

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MŰSZAKI ÉS HUMÁNTUDOMÁNYOK KAR,
MAROSVÁSÁRHELY
SZOFTVERFEJLESZTÉS SZAK

Párhuzamos képstílus átruházás konvolúciós
neuronhálókkal

MESTERI DISSZERTÁCIÓ



TÉMAVEZETŐ:
dr. Iclánzan Dávid
Egyetemi tanár

SZERZŐ:
Szilágyi Ervin

2017 Július

UNIVERSITATEA SAPIENTIA TÂRGU-MUREŞ
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE
SPECIALIZAREA DEZVOLTARE DE SOFTWARE

DeepArt

Lucrare de master



Coordonator științific:
dr. Iclăzan Dávid

Absolvent:
Szilágyi Ervin

2017 Iulie

SAPIENTIA UNIVERSITY TÂRGU MUREŞ
FACULTY OF TECHNICAL AND HUMAN SCIENCES
SOFTWARE DEVELOPMENT SPECIALIZATION

**Parallel artistic style transfer using deep
convolutional neural networks**

Master Thesis



Advisor:
dr. Iclănanzán Dávid

Student:
Szilágyi Ervin

2017 July

Declarație

Subsemnata/ul , absolvant(ă) al/a specializării, promoția cunoscând prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de disertație se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea,

Data:

Absolvant:

Semnătura.....

Declarație

Subsemnata/Subsemnatul ,
funcția, titlul științific
declar pe propria răspundere că absolventa/absolventul
de la specializarea a întocmit prezenta lucrare sub
îndrumarea mea.

Forma finală a lucrării a fost verificată de mine și aceasta corespunde cerințelor de formă și conținut precizate în Metodologia proprie de organizare a examenului de finalizare a studiilor (examen de licență/diplomă și disertație) în cadrul Universității Sapientia din Cluj-Napoca. Lucrarea/proiectul corespunde și cerințelor impuse de Legea Educației Naționale 1/2011 cu modificări ulterioare, Codului de etică și deontologie profesională a Universității Sapientia referitoare la furt intelectual. Sunt de acord cu susținerea lucrării în fața comisiei de examen de disertație.

Localitatea,

Data:

Semnătura îndrumătorului

KIVONAT

Az évek során számos festőművészeti irányzat alakult ki. mindenik rendelkezik a saját stílusával, ismertetőivel. Manapság a számítógépes látás eljutott arra a pontra, hogy képes legyen ezeket vonásokat felismerni és kategorizálni. Feltevődik a kérdés, hogy nem-e lehetne megfordítani a folyamatot oly módon, hogy a számítógép képes legyen művészeti képet alkotni?

Különböző stílusirányzatok alkalmazása képzett festőművészkeknek is kihívást jelenthet, nem beszélve arról, hogy a munka elvégzése rengeteg időt vehet igénybe. Dolgozatom célja egy olyan alkalmazás készítése ami rövid idő alatt képes híres magyar festők művészeti irányzatait alkalmazni és átruházni azt minden nap képekre illetve mozgóképekre.

A dolgozat a deep learning gépi tanulási módszert használja arra, hogy a művészeti kép vonásait alkalmazza minden nap képekre illetve videókra. Mozgóképek esetében egy olyan megoldást mutat be az átruházásra, ami figyelembe veszi képkockák közötti átmeneleteket, váltakozásokat is. Az átruházás folyamata kihasználja a számítógép videókártyája által biztosított párhuzamosítási lehetőségeket, lényegesen gyorsítva ezzel a folyamatot.

Szilágyi Ervin,

ABSTRACT

Pe parcursul anilor s-au dezvoltat numeroase curente artistice în domeniul picturii și a artelor plastice. Fiecare curent artistic posedă un stil unic care îl caracterizează. În prezent știința prelucrării imaginilor pe calculator a ajuns la un nivel, la care are posibilitatea de a recunoaște și de a cataloga curentele artistice. Se poate pune întrebarea dacă s-ar putea inversa acest proces îtrucât calculatorul să fie capabil de a crea imagini artistice.

Aplicarea trăsăturilor ale anumitor curente artistice la crearea unor picturi artistice reprezintă o provocare chiar și pentru un artist instruit. Totodată, timpul necesar creării unor picturi sau imagini artistice nu poate fi neglijat. Scopul acestei teze este de a prezenta și de a implementa o aplicație care să fie capabilă să preia trăsăturile artistice de pe picturi recunoscute ale marilor pictori maghiari. Aceste trăsături vor transferate și aplicate pe imagini neartistice și pe videouri.

Modalitatea de transfer a stilului artistic prezentată în această lucrare se bazează pe învățare automată folosind metoda deep learning. În cazul videoclipurilor se prezintă o metodă care ia în atenție și tranzițiile de pe un frame pe celălalt. Procesul de transfer a stilului se folosește de capacitatea de multithreading a plăcii video, reducând astfel timpul procesului semnificativ.

Szilágyi Ervin,

.....

ABSTRACT

Over the years several artistic movements were developed which had a considerable impact on the painting and fine arts. Every movement has an unique style with its own properties. In the present days computer vision reached to the point where distinguishing artistic style features does not represent a problem. A simple computer can recognize and group painting by their art movements. We can ask the question if there is possibility to reverse this process meaning that the machine itself would be able to create artistic pictures.

Applying style features from a give artistic movement consists a challenge even from a well-trained artist. Meanwhile, the necessary time for being able to create a good looking artistic picture is not neglectable at all. The scope of this project is to present and implement a solution which is able to transfer the style features from creation of well-known Hungarian painters and apply the features to images and videos created by the user.

The artistic style transfer is done using deep learning, a widespread machine learning method used for computer vision applications. The project also contains a solution which takes in consideration the transition and the temporal differences between frames in case of video input. The style transferring process is using the capability of the video card which is running massively parallel application. By using the data parallelization capability of a video card, we can reduce the learning time considerably.

Szilágyi Ervin,

.....

Tartalomjegyzék

| | |
|---|-----------|
| 1. Bevezető | 15 |
| 2. Hasonló rendszerek feltérképezése | 17 |
| 3. A rendszer | 19 |
| 3.1. Áttekintés | 19 |
| 3.2. Párhuzamos gépi tanulás Tensorflow segítségével | 21 |
| 3.2.1. A Tensorflow alap konceptusai | 21 |
| 3.2.2. Futtatás egyetlen készülék esetében | 22 |
| 3.2.3. Futtatás több készülék esetében | 22 |
| 3.2.4. Párhuzamos tanítás | 23 |
| 3.3. A tanítási módszer állóképek esetében | 24 |
| 3.3.1. Az eredeti kép tanításának a veszteségi függvénye | 24 |
| 3.3.2. Az stílus kép tanításának a veszteségi függvénye | 27 |
| 3.3.3. A stilizált kép tisztítása | 29 |
| 3.3.4. A teljes veszteségfüggvény felírása és tanítása statikus képek esetében | 30 |
| 3.4. A tanítási módszer mozgóképek esetében | 31 |
| 3.4.1. Naiv megközelítés | 31 |
| 3.4.2. A haló inicializálása képkockák esetében | 32 |
| 3.4.3. Optical flow bevezetése | 32 |
| 3.5. A rendszer tervezése és kivitelezése | 33 |
| 3.5.1. A rendszer interakciós és viselkedési modellje | 33 |
| 3.5.2. A rendszer strukturális modellje | 34 |
| 3.5.3. Többszálas megoldás és kommunikáció a komponensek között . . . | 35 |
| 3.5.4. A rendszer implementálása | 38 |
| 3.5.4.1. A főablak (Main Window) komponens implementálásának bemutatása | 38 |
| 3.5.4.2. A beállítások (Settings) komponens implementálása | 43 |
| 3.5.4.3. A progress sáv (Progress bar) komponens implementálása | 44 |
| 3.5.4.4. A stílusátruházó (Artistic video creator) komponens implementálása | 45 |
| 3.6. Limitációk | 48 |

| | |
|--|-----------|
| 4. A rendszer tesztelése | 49 |
| 4.1. A tesztrendszer ismertetése | 49 |
| 4.2. Egy képkockára történő stílusátruházási idő és memória igénye | 50 |
| 4.3. CPU-GPU futásidő összehasonlítása | 53 |
| 4.4. Videóra történő stílusátruházás időigénye | 53 |
| 5. Összefoglaló | 58 |

