazokkal a paraméterekkel és feliratokkal. Ezek alatt pedig van egy vissza gomb, amivel az elsődleges oldalra tudunk vissza navigálni.

8. Android applikáció elsődleges ablak



8-1. ábra: elsődleges ablak kinézete az applikáció indítását követően

Ez az ablak felelős az alapértelmezett kamera vagy galéria applikációt meghívni, hogy egy képet kapjon tőlük, amit aztán a TensorFlow lite modell kiértékel. Az applikáció indítás utáni állapotát látható a 8-1. ábrán fentebb. Mint az látszik is, mivel még nem adtunk az applikációnak egy bemeneti képet, így a kép helye fehér, mint a háttér és nem jelenik meg addig a „Leaf type" és a „Confidence" feliart alatt semmi. Ez természetesen változni fog miután kapott már egy képet és utána csak a már megjelenített adatokat, és képet frissítjük.

8.1 onCreate metódus

Az activity létrehozásakor inicializáljuk a felhasználói felület azon elemeit, amiket vagy frissíteni akarunk, majd értékekkel vagy hozzá akarjuk rendelni metódusokhoz, mit például a gombokat. A kamera gombra kattintás esetén először le kell ellenőriznünk, hogy az applikáció kapott e már hozzáférést az eszköz kamerájához az Android-tól. Ezt, ha még nem kapta meg akkor 100-as request kóddal kérjük el az android-tól. Ha már kapott hozzáférést, akkor egy új Intent-el megnyitja az eszköz alapértelmezett kameráját, majd attól vár 3-as request kóddal eredményt, amit majd később fel tud használni az elsődleges ablak, mint bemeneti kép. Az élő

kamerakép gombja is először lekéri, hogy van-e már hozzáférése az applikációnak az eszköz kamerájához és azonos módon kér hozzáférést, ha nincs. Ha már kapott, akkor viszont szimplán megnyitja az elő kameraképet kiértékelő ablakot. A galéria gombja pedig egy új Intent-el megnyitja az eszköz alapértelmezett galéria applikációját egy képért, és erről 1-es request kóddal vár eredményt.

8.2 classifyImage metódus

Ez a metódus felel azért, hogy egy adott képet ezen az ablakon kiértékeljünk a már betanított TensorFlow Lite modell segítségével. Tehát egy Bitmap-et vár paraméterül a metódus. A metódus először inicializálja a modellt, majd létrehoz egy új fix méretű és adattípusú TensorBuffer-t és egy fix méretű ByteBuffer-t. Ezután létrehozunk egy integer tömböt, ami hivatott majd a képet fogja magába foglalni. Utóbbi azért szükséges, mert a modellünk lebegő pontos számértékeket szeretne a pixelek értékeire kapni, mint bemenet és ezek az értékek 0-tól 1-ig terjedhetnek. Tehát ha például egy pixel RGB értékén a vörös színcsatorna értéke 255, akkor az az 1, ha 127, akkor pedig 0.5 mikor átkonvertáljuk ezeket a modell bemeneti értékeire. Ez volt eddig az inicializálása a szükséges változóknak, ezután pedig lekérjük a paraméter kapott kép pixel értékeit az előbb említett integer tömbbe. Innentől sok fontos átalakítást fog végezni a metódus, ezeknek a főbb sorrendje: Bitmap-ből Integer tömb, Integer tömbből ByteBuffer, ByteBuffer-ből TensorBuffer. Ez mind azért szükséges, mert a modell csak TensorBuffer-t képes kiértékelni. Tehát ezután egy dupla for cikluson belül pixelenként betöltjük a ByteBuffer-be a pixel RGB értékeit lebegő pontos számérték ként. Miután ezzel kész a metódus, a ByteBuffer-t betöltjük a TensorBuffer-be és az így kapott TensorBuffer már kiértékelhető a modellünk által. A modell saját kimeneti érték típusával létrehozunk egy új objektumot, amire kiértékeljük a TensorBuffer-t a modell segítségével. Egy újonnan létrehozott TensorBuffere pedig lekérjük ennek az objektumnak az értékeit. Az így kapott TensorBuffert pedig átalakítjuk egy lebegő pontos számértékű tömbbé és ez lesz a modell becsült értéke minden betanult osztályra. Itt egy for ciklussal végig megyünk rajta, hogy megtaláljuk, melyik indexen van a legnagyobb érték, valamint a legnagyobb értéket saját magát is kimentjük. A maximum értéket átalakítjuk emberi szemmel könnyen olvasható százalékos formába, mivel ez lesz a modell által adott becslés magabiztossága százalékos formában. Azonban mivel az osztályok nevei nincsenek a modellen belül elmentve, így a tanítás szerinti sorrendjük szerint