Cuprins

| | |
|--|-----------|
| 1. Întroducere | 15 |
| 2. Studiu bibliografic | 17 |
| 3. Sistemul | 19 |
| 3.1. Privire de ansamblu asupra | 19 |
| 3.2. Învățare automată cu ajutorul librăriei Tensorflow | 21 |
| 3.2.1. Conceptele de bază a librăriei Tensorflow | 21 |
| 3.2.2. Execuție pe un simplu device | 22 |
| 3.2.3. Execuție pe mai multe device-uri | 22 |
| 3.2.4. Antrenare paralelă | 23 |
| 3.3. Metoda de antrenare în cazul imaginilor statice | 24 |
| 3.3.1. Funcția de pierdere în cazul imaginii originale | 24 |
| 3.3.2. Funcția de pierdere în cazul imaginii de stil | 27 |
| 3.3.3. Reducerea zgomotului din imaginea stilizată | 29 |
| 3.3.4. Definirea și învățarea funcției de pierdere totală la imagini statice . | 30 |
| 3.4. Metoda de antrenare în cazul imaginilor video | 31 |
| 3.4.1. Abordarea naivă | 31 |
| 3.4.2. Inițializarea rețelei pentru frame-uri | 32 |
| 3.4.3. Introducerea optical flow-ului | 32 |
| 3.5. Proiectarea și implementarea sistemului | 33 |
| 3.5.1. Modelul de interacțiune și comportament al sistemului | 33 |
| 3.5.2. Modelul structural al sistemului | 34 |
| 3.5.3. Proiectarea pe mai multe fire computaționale a unelor componente . | 35 |
| 3.5.4. Implementarea sistemului | 38 |
| 3.5.4.1. Implementarea componentei Main Window | 38 |
| 3.5.4.2. Implementarea componentei Settings | 43 |
| 3.5.4.3. Implementarea componentei Progress bar | 44 |
| 3.5.4.4. Implementarea componentei Artistic video creator | 45 |
| 3.6. Limitații | 48 |
| 4. Testarea sistemului | 49 |

| | | |
|-----------|--|-----------|
| 4.1. | Configurația de test | 49 |
| 4.2. | Timpul și memoria necesară transferări stilului pe o imagine statică . . . | 50 |
| 4.3. | Compararea timpului necesar pe CPU cu cel necesar pe GPU | 53 |
| 4.4. | Timpul necesar transferări stilului pe un videoclip | 53 |
| 5. | Concluzie | 58 |

Table Of Contents

| | |
|---|-----------|
| 1. Introduction | 15 |
| 2. Bibliographic study | 17 |
| 3. The system | 19 |
| 3.1. Overview | 19 |
| 3.2. Parallel machine learning using Tensorflow | 21 |
| 3.2.1. The basic concepts of Tensorflow library | 21 |
| 3.2.2. Execution in case of a single device | 22 |
| 3.2.3. Execution in case of multiple devices | 22 |
| 3.2.4. Parallel trainig | 23 |
| 3.3. Learning method in case of static image inputs | 24 |
| 3.3.1. The loss function of the content image | 24 |
| 3.3.2. The loss function of the style image | 27 |
| 3.3.3. Denoising the stylized image | 29 |
| 3.3.4. The definition and optimization of the total loss function for static images | 30 |
| 3.4. Learning method in case of video inputs | 31 |
| 3.4.1. Naiv approach | 31 |
| 3.4.2. The initialization of the neural net for the frames | 32 |
| 3.4.3. Introduction of the optical flow | 32 |
| 3.5. The design and implementation of the system | 33 |
| 3.5.1. Interraction and behavioral model of the system | 33 |
| 3.5.2. Structural model of the system | 34 |
| 3.5.3. Multithreaded design and communication of components | 35 |
| 3.5.4. The implementation of the system | 38 |
| 3.5.4.1.The implementation of the Main Window component | 38 |
| 3.5.4.2.The implementation of the Settings component | 43 |
| 3.5.4.3.The implementation of the Progress bar component | 44 |
| 3.5.4.4.The implementation of the Artistic video creator component | 45 |
| 3.6. Limitations | 48 |

| | |
|---|-----------|
| TABLE OF CONTENTS | 14 |
| 4. The testing of the system | 49 |
| 4.1. Test configuration | 49 |
| 4.2. The amount of time and memory needed for transferring the style to a frame | 50 |
| 4.3. Comparison of the running time between CPU and GPU | 53 |
| 4.4. The amount of time needed for transferring the style to a video | 53 |
| 5. Conclusion | 58 |

1. fejezet

Bevezető

Napjainkban a képfeldolgozás egy elégé elterjedt kutatási terület. A kutatások célja főleg az információ kinyerésére, gépi látás kivitelezésére irányult. Minderre kiváló megoldást jelentett a mély konvolúciós hálók (ConvNets)[1][2] sikeres használata növelte ezzel ezek népszerűségét. Fontos megjegyezni, hogy a konvolúciós neuron hálók felfedezése már pár évtizede történt, tehát maga a technológia már régebb ismert volt. Az újrafelfedezésüköt és hirtelen népszerűség növekedését annak köszönhetik, hogy az utóbbi években olyan hardveres megoldások jelentek meg, amik lehetővé teszik az ilyen típusú hálók létrehozását és működtetését.

Az Nvidia cég 2007-ben bevezette az Nvidia CUDA platformot[3]. Ez egy komoly, használható fejlesztő környezetet jelentett olyan fejlesztők számára akik nagy méretű adatpárhuzamos algoritmusokat szerettek volna fejleszteni. A CUDA környezet direkt elérhetőséget nyújt a videókártya utasításkészletéhez megengedve ezzel ennek a programozását. Ugyanakkor számos olyan videókártya került piacra ami egyre komolyabb számítási készségekkel bírt. Ezt a lehetőséget értelemszerűen a kutatók ki is használták így számos újabb publikáció és javaslat jelent meg amik neuron hálókat használnak az illető probléma megoldására.

A deep konvolúciós hálók népszerűségének növekedésével egyre több olyan fejlesztői környezet jelent meg amiknek célja a mesterséges intelligencia feladatok megoldása. Ilyen könyvtárak például a Caffe[4], Keras[5], Theano[6], Tensorflow[7], Torch[8] stb. Ezek a környezetekben, habár különböző stílusban de egyazon problémáakra hivatottak gyors és egyszerű megoldásokat ajánlja ugyanúgy mezei szoftverfejlesztők, mint kutatók számára.

Az gépi látás egyik fontos alkalmazási területe a képen levő tárgyak, élőlények emberek felismerése. Ilyen területen a konvolúciós hálók kimagasló teljesítményt nyújtanak, olyannyira, hogy egyes kísérletek szerint ez már nemhogy az emberi látással megegyező, hanem azt felülműlő teljesítményt nyújtanak[9]. Feltevődik a kérdés, hogyha ennyire szofisztikált a gépi látás, akkor nem-e lehetne használni arra, hogy új képeket alkosszon. Amint kiderült erre is alkalmasak. Az általam bemutatandó dolgozat is ezt a témát próbálja megcélozni. A gépi látás a tárgyak, élőlények mellett képes felismerni maga a kép

művészeti stílusát. Ez elsősorban kihasználható arra, hogy híres művészek alkotásait csoportosítsuk, rendszerezzük[10], de amint e dolgozatból ki fog derülni, ki lehet használni arra is, hogy egy művészeti stílust egy adott festményről átvigyük egy minden nap képre, fotóra.

A dolgozatom célja magyar híres festőművészek festészeti stílusát átvenni és ezt alkalmazni minden nap képekre illetve mozgóképekre. Eddigiekben, ahhoz hogy egy minden nap fényképből művészeti képet varázsolunk, képszerkesztő szoftverek segítségével lehetett elérni manuálisan. Mindezt egy olyan egyén végezhette, akinek képszerkesztési illetve képmobilálási szakismere volt adott képszerkesztési szoftverkörnyezetben. Magától értődik az, hogy ez mozgóképek esetében egy időigényes folyamat. Dolgozatom mindenekre megoldást próbál adni, azáltal, hogy az általam elkészített szoftvert bárki használhatja, nincs szükség különböző képszerkesztői szakértelemre, emellett a folyamat ideje jelentősen csökkeni fog.

2. fejezet

Hasonló rendszerek feltérképezése

A neuron hálók használata a számítástechnikában nem egy újonnan kialakult terület. Frank Rosenblatt 1958-ban publikált egy olyan mintafelismerő algoritmust[11], ami egyszerű összeadást és kivonást használva képes volt "tanulni". A rendszer képes volt finomhangolni állapotát a bekövetkező iterációk során. Ezt az algoritmust perceptronnak nevezzük. 1975-ben Paul Werbos vezette be a backpropagation algoritmust[12], amit a perceptronnal együtt használva megoldotta a perceptron azon problémáját miszerint az csak lineárisan elválasztható osztályokat volt képes kategorizálni. Habár a neuron hálók tanulmányozása eléggyé ígéretesnek látszott, számítási igényük, komplexitásuk és lassú válasz idejük miatt a kutatók arra következtetésre jutottak, hogy a gyakorlatban még nem lehet alkalmazni őket.

Yann LeCun professzor és csapata 1998-ban egy újabb topológiójú hálót vezetett be[13]. A LeNet-5 elnevezésű háló konvoluciós rétegeket is tartalmazott ezért konvoluciós neuron hálónak nevezzük. A publikáció célja kézzel írott számjegyek kategorizálása volt, létrehozva ezáltal a MNIST adatbázist, ami 60000 28x28-as felbontású kézzel írott számjegyet tartalmaz, emellett tartalmaz egy 10000 tagból álló teszthalmazt. A dolgozatban bemutatott LeNet-5 háló 0,7%-os hiba aránnyal volt képes kategorizálni a számjegyeket, ami messze felülmúltja a többrétegű perceptronos megoldást.

Dave Steinkraus, Patrice Simard és Ian Buck 2005-ben publikált dolgozata[14] letette az alapjait a neuronhálók videokártyán történő programozásának. A videókártyán történő adatpárhuzamos programozás hatalmas performancia növekedést jelentett a processzoron futó neuronhálókkal szemben. Előtérbe kerül a deep learning és a mély konvoluciós hálók használata[1][2].

Eddigiekben sikerült nagyon pontos felismerő illetve osztályozó rendszereket alkotni. A mély konvoluciós hálók használata azonban nem merül ki ennyiben. 2015-ben publikálásra került egy olyan deep learning-et használó algoritmus, ami képek illetve festmények művészeti stílusát átvinni egy másik digitális képre[15]. Mostani dolgozatom is erre a publikációra alapoz, az ebben bemutatott módszereket próbálja alkalmazni illetve továbbfejleszteni. A tanuláshoz egy korábban bevezetett és gépi látáshoz használt, előre

betanított neuron hálót használnak fel, a VGG-19-et. Yaroslav Nikulin és Roman Novak tudományos kutatása[16] ezzel szemben eddig ugyanezt a módszert alkalmazta más ismertebb előre betanított hálókra, mint például AlexNet, GoogLeNet vagy VGG-16. Ugyanúgy a VGG-16 háló használata is kiváló eredményeket mutatott még a GoogLeNet és az AlexNet architektúrájuk miatt komolyabb információvesztéshez vezetnek így a végeredmény nem lesz annyira látványos. Ugyanúgy kísérletek irányultak az eredeti eljárás optimalizálására, megjelentek olyan rendszerek amik sajátos, erre a célre betanított neuron hálókat alkalmaznak[17][18][19].

2016-ban a Prisma labs inc. kiadta mobilos applikációját Prisma név alatt[20]. Az applikáció előre megadott ismert festői/grafikai stílusokat alkalmazza a telefon kamerája által készített képekre. Az applikáció az előbbieken bemutatott kutatásokra alapoz. Ugyanakkor fontos megjegyezni, hogy maga a stílus alkalmazását a különböző fotókra nem az okostelefon végzi. A szerkeszteni kívánt képet a telefon felkülde egy szervergépre ami majd válaszként a szerkesztett képet küldi vissza.

Maga stílusátvitel nem csak állóképekre alkalmazható, ezt bizonyította Manuel R., Alexey D., Thomas B. tudományos dolgozata[21], valamit ezt próbalja megoldani a jelenlegi dolgozatom is. Értelemszerűen egy adott videót több álló képkocka alkot. Viszont ahhoz, hogy látványos művészeti mozgóképet gyártunk, nem elegendő maga a videót darabokra vágni és minden képkockára alkalmazni a stílust. Erre adott megoldást Manuel R. és társainak kutatása.

3. fejezet

A rendszer

3.1. Áttekintés

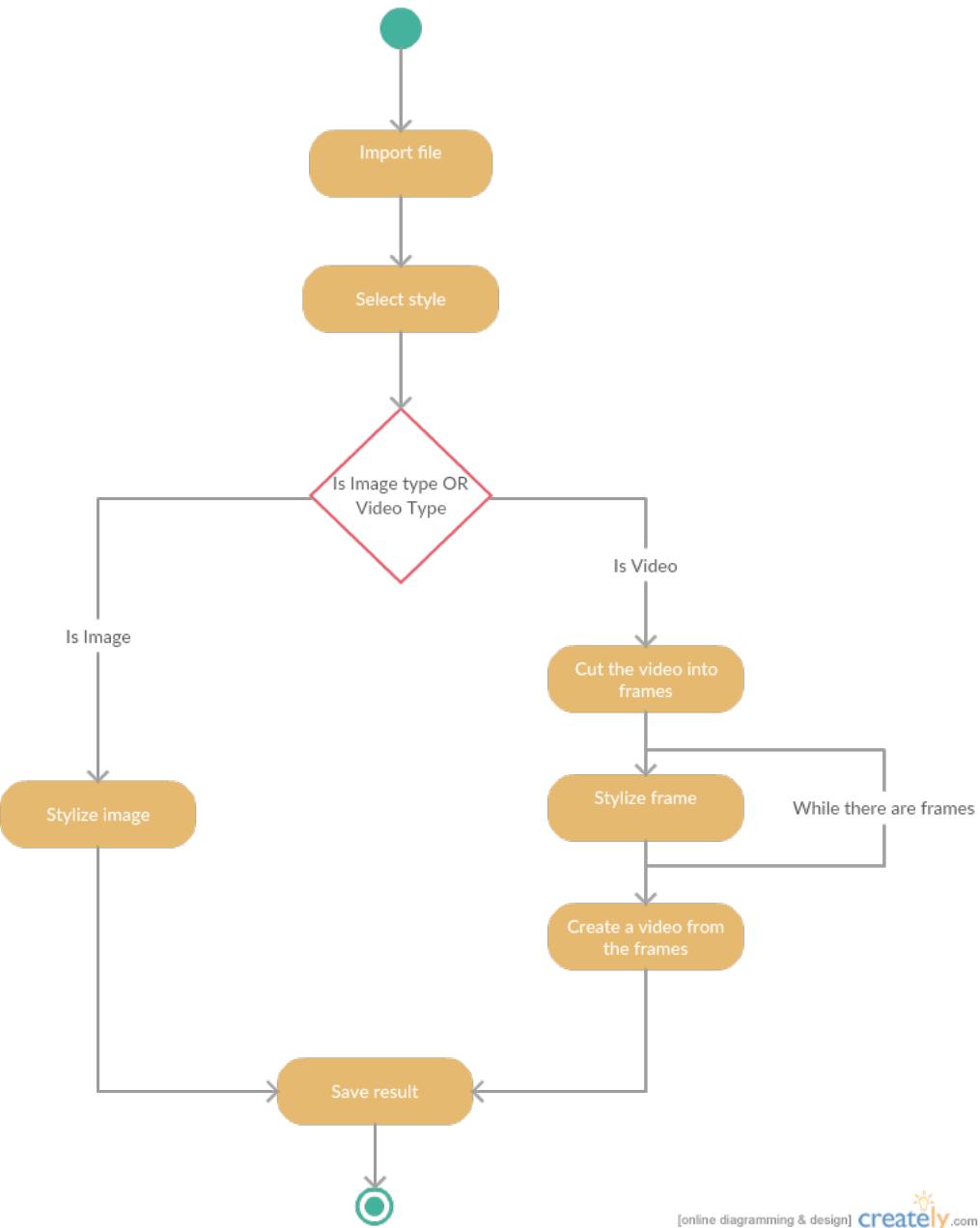
Dolgozatom célja egy olyan multiplatform számítástechnikai szoftver tervezése és fejlesztése ami deep learning-et használva képes híres magyar festők festményeinek a stílusát átvenni és alkalmazni a felhasználó által megadott digitális képekre illetve videókra. A fejlesztett szoftver könnyen használható grafikus felhasználói felülettel rendelkezik és támogatja a Linux valamint a Microsoft Windows alapú operációs rendszereket. A szoftver futtatásához a felhasználónak rendelkeznie kell egy olyan videókártyával ami támogatja az Nvidia CUDA platformot.

A szoftver fejlesztése Python3.5[23] programozási nyelvben történt, viszont egyes esetekben felhasználásra kerülnek egyes előre legyártott önállóan is futtatható állományok. Emellett még használva vannak a következő Python könyvtárak:

- numpy[24]: használata elengedhetetlen akkor, ha többdimenziós tömbökkel szereznénk dolgozni. Nagyon sok matematikai problémára tartalmaz előre definiált megoldást és efelett tökéletesen használható a tensorflow könyvtár mellett
- tensorflow[7]: talán egyik legismertebb deep learning és mély konvolúciós hálók tanítására kifejlesztett könyvtár. Teljes mértékben támogatja a videókártyán történő programozást.
- PyQt[25]: a szoftver grafikus felhasználói felületének a megvalósításához használatos, emellett komoly feladat orientált párhuzamosítást tud biztosítani.
- OpenCV[26]: képfeldolgozásra szakosodott könyvtár, főleg a különböző kiterjesztésű képek beolvasására és mentésére volt használva.