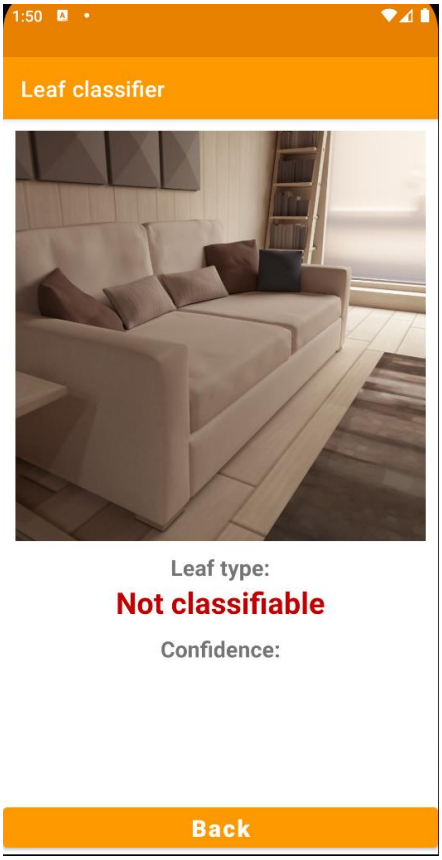
felvesszük ezeket az osztályneveket egy String tömbbe. Ezek után megnézzük, hogy az így kapott maximum becslési nagyobb-e 92%-nál és ha nem akkor úgy vesszük, hogy nem tudtunk elég biztos becslést adni. Tehát „Not classifiable” feliratot íratunk ki, nem pedig a levéltípust, valamint null-ra állítjuk a felhasználói felületen a becslési százalékot, mert ha nem első alkalommal futtatjuk ezt a metódust, és már valamire kaptunk 92%-nál nagyobb értéket akkor már van kiírva egy százalék, amit célszerű eltüntetni. Ha viszont 92%-nál nagyobb értéket kapunk, akkor kiírjuk a levél típusát és a hozzátartozó betegség típusát, valamint kiíratjuk a felhasználói felületre az ehhez tartozó becslési százalékot is.