A rendszer működése felülnézetből nagyon egyszerű (3.1 ábra). A felhasználó kiválaszt egy bemeneti állományt, ami kép kiterjesztésű (.jpg, .png) vagy mozgókép kiterjesztésű (.gif, .avi, .mp4) lehet valamint kiválaszt egy stílust a megadottak közül. A rendszer

annak függvényében, hogy milyen bemenetet adtunk, eldönti, hogy kép vagy mozgóképpel kell dolgozna. Kép esetén egyszerűen lefuttatja a tanulási algoritmust amely során az átveszi a művészeti kép stílusát. Mozgókép esetén képkockákra bontja azt, majd minden képkockára alkalmazva lesz a tanítási algoritmus. Ha összes képkocka szerkesztve lett, akkor a rendszer felépíti a képkockákból a kimeneti videót. Ezek után, függetlenül, hogy kép vagy videó lett a végeredmény, a rendszer kimenti azt egy megadott folderbe.



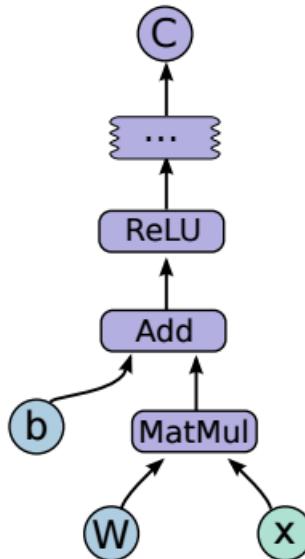
3.1. ábra. A rendszer működése felülnézetből

3.2. Párhuzamos gépi tanulás Tensorflow segítségével

3.2.1. A Tensorflow alap konceptusai

A Tensorflow egy gépi tanításra alkalmas könyvtár. Tensorflow segítségével leírt algoritmus nagyon kevés kódmodosítással számos eszközön futtatható, kezdve a mobil eszközöktől egészen a GPU-kal ellátott osztott rendszerekig. Főleg a deep learning típusú algoritmusok implementálására használatos, viszont flexibilitásából adódóan számos más tanítási folyamat elvégzésére is képes [35].

A Tensorflow működése leírható egy irányított gráf segítségével. A gráf az adat áramlását reprezentálja, míg a gráfban levő csúcsok jelentik a kontrollt az adat áramlása fölött. Egy ilyen gráfban minden csúcsnak 0 vagy több bemenetele van, valamint 0 vagy több kimenettel rendelkezik. minden csúcs egy operációt jelent. A csúcsok között folyó adathalmazokat tensoroknak nevezünk. Valójában egy tenzor egy N dimenziós tömböt jelent. Speciális élek is létezhetnek a gráfban az adatfolyam kontrollálására. Ilyen él lehet például egy olyan, ami megtiltja, hogy adat folyjon át rajta [35].



3.2. ábra. Tensorflow számítási gráf[35]

Az operációknak lehetnek nevük és attribútumaik. A kernel egy operáció implementációja, ami egy adott típusú hardveren (CPU vagy GPU) fut. Különféle típusú kerneleket különböztetünk meg, amik lehetnek alap matematikai műveletek (összeadás, kivonás, szorzás, osztás, logaritmus, kisebb, nagyobb, egyenlőség, stb.), tömb műveletek (konkatenálás, felvágás, rang, stb.), mátrix műveletek (mátrix szorzás, inverz, determináns, stb.), neuron háló réteg típusok (softmax, sigmoid, ReLu, konvolúciós, stb.), szinkronizációs műveletek (mutex műveletek, sorba állítás) és adatfolyam kontrollálásra szolgáló műveletek.

Ahhoz, hogy a számítási gráfban definiált műveleteket elvégezhessük, szessziókat (session) kell definiálni. Szessziók használata megengedi a teljes gráf kiértékelését vagy akár részleges kiértékelést is. A gráf kiértékelése a szesszió Run függvényével történik.

A változó egy olyan tenzor ami megtartja az értékét többszörös számítási gráf kiértékelés esetén is. A változó egy olyan tenzorra utal aminek tartalma megváltozhat, ezzel ellentében létezik a konstans aminek tartalma nem változhat.

A Tensorflow párhuzamosításra úgynevezett processzeket használ (3.3. ábra). A kliens szoftver egy szesszió interfészen keresztül kommunikál a master processsel és a worker proceszekkel. mindenik procesz felelős egy vagy több számítási eszközért (computation device), amik lehetnek CPU magok vagy GPU kártyák. Ezek az eszközök lehetnek lokálisak, ami azt jelenti, hogy ugyanabban a rendszerben helyezkednek el, vagy lehetnek osztottak, ez esetben külön rendszerekben helyezkednek el a különféle eszközök. minden eszköz saját névvel rendelkezik[35].

3.2.2. Futtatás egyetlen készülék esetében

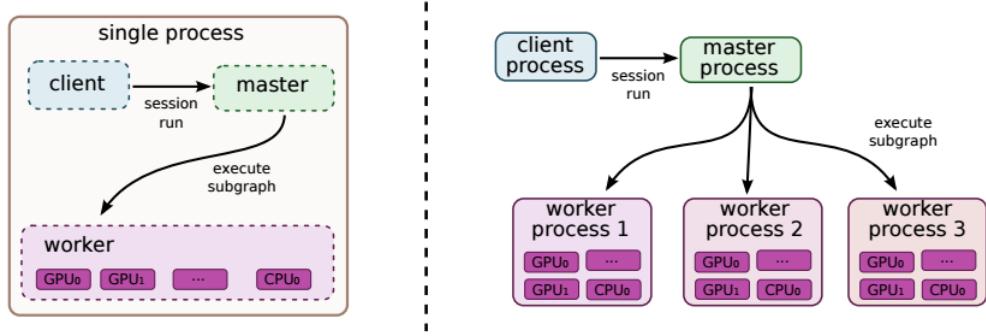
Egyetlen eszköz jelenlétében a csúcsok egymást követően lesznek kiértékelve figyelembe véve a közöttük levő dependenciákat. minden csúcs esetében számolva van, hogy hány dependenciát tartalmaz, ha ezek közül egyik ki volt értékelve, akkor ez a számláló csökken. Mikor a számláló elérte a 0-át, akkor tovább lép és kiértékeli e következő csúcsot. A kiértékelésre várakozó csúcsok egy listába vannak elhelyezve.

3.2.3. Futtatás több készülék esetében

Több készülék esetében szükséges eldönteni, hogy az adott csúcsok melyik készüléken fognak kiértékelődni, valamint fontos megoldani a készülékek közötti kommunikációt.

A csúcsok elhelyezése egy mohó esztimációs módszert követ. minden csúcs esetében megesztimálható, hogy a bemeneti és kimeneti tenzorok mérete byte-ban. Ugyanakkor megesztimálható a futási idő adott csúcsok esetében. Ezután egy szimuláció fog lefutni ami az összes csúcs esetében kilistázza a használható eszközöket majd kiértékeli a futási időt valamit a kommunikáció idejét az eddigi elhelyezett csúcsokkal. A mohó algoritmus szerint arra az eszközre fog kerülni, ami a szimuláció során a legjobb eredményt adta.

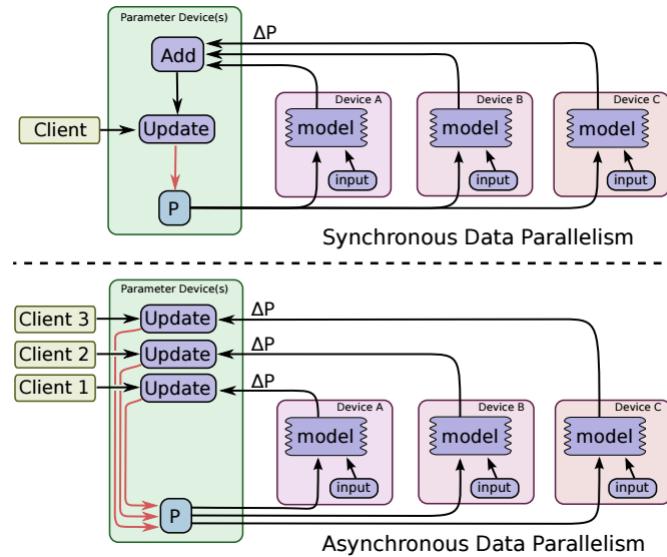
Az eszközök közötti kommunikációt Send-Receive csúcsok beiktatásával oldja meg. Ezek a csúcsok ugyanakkor megoldják az eszközök közötti szinkronizációt is. Osztott rendszerek esetében ugyancsak Send-Receive csúcsok oldják meg a kommunikációt. Ezek TCP és RDMA protokollokat használnak az adat mozgatásához[35].



3.3. ábra. Egyetlen procesz összehasonlítása több proceszes működéssel[35]

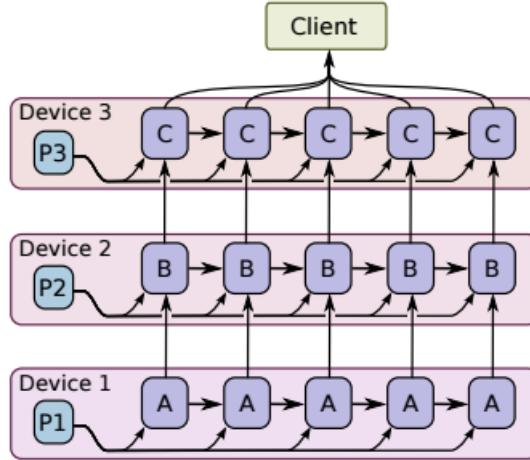
3.2.4. Párhuzamos tanítás

A Tensorflow könyvtár az adatpárhuzamosítást megközelíti úgy adatpárhuzamosítás szempontjából, mint feladat párhuzamosítás szempontjából. Adatpárhuzamosítás esetében az adatot kisebb csomagokba bontja, majd ezeket párhuzamosan bemenetként táplálja a megfelelő csomópontonknak. A kiértékelendő csomópontokból egy másolat kerül az összes szálra (3.4. ábra). A végeredményt egy redukciós művelet gyűjti össze.



3.4. ábra. Adatpárhuzamos szinkron és aszinkron csomópont kiértékelés[35]

Feladatpárhuzamosítás esetében különböző gráfrészletek lesznek megtanítva ugyanabban az időben különböző szálakon. Mikor a tanítás befejeződik, akkor az a procesz egy újabb gráfrészlet kiértékelésének fog neki. Ilyenfajta tanítás esetében fontos figyelembe venni azt, hogy egyes gráfrészletek csak akkor értékelhetőek ki, ha már előttük egy másik gráfrészlet le volt futtatva. Feladatpárhuzamos tanítást a 3.4. ábra szemlélteti.



3.5. ábra. Feladatpárhuzamos tanítás[35]

3.3. A tanítási módszer állóképek esetében

A rendszer tanításához állókép esetében két legalább képet bemenet szükséges, az eredeti kép, amire át szeretnénk ruházni a stílust és a stílust tartalmazó kép, aminek a stílusát át szeretnénk ruházni. Mindkét bemenet esetében felírunk egy veszteség függvényt amik részei a végső nagy tanítási függvénynek.

3.3.1. Az eredeti kép tanításának a veszteségi függvénye

A mély neurálos hálók (Deep Neural Networks) azon típusai amik a legeredményesebbek a képfeldolgozási feladatok elvégzésben a konvolúciós hálók. Mesterséges intelligencia területén a konvolúciós hálók olyan feed-forward típusú neuronhálók, amiket a biológiai elsődleges látókéregről mintáztak. A háló kifejezetten arra volt tervezve és kifejlesztve, hogy kétdimenziós formákat ismerjen fel. A háló alapértelmezetten több rétegből tevődik össze:

- konvolúciós réteg: a konvolúciós hálók alap kövei. A réteg súlyai úgynevezett konvolúciós szűrők alkotják, amelyek a forward pass lépés során tanítva vannak. Ez a tanítási lépés úgy történik, hogy a szűrőt végig toljuk a bemeneten és konvolúciónak nevezett műveletet végzünk.
- pooling layer: arra használatos, hogy a bemeneti adathalmazt méretét leszűkítsük úgy, hogy a halmaz adott értékein valamely matematikai műveletet végzünk (át-lagszámolás, maximum számolás, minimum számolás). Erre azért van szükségünk, hogy növeljük a tanulás gyorsaságát, csökkentsük a számítások komplexitását megakadályozva ezzel az "overfitting" bekövetkezését.
- fully connected layer: minden bemenet minden más bemenettel kapcsolatban áll.

A konvolúciós hálók súlyzókként konvolúciós szűrőket tartalmaznak. Ezek, objektum felismerés tanítása esetében, az adott bemeneti kép különböző tulajdonságait fogják tartalmazni. Annak függvényében, hogy a háló topolójában egy adott réteg hol helyezkedik el, más tulajdonságokat fognak raktározni a szűrök. Az bemeneti réteghez közel álló alacsony szintű rétegek az adott kép pixelinformációt próbálják megjegyezni ezzel szemben a magasabb szinten levő rétegek szűrői különböző tárgyakat, formákat próbálnak megjegyezni[27][28]. A hálók által tartalmazott információ vizualizálható azáltal, hogy rekonstruáljuk a bemeneti képet a súlyzók alapján. Alacsony rétegek esetében apró módosításokkal visszányerhető az eredeti kép míg magasabb rétegek esetében objektumok, formák nyerhetők vissza.

Az eredi kép tanításához egy már előre betanított mély konvolúciós neuronhálót használunk fel. A háló tudományos kontextusban VGG-19 név alatt terjedt el amit Simonyan K. és Zisserman A. vezetett be publikációjukban[29] ahol még a "model E" nevet viselte. Ez a háló gépi látás és tárgyfelismerés céljából volt bevezetve olyan eredményeket produkálva e téren amik az emberi látással versengenek. Rétegei és topolójának részletes bemutatása a [29] publikációban történik, valamint a 3.6 figurán is látható. Fontos megjegyezni, hogy a háló rétegeiből használva volt a 16 konvolúciós réteg, valamint az 5 pooling réteg, a teljesen összekötött rétegek nem voltak használva.

A választás azért esett erre az előre betanított hálóra, mivel Yaroslav N. és Roman N. kutatása alapján[16] összehasonlítva a AlexNet, GoogLeNet VGG-16 és VGG-19 halókat, a VGG-19 használata során sikeres volt a leglátványosabb képeket készíteni.

A konvolúciós neuron háló minden rétege tartalmaz egy adott számú szűrőt, amik különböző tulajdonságokat tartalmaznak a bemeni képről. Ebből kifolyólag maga a bemenet kódolva van a szűrőkben. Egy N tulajdonságot tartalmazó réteg N szűrővel rendelkezik amelyek mérete M . Maga a réteg kimenete lementhető egy $N * M$ -es mátrixban. Egy adott réteg válasza egy bemeneti képre vizualizálható, ha gradient descent módszert alkalmazunk egy fehér zajt tartalmazó bemeneti képen. Tehát, legyen R^l egy adott az l haló válasza egy adott bemeneti képre. Legyen W^l ugyanaz az l háló válasza egy adott fehér zajt tartalmazó bemeneti kép esetében. Akkor felírható a veszteségfüggvény mint:

$$L_{content}(\vec{x}, \vec{r}, l) = \frac{1}{2} \sum R_{ij}^l - W_{ij}^l \quad (3.1)$$

Ahol \vec{x} a bemeneti képet jelenti, \vec{r} pedig azt a kimeneti képet jelenti amit a rendszer generál a rétegek tulajdonságaiból. Mindez Tensorflow környezetben a következőképpen nézne ki:

```
for layer_name in CONTENT_LAYER:
    content_loss += content_weight * (2 * tf.nn.l2_loss(
        all_layers[layer_name] - content_features[layer_name]) /
        content_features[layer_name].size)
```

| ConvNet Configuration | | | | | |
|-------------------------------------|------------------------|-------------------------------|--|--|--|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224×224 RGB image) | | | | | |
| conv3-64 | conv3-64 LRN | conv3-64 conv3-64 | conv3-64 conv3-64 | conv3-64 conv3-64 | conv3-64 conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128 conv3-128 | conv3-128 conv3-128 | conv3-128 conv3-128 | conv3-128 conv3-128 |
| maxpool | | | | | |
| conv3-256 conv3-256 | conv3-256 conv3-256 | conv3-256 conv3-256 | conv3-256 conv3-256 conv1-256 | conv3-256 conv3-256 conv3-256 | conv3-256 conv3-256 conv3-256 conv3-256 |
| maxpool | | | | | |
| conv3-512 conv3-512 | conv3-512 conv3-512 | conv3-512 conv3-512 | conv3-512 conv3-512 conv1-512 | conv3-512 conv3-512 conv3-512 | conv3-512 conv3-512 conv3-512 conv3-512 |
| maxpool | | | | | |
| conv3-512 conv3-512 | conv3-512 conv3-512 | conv3-512 conv3-512 | conv3-512 conv3-512 conv1-512 | conv3-512 conv3-512 conv3-512 | conv3-512 conv3-512 conv3-512 conv3-512 |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

3.6. ábra. VGG-19 (Model E) háló topológiája[29]

```
content_loss /= len(CONTENT_LAYER)
return content_loss
```

A *CONTENT_LAYER* tuple típusú ami tartalmazza azoknak a rétegeknek az azonosítóját amik részt vesznek a veszteség függvény kiszámításában. A mi esetünkben ez egyedül a *relu4_2* kimenetét jelenti, tehát valójában a *conv4_2* konvolúciós réteget használtuk. A *content_features* változó egy lista, ez tartalmazza az összes réteg válaszát a bemeneti eredeti képre. A visszaküldött érték egy tensor típusú, egyik részét fogja képezni a végső optimalizálálandó veszteségfüggvénynek.