8.3 onActivityResult metódus

Ez a metódus felel azért, hogy mikor megnyitjuk vagy a kamerát vagy a galériát, hogy egy képet készítsünk vagy kérjünk, akkor ténylegesen átvegyük azt és fel is használjuk. Tehát ha például megnyitjuk a galériát, és egy képet sikeresen kiválasztunk, akkor azt ez a metódus átveszi, lekonvertálja felhasználható méretűre és átadja azt a classifyImage metódusnak. A metódus paraméterei a request kód, amivel elindítottuk a kamerát vagy a galériát, a visszatérési értéke ennek, valamint egy Intent amin keresztül a képet fogják átadni. Először ellenőrizzük, hogy a visszatérési érték rendben van-e, azaz hogy sikeresen lefutott a kamera vagy a galéria rész. Ha igen, akkor a request kódon keresztül ellenőrizzük, hogy a kamera vagy a galéria rész futott le. Itt pedig, ha a kamera részt érzékeljük, hogy lefutott, akkor a kapott Intent-ben lévő képet kikérjük egy Bitmap-re, megnézzük, hogy a kép melyik tengelye kisebb, és ennek a hosszúsága szerint kivágunk belőle egy kisebb négyzetet, mivel a modell 1:1 képarányú képeket szeretne, mint input. Az így kapott képet megjelenítjük a felhasználói felületen az ImageView-ban. Ez a kép viszont még lehet akár nagyobb is vagy kisebb, mint a modell bemeneti paraméterei, tehát csak szimplán átméretezzük ezt a képet, hogy megfeleljen a modellnek, mint bemeneti kép. Az átméretezett képet pedig átadjuk a classifyImage metódusnak. Ha viszont a request kódból nem az látszik, hogy a kamera rész futott le, akkor az Intent-ben kapott képet egy Uri objektumra kérjük ki. Erre azért van szükség, mert rengeteg galéria applikáció elérhető az interneten, és pár gyártó már nem is az Android saját galéria applikációját használja, mint alapértelmezett, és így van esély rá, hogy olyan applikációt futtatunk, ami nem csak képeket, de akár videókat is átadhat nekünk. Azonban videót nem akarunk használni, csak képeket a lehetséges hibák elkerülése végett és az Uri objektumok adattípusát könnyen le lehet kérni. Tehát lekérjük, hogy

a kapott Uri objektumunk típusa valamiféle kép-e és ha igen, akkor egy Bitmap-be kérjük át belőle a képet és ezt jelenítjük meg a felhasználói felületen az ImgaeView-n keresztül. A képet átméretezzük a szükséges méretre, hogy lehessen használni, mint bemeneti érték a modell számára, majd átadjuk a classifyImage metódusnak. Ellenben, ha nem kép a típusa az Uri objektumnak, akkor egy kis hibaüzenettel térünk vissza, amin a „Not usable file format” olvasható.

9. Android applikáció élő kamerakép ablak



9-1. ábra: az élő kamerakép ablak kinézete

Ezt az ablakot akkor nyitjuk meg, ha az elsődleges ablakon rányomunk az ide vezető gombra. Ez az ablak azért felel, hogy a kamera előnézetet használva valós időben kiértékeljük, amit épp lát az eszköz kamerája. Itt az onCreate metódus hasonlóan az elsődleges ablak onCreate metódusához először inicializálja és létrehozza a kapcsolatokat a szükséges felhasználói felület elemekhez, azzal a kivétellel, hogy itt csak egy vissza gomb található, ami bezárja ezt az ablakot és mivel az elsődleges ablakot nem zártuk be arra, fog átugrani. Ezen felül meghívja a startCamera metódust, amivel elindul az élő kamerakép folyamatos kiértékelése. Ennek a főbb lépései nagy vonalakban: elindítjuk a kamerát és hozzácsatoljuk a felhasználó felületen a previewView elemhez, valamint egy kép analizáló objektumhoz, aminek a segítségével aztán képkockán ként hozzáférünk az épp aktuális kameraképhez, és a modellünk segítségével kiértékeljük az épp aktuális képkockát, majd az így kapott eredményt kiíratjuk a felhasználói felületre. Ez az ablak már jelentősen függ pár Androidx könyvtártól. Erre az ablakra való navigálás utáni állapotot látható a 9-1. ábrán fentebb. Mint az látszik is, nem sokban tér el az elsődleges ablak kinézetétől a felhasználó számára.

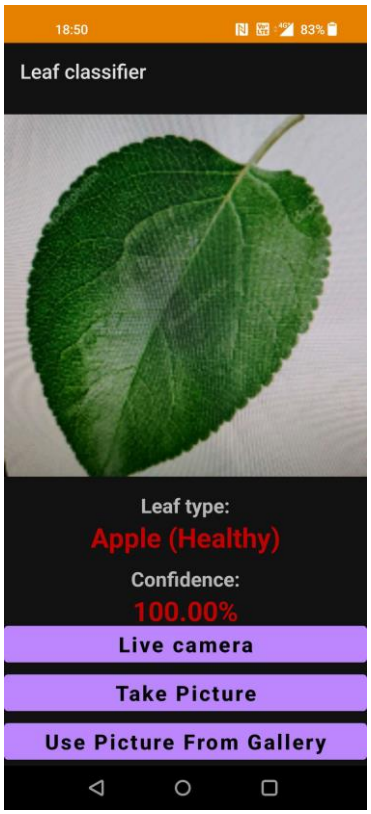
9.1 startCamera és bindPreview metódusok