3.3.2. Az stílus kép tanításának a veszteségi függvénye

Az előbbiekben az eredeti bemeneti kép tartalmára voltunk kíváncsiak. A stílus kép esetében viszont nem maga a kép tartalma a fontos. Ebben az esetben egy mintázatot szeretnénk kinyerni a stílusképből. Ehhez fontos megismerni maga a Gramm mátrix fogalmát. Hogyha adott egy vektorhalmazunk $v_1 \dots v_n$, akkor a G Gramm mátrixot a következő eljárás szerint határozzuk meg[30]:

$$G_{ij} = v_i \cdot v_j \quad (3.2)$$

Tehát hogyha az A mátrixunk oszlopait maga a $v_1, v_2 \dots v_n$ vektorok képezik, akkor a G Gramm mátrix a következőképpen kapható meg:

$$G_{ij} = AA^T \quad (3.3)$$

A Gramm mátrix egy szorzatot jelen egy adott vektorhalmaz összes elemei között. A mi esetünkben ez a vektorhalmaz jelentheti egy adott konvolúciós réteg által kiszűrt tulajdonságokat az adott bemeneti stílusképből. Amint már említettem, maga a konvolúciós réteg szűrői egy adott kép tulajdonságait tartalmazzák. A Gramm mátrix ij pozíciójában elhelyezkedő elem megadja, hogy egy adott réteg i -dik tulajdonsága mennyire teljesül a j -dik tulajdonság jelenlétében, tehát a két tulajdonság milyen mértékben aktiválódik egyszerre. Ha az ij pozícion levő elem közelít a 0-hoz, akkor az azt jelenti, hogy a két tulajdonság nem aktiválódik egyszerre, ezzel ellentétben, ha az érték nagy, akkor az azt jelenti, hogy a két tulajdonság nagy valószínűséggel aktiválódik egyszerre.

Hogyha az l rétegnek N szűrője van, akkor $G \in R^{N_l \times N_l}$, ahol:

$$G_{ij}^l = \sum_k F_{ik}^l \cdot F_{jk}^l \quad (3.4)$$

Ahhoz, hogy megkapjuk egy adott háló által generált mintát, textúrát, hasonlóan mint az előzőekben, fehér zaj képet adunk bemenetként. A veszteségfüggvény egyetlen rétegre felírható mint a fehér zaj kép Gramm mátrixa és a stílus kép Gramm mátrixának átlagos négyzetes hibájaként. A G jelöli a fehér zaj kép Gramm mátrixát, az A pedig a stílus kép Gramm mátrixát.

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij})^2 \quad (3.5)$$

Az összes stílus réteg válasza felírható, mint egy összeg, ahol w_l egy súlyzó faktort jelent:

$$L_{style}(\vec{a}, \vec{x}) = \sum w_l E_l \quad (3.6)$$

A 3.7. ábra szemlélteti a stílus súlyzó változtatásának hatását. A bemeneti kép a "Sapientia épülete", a stíluskép Munkácsy Mihály "Vihar a pusztán" mesterműve. Az ábra bemutatja, hogy miképpen változik a kimeneti kép a stílus súlyzójának függvényében.



Sapientia épülete



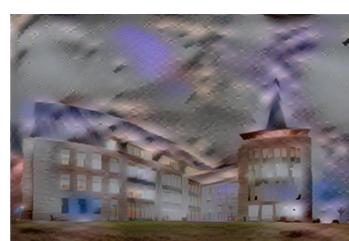
Munkácsy Mihály -
Vihar a pusztán



$w = 10$



$w = 100$



$w = 200$



$w = 500$

3.7. ábra. A stílus súlyzó módosításának eredményei

A Gramm mátrix számolása Tensorflow környezetben a következőképpen történik:

```
# reshape the tensor to be 2 dimensional
features = tf.reshape(layer, (-1, number_of_channels))
# create Gram matrix which basically is the product of
# the mat with itself
gram = tf.matmul(tf.transpose(features), features) / size
```

Maga a veszteségfüggvény meg a következőképpen implementálható Tensorflow könyvtár függvényei felhasználva:

```
# iterate through style layers
for style_layer in STYLE_LAYERS:
    # get the style layer
    layer = all_layers[style_layer]
    # compute the Gram matrix for the layer
    gram = self._create_gram_matrix(layer)
    # append the gram value to a list
    style_gram = style_features[i][style_layer]
    # compute the style loss for the layer and
    # add it to a list
    style_loss = style_weight * (2 *
                                 tf.nn.l2_loss(gram - style_gram) / style_gram.size)
```

A *style_features* változó, ugyanúgy mint a *content_features* esetében, egy lista, ez tartalmazza az összes réteg válaszát a bemeneti stílus képre. Az alábbi konvolúcióirrétegek kerülnek felhasználásra: *conv1_1*, *conv2_1*, *conv3_1*, *conv4_1*, *conv5_1*.

3.3.3. A stilizált kép tisztítása

A két veszteségfüggvényt alkalmazva sikeresen át tudjuk ruházni a stílust a művészkiéről a bemeneti minden napireprént. Viszont észlelhetjük, hogy az eredmény kép elégé zajos. Ennek érdekében vezetünk egy újabb veszteségfüggvényt a zaj csökkentésére, ami a Total Variation Denoising algoritmusra alapszik. Az algoritmus szerint vesszük a stilizált képet és eltoljuk X koordináta mentén egy pixellel, majd az Y koordináta mentén is eltoljuk egy pixellel. Az eltolt képeket kivonjuk az eredeti képekből, majd az eredmények abszolút értékeit pixelenként összeadjuk. Ezáltal egy újabb veszteségi függvényt alítunk elő, amit ugyancsak minimalizálni kell.

$$L_{tv}(\vec{a}, \vec{x}) = \sum_{i,j} |(X_{ij} - A_{i+1,j})| + \sum_{i,j} |(X_{ij} - A_{i,j+1})| \quad (3.7)$$

Ennek az implementációja pedig:

```

tv_loss = tv_weight * 2 * (
(tf.nn.l2_loss(image[:, 1:, :, :] - image[:, :, :shape[1] - 1, :, :]) /
self.size_of_tensor(image[:, 1:, :, :])) +
(tf.nn.l2_loss(image[:, :, 1:, :] - image[:, :, :, :shape[2] - 1, :, :]) /
self.size_of_tensor(image[:, :, 1:, :]))
)

```

3.3.4. A teljes veszteségfüggvény felírása és tanítása statikus képek esetében

Eddig bemutatásra került a stílus kép tanításának veszteségfüggvénye, az eredeti kép veszteség függvényének tanítása valamint egy veszteségfüggvény a kimeneti kép zajtalanítására. A végső veszteségfüggvény egyszerűen felírható a három veszteségfüggvény összegeként.

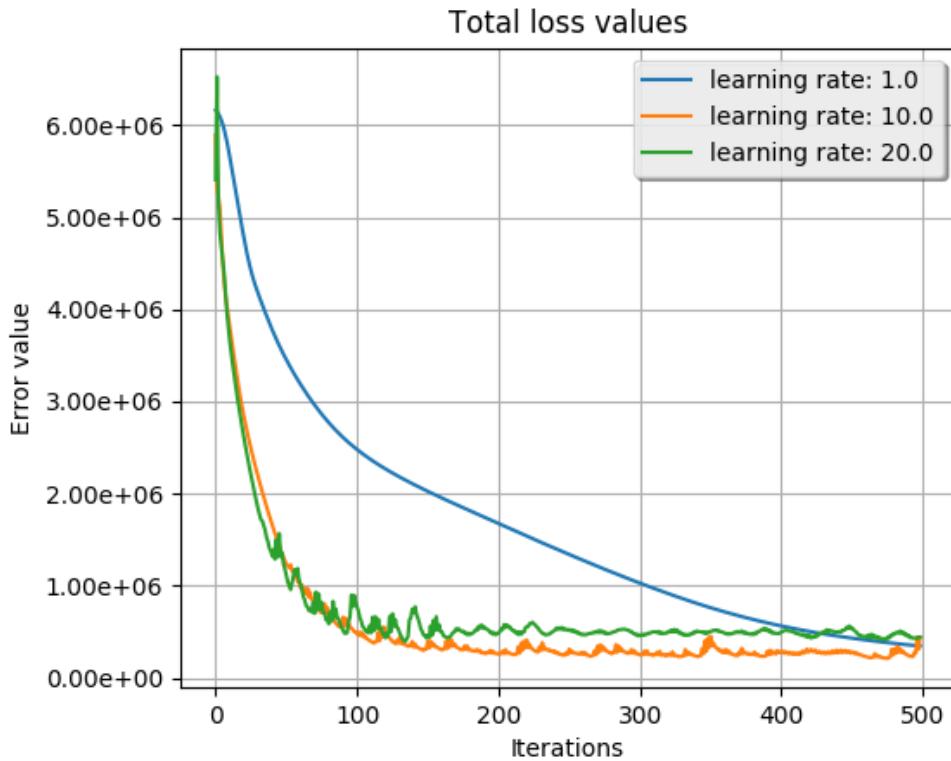
$$L = L_{content} + L_{style} + L_{tv} \quad (3.8)$$

A teljes veszteségfüggvény tanítására az Adam[32] optimalizáló algoritmus alkalmaztuk. Az Adam (Adaptive Moment Estimation) momentum algoritmus egy sztochasztikus gradiens alapú optimalizáló algoritmus. Az algoritmus felhasználja a elsődleges, valamint a másodlagos gradiensek átlagát ahhoz, hogy a veszteségfüggvényt minimalizálja. Az optimalizálási lépések a következő eljárás szerint történnek:

$$\begin{aligned}
(m_t)_i &= \beta_1(m_{t-1})_i + (1 - \beta_1)(\nabla L(W_t))_i \\
(v_t)_i &= \beta_2(m_{t-1})_i + (1 - \beta_2)(\nabla L(W_t))_i^2 \\
(W_{t+1})_i &= (W_t)_i - \alpha \frac{\sqrt{1 - (\beta_2)_i^t}}{1 - (\beta_1)_i^t} \frac{(m_t)_i}{\sqrt{(v_t)_i} + \varepsilon}
\end{aligned} \quad [33] \quad (3.9)$$

Ahol, m és v jelentik az elsőfokú valamint másodfokú gradienseket, W jelenti a súlyzókat, β_1 és β_2 jelentik a momentumokat, emellett ε egy nagyon kis értékű szám annak érdekében, hogy elkerüljük a 0-val való osztást.

Értelemszerűen a tensorflow deep learning könyvtár beépítetten tartalmazza a Adam optimalizációs algoritmust, ezért ennek a leprogramozására nincs szükség.



3.8. ábra. Tanítási függvény optimalizálása különböző tanítási ráták esetében

3.4. A tanítási módszer mozgóképek esetében

3.4.1. Naiv megközelítés

Eddigiekben bemutatásra készült miképpen vihető át a stílus egyik statikus képről a másikra. mindenki tisztába van azzal, hogy egy videó statikus képek sorozatából tevődik össze, amiket képkockáknak (frame) nevezünk. Ennek megfelelően a naiv megközelítés az lenne, hogy ha adott egy videó, akkor vágjuk azt darabokra, pontosabban statikus képkockákra, majd az összes képkockára alkalmazzuk a stílus átruházási módszert.

A bemeneti videót frame-ekre való bontásához az ffmpeg nyílt forráskódú alkalmazást használtuk. Az ffmpeg parancssorból futtatható, a következő paraméterek szükségesek, ahhoz, hogy a mozgóképet statikus képek sorozatává alakítsa:

```
ffmpeg -i <input_video_path> -f image2 frame%05d.<ext>
```

A *frame%05d. <ext>* egy mintát jelent ami szerint a képkockák elnevezését határozza meg. Az *<ext>* a képkockák kiterjesztését jelenti.

Miután a stílus átruházása az összes képkockára megtörtént, vesszük az összes stilizált képkockát és felépítjük belölük a kimeneti videót. Ez ugyancsak megoldható az ffmpeg alkalmazással.

```
ffmpeg "-i" frame%05d.<ext> output_name.<ext>
```

A naiv megközelítés működik, a kimenet egy olyan videó lesz, amely tartalmazza a bemeneti stíluskép jellegzetességeit. Ugyanakkor észlelhető, hogy az eredmény nem valami esztétikus. Észlelhető, hogy a képkockák közötti átmenet nem folyamatos. Emellett különböző zajokat, úgynevezett "artifact"-eket észlelhetünk.

3.4.2. A haló inicializálása képkockák esetében

Amint említettük, a háló inicializálására egy fehér zaj képet használtunk statikus bemeneti képre való stílusátruházáskor. Több képkocka esetében ezzel az a gond, hogy a képkockák nem fognak egyanabba a lokális minimumba konvergálni tanításkor. Végeredményképp az átmenet egyik képkockáról a másikra nem lesz folyamatos.

Egy megoldás kísérlet erre az lenne, hogy a ha egy képkocka stilizálása történik, a hálót ne fehér zaj keppel inicializáljuk, hanem az előtte levő stilizált képkockával. Ez simább átmenetet fog eredményezni abban az esetben, ha a képkockákon nincs mozgás. Ennek a módszernek a használata szükségszerű viszont nem elégsges ahhoz, hogy szemnek is kellemes eredményt kapunk.

3.4.3. Optical flow bevezetése

Az optical flow egy mozgóképen megjelenő objektum különböző sebességvektorait jelenti. A képet képkocka közötti optical flow megesztimálásával mérhető az adott objektum valós mozgási sebessége. A kamerához távolabb levő objektumok kisebb sebességvektorokkal rendelkeznek, mint azon objektumok amelyek ugyanazon sebességgel mozognak, viszont közelebb vannak a kamerához[34]. Az optical flow meghatározásához a DeepMatching - Deepflow algoritmust használtuk. Ennek a kimenete egy .flo kiterjesztésű fájl ami tartalmazza adott pixelek esetében a horizontális és vertikális elmozdulás vektorokat.

Az optical flow meghatározásával egy mozgóképen meghatározhatók azok a régiók, amik statikusak (adott időn belül az illető régióban nincs változás, úgymond nincs mozgás) valamint azokat a régiókat ahol változás történik. Ami még fontos nekünk meghatározni egy adott mozgó objektum/személy esetében azt az kétdimenziós intervallumot, amelyben mozog. Meghatározzuk a mozgási határait (lásd:). Ennek a metódusát Sundaran és társai írják le a ... publikációjukban.

Két egymást követő képkocka esetében meghatározható az optical flow előre, tehát az képkockák időrendi sorrendjében, valamint meghatározható visszafele, az időrendi sorrend ellentett irányában. Jelölje $w(i, j)$ az előre történő optical flow-t, valamint $\hat{w}(i, j)$ a visszafele történő optical flow-t. Ekkor felírható a w torzítása a w alapján:

$$\tilde{w}(i, j) = w((i, j) + \hat{w}(i, j)) \quad (3.10)$$

A mozgás nélküli zónák meghatározhatóak a következőképpen:

$$|\tilde{w} + \hat{w}|^2 > 0.01(|\tilde{w}|^2 + |\hat{w}|^2) + 0.5 \quad (3.11)$$

A mozgási határok pedig meghatározhatóak:

$$|\nabla \hat{u}| + |\nabla \hat{v}| > 0.01|\tilde{w}|^2 + 0.002 \quad (3.12)$$

A 3.11 és 3.13 képletek alapján alkotható egy olyan mátrix, ami tartalmazza a mozgás nélküli zónákat valamint a mozgási határokat. Ezt a mátrixot átalakítjuk olyan módon, hogy a mozgási határokra 1-est rakunk, a többi érték 0-ás lesz, jelöljük ez a mátrixot $c^{(i-1,i)}$ -vel az $i-1$ és i -dik képkocka között. Ezt felhasználva felírhatjuk egy újabb veszteség függvényt:

$$L_{temporal}(x, w, c) = \frac{1}{D} \sum_{k=1}^D c_k \cdot (x_k - w_k)^2 \quad (3.13)$$

3.5. A rendszer tervezése és kivitelezése

3.5.1. A rendszer interakciós és viselkedési modellje

A rendszer interakciós modellje (3.9. ábra) a lehető legegyeszerűbb. Egyetlen aktort határozunk meg, aki maga a felhasználó. A rendszer esetében nem beszélhetünk bármilyen adminisztrációs felületről mivel nem rendelkezik semmiféle háttértárral valamint nem pillanatnyilag nem definiál semmifel hálózati komponenst.