A startCamera metódusban létrehozunk egy ListenableFuture<ProcessCameraProvider> típusú objektumot és hozzárendeljük a kamerát, majd ebből kiszedünk egy ProcessCameraProvider típusú objektumot, amit tovább adunk a bindPreview metódusnak. A bindPreview metóduson belül pedig a biztonság kedvéért lecsatlakoztatunk minden előző folyamatot a kameráról, már ha volt épp aktív rajta, majd kiválasztjuk a készülék hátsó kameráját. Létrehozunk egy új ImageAnalysis objektumot, amin keresztül majd képkockánként hozzá tudunk férni a kamera képéhez. Létrehozunk egy előnézet objektumot, és hozzácsatoljuk a felhasználói felület previewView eleméhez. Ezek után a kamerához csatoljuk sorban a kamera kiválasztót, az előnézetet és az ImageAnalysis objektumot.

9.2 analyze és classifyImage metódusok

Mivel létrehoztunk egy ImageAnalysis objektumot és hozzácsatoltuk a kamerához, így mostmár hozzáférhetünk a kamera épp aktuális képeihez, viszont ehhez a beépített analyze metódust kell felül írnunk. Ez a metódus szolgáltat nekünk egy ImageProxy objektumot, ám az ezzel járó sok szükséges átalakítás miatt egyszerűbb, ha csak a felhasználói felületen a previewView elemtől kérjük le a Bitmap-et és az ImageProxy-t bezárjuk, hogy tovább tudjon futni a program, ha egyszer már lefutott az analyze metódus. Majd a már ismert módon átméretezzük a képet és átadjuk a classifyImage metódusnak. A classifyImage metódus itt ugyanúgy működik, mint az elsődleges ablakon, viszont azok a sorok, amik a felhasználói felületet frissítenék adattal, be kell, kerüljenek egy runOnUiThread metódusba mivel nem csak egyszer lesznek meghívva, hanem folyamatosan minden kiértékelt képkockára.

10. Elért eredmények, továbbfejlesztési lehetőségek



10-1. ábra: az elsődleges ablak rendeltetésszerű működése

A feladat során elkészült egy python script, amivel pár paraméter megváltoztatásával lehet azonos struktúrájú dataset-ekre tanítani egy MobileNet alapú modellt. A script minden epocra ment egy „callback” pontot, amit biztonsági mentéseknek tudunk használni a tanítási folyamat során az epoch végi állapotokról. Miután a script végez a tanítással kimenti a tanított modellt TensorFlow Lite formátumban, amit aztán tudunk használni Android applikációkkal is. Ezen felül elkészült egy Android applikáció, ami az előbb említett TensorFlow Lite modell segítségével képes tetszőleges képeket kielemezni, hogy a modell szerint melyik általa tanult osztályba tartozik és mekkora biztossággal. Erre az applikáció több módszert tud alkalmazni, tudja alkalmazni, az eszköz alapértelmezett kamera vagy galéria applikációját, hogy egy darab statikus képet értékeljen ki, vagy akár tudja az eszköz kameráját is használni, hogy az előnézeti képét folyamatosan kiértékeli. A fenti 10-1. ábrán látható az elsődleges ablak rendeltetésszerű működése. Mind az applikációt, mind magát a neurál háló modelljét számos szempontból tovább lehet fejleszteni.

10.1 Modell továbbfejlesztési lehetőségei

A modell jelenleg körülbelül 98.5%-os pontossággal tudja megállapítani a bemeneti képek osztályát, ezt természetesen egy nagyobb dataset-el való tanítás tudná könnyedén javítani, vagy ha a jelenlegi MobileNet alapú modell összes súlyozását eldobnánk és kizárólagosan a saját dataset elemeivel, jóval több epoch-al újra tanítanánk az is feltehetően javíthat a pontosságon. Ezen felül a modell jelenleg nem próbálja kiszűrni, hogy a bemeneti kép esetlegesen nem a betanított osztályok egyikébe tartozik, tehát minden képre nagy magabiztossággal fogja tippelni, hogy valamelyik általa ismert osztályba tartozik bármi is legyen a képen. Erre egy egyszerű megoldás lenne, ha rengeteg teljesen irreleváns tartalommal rendelkező, illetve más, a modell számára ismeretlen falevél típusok képet egy új osztály alatt beraknánk a tanulási folyamatba egy pár epoch erejéig legalább. Az applikációban pedig, ha ezt az osztályt kapjuk, mint becsült eredmény, akkor tudjuk, hogy a kép számunkra nem beazonosítható. Ez nem egy tökéletes módszer, de jó kiindulási pont lehet.

10.2 Applikáció továbbfejlesztési lehetőségei

Az applikációt főleg funkcionalitás téren lehetne fejleszteni. A jelenlegi funkciókban pedig maga a felhasználói felületet lehet tovább fejleszteni, valamint az élő kamerakép kiértékelésnél lehet egy olyan felületet írni, ami kezelné az eszköz kamerájának funkcióit is, mint például a fókuszpont állítását, kép közelítését, fényességének állítását, stb. Ez segíthetne jobb eredményeket produkálni a modell számára is, mert tisztább képeket tudunk átadni, ha ezekre lehetőséget nyújtunk a felhasználónak.

11. Irodalomjegyzék

[1] PyCharm: https://www.jetbrains.com/pycharm/ Utolsó megtekintés: 2022.12.10.

[2] Android Studio: https://developer.android.com/studio Utolsó megtekintés: 2022.12.10.

[3] TensorFlow: https://www.tensorflow.org/ Utolsó megtekintés: 2022.12.10.

[4] New Plant Diseases Dataset: https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset Utolsó megtekintés: 2022.12.10.

[5] W3schools Python tutorial: https://www.w3schools.com/python/ Utolsó megtekintés: 2022.12.10.

[6] Androidx.camera könyvtár dokumentáció: https://developer.android.com/jetpack/androidx/releases/camera Utolsó megtekintés: 2022.12.10.

[7] TensorFlow Basic Image classification tutorial: https://www.tensorflow.org/tutorials/keras/classification Utolsó megtekintés: 2022.12.10.

[8] TensorFlow Lite tutorial: https://www.tensorflow.org/lite/android/quickstart Utolsó megtekintés: 2022.12.10.

[9] Build your first android app in java from Google: https://developer.android.com/codelabs/build-your-first-android-app Utolsó megtekintés: 2022.12.10.

[10] Max-polling wikipedia: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling Utolsó megtekintés: 2022.12.10.

[11] Sandeep Balachandran: Machine Learning – Convolutional Neural Networks: https://dev.to/sandeepbalachandran/machine-learning-convolutional-neural-networks-cnn-3npd Utolsó megtekintés: 2022.12.10.

[12] Mesterséges neurális hálózat wikipédia: https://hu.wikipedia.org/wiki/Mesters%C3%A9ges_neur%C3%A1lis_h%C3%A1l%C3%B3zat

Köszönetnyilvánítás

Nagyon szépen köszönöm a segítséget Dr. Varga László Gábor konzulensemnek, aki a szakdolgozatom elkészítésének minden fázisát segítette. Hozzáértése, pontossága, türelmessége és érthető magyarázatai rendkívül sokat segítettek.

Nyilatkozat

Alulírott Antal Zsolt, programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, programtervező informatikus BSC diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2022.12.10. aláírás: ……