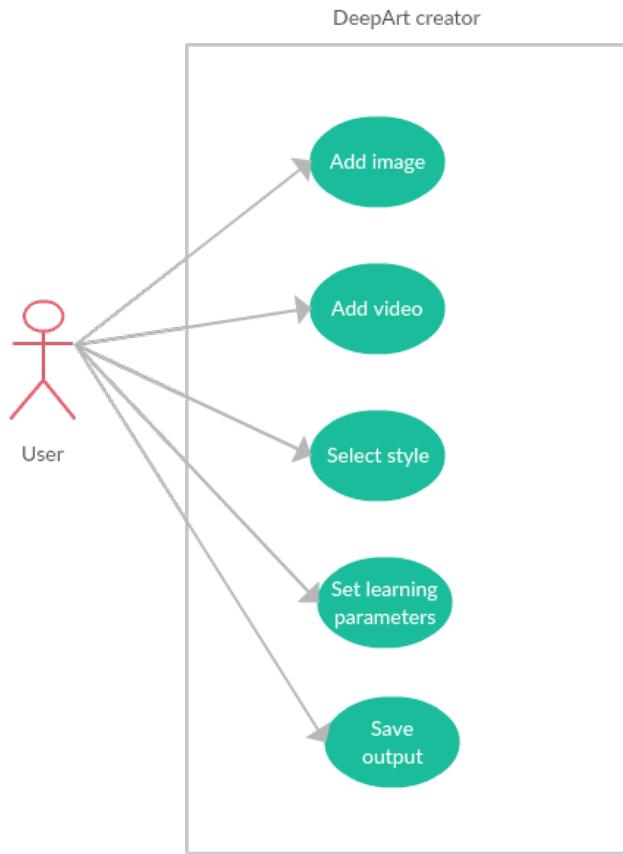
A felhasználónak lehetősége van új kép illetve videó anyag bevitelére. Ez a funkcionálitást egyazon komponens végzi aminek az egyik feladata eldönteni, hogy a kijelölt bemenet kép vagy videó típusú. A felhasználónak lehetősége van a meghatározott kiterjesztésű fájlok közül választani bemenethez.

A felhasználónak lehetősége van a rendszer által biztosított stílusok közül választani. A felhasználó nem határozhat meg új stílust. A rendszer nem biztosít erre adminisztrációs felületet.

A felhasználó meghatározhatja az átruházási folyamat tanítási paramétereit. A rendszer erre biztosít alapértelmezett értékeket, amiket a felhasználó változtathat. A rendszer megjegyzi az új paramétereket, ezek új indításkor is megmaradnak. A rendszer az alapértelmezett paramétereket is megjegyzi, ezeket a felhasználó bármikor visszaállíthatja az erre kifejlesztett opció használatával.

A felhasználó meghatározhatja az útvonalat ahova a kimenet le lesz mentve. A mentést a rendszer automatikusan el fogja végezni az átruházási folyamat után.

A rendszer interakciós modelljét az 3.1. ábra tartalmazza. A felhasználó kiválaszt egy bemeneti állományt a rendszer által meghatározott kiterjesztések közül (.jpg, .png, .gif,



3.9. ábra. A rendszer viselkedési modellje

.avi, .mp4), majd kiválaszt egy stílust a megadottak közül. A rendszer annak függvényében, hogy milyen bemenetet adtunk, eldönti, hogy kép vagy mozgóképpel kell dolgozni. Kép esetén lefuttatja a 3.2-es fejezetben ismertetett tanítási algoritmust amely során az átruházza a művészeti kép stílusát. Mozgókép esetén képkockákra bontja azt, majd alkalmazva lesz a 3.3-as fejezet tanítási metódusa. Ha összes képkocka szerkesztve lett, akkor a rendszer felépíti a képkockákból a kimeneti videót. Végezetül a rendszer lementi a megadott helyre a kimeneti állományt.

3.5.2. A rendszer strukturális modellje

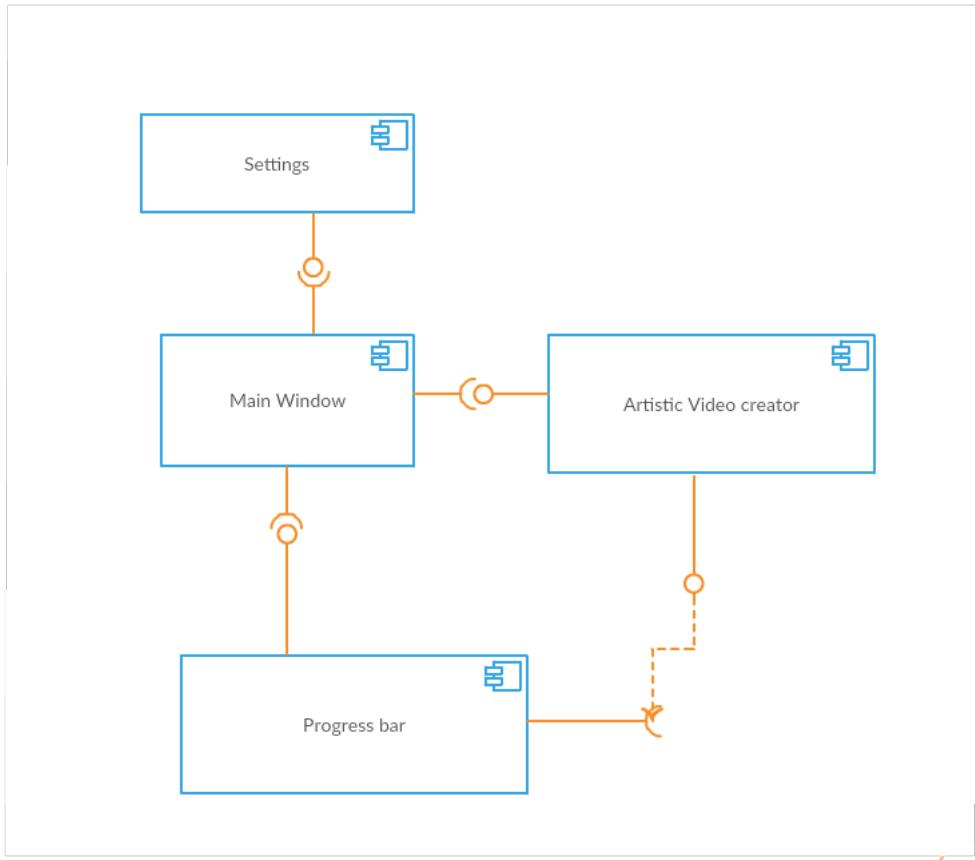
A rendszer a következő komponensekből tevődik össze (3.10. ábra):

- Main Window: az applikáció fő ablaka, ez az ablak jelenik meg az applikáció indításakor, valamint erről az ablakról lehet kiválasztani a bemeneti adatot és a stílus képet. Az ablak Start gombjának a lenyomásával indítható az átruházási folyamat. Az ablak tartalmaz két beágyazott területet, ahol a bemeneti adat, valamint a végeredmény tekinthető meg.

- Settings: a főablak menüsorából (Menu Bar) jeleníthető meg. Fő feladata egy grafikus felület biztosítása a különféle beállítások és tanítási paraméterek megadásához. A felhasználó megadhatja az útvonalat ahoz a mappához, ahova a kimenet lesz tárolva, megadhatja a bemeneti VGG19 háló útvonalat valamint a tanítási paramétereket.
- Progress bar: a feladata kimutatni a felhasználó számára, hogy az átruházás milyen státusban van, mennyi van még hátra amig a folyamat befejeződik. Erre azért van szükség, mivel az átruházási folyamat időigényes, ezért fontos a felhasználó tudtára adni, hogy milyen állapotban van a folyamat. A folyamat befejeztekor egy jelzést küld a Main Window komponensnek, ami ki fogja jelezni a felhasználónak a végeredményt.
- Artistic Video creator: ez képezi az applikáció magját, ez az a komponens, ami elvégzi maga az átruházási folyamatot. A bemeneti paramétereket/adatokat a Main Window komponenstől kapja. Az átruházási folyamat a videó kártyán fog futni, de ettől függetlenül a komponens jelzéseket fog küldeni a Progress bar komponensnek. Ehhez a komponenshez tartoznak a következő modulok:
 - Képkezelő: feladata a bemeneti képek beolvasása és kiíratása a merevlemezre. Beolvasáskor egy preprocesszálás műveletet végez, majd kimenetkor egy posztprocesszálás művelet lesz elvégezve. Fontos megjegyezni, hogy ezt a modult a processzálások miatt nem ajánlott használni a komponensen kívül.
 - Videó kezelő: feladata a bemeneti videot képkockákra vágni és a kimeneti képkockákból videot készíteni. A képkockák egy temporális folderbe lesznek elmentve vágás után, ezeknek a beolvasását a képkezelő modul végzi majd.
 - VGG19 kezelő: a modul feladata ez előre betanított és kimentett VGG19 háló beolvasása és átalakítása egy olyan hálóvá amit a Tensorflow könyvtár fel tud dolgozni majd.

3.5.3. Többszás megoldás és kommunikáció a komponensek között

Egy bemeneti képre való stílus átruházás időigényes folyamat. Mozgókép esetében ez a folyamat időigénye lineárisan növekedik a képkockák számának szorzatával. Annak érdekében, hogy az alkalmazásunk rezponzív legyen elhagyhatatlan aspektus egy olyan modell kialakítása ami igénybe veszi a modern processzorok többszásas működési tulajdonságát. Az Artistic video creator komponens működése során ugyan igénybe veszi a videókártya számítási kapacitását, de ez semmiképp sem jelenti azt, hogy a futási idő elhanyagolható. Ugyanakkor fontos kiemelni azt is, azon függvényhívások sorozata ami a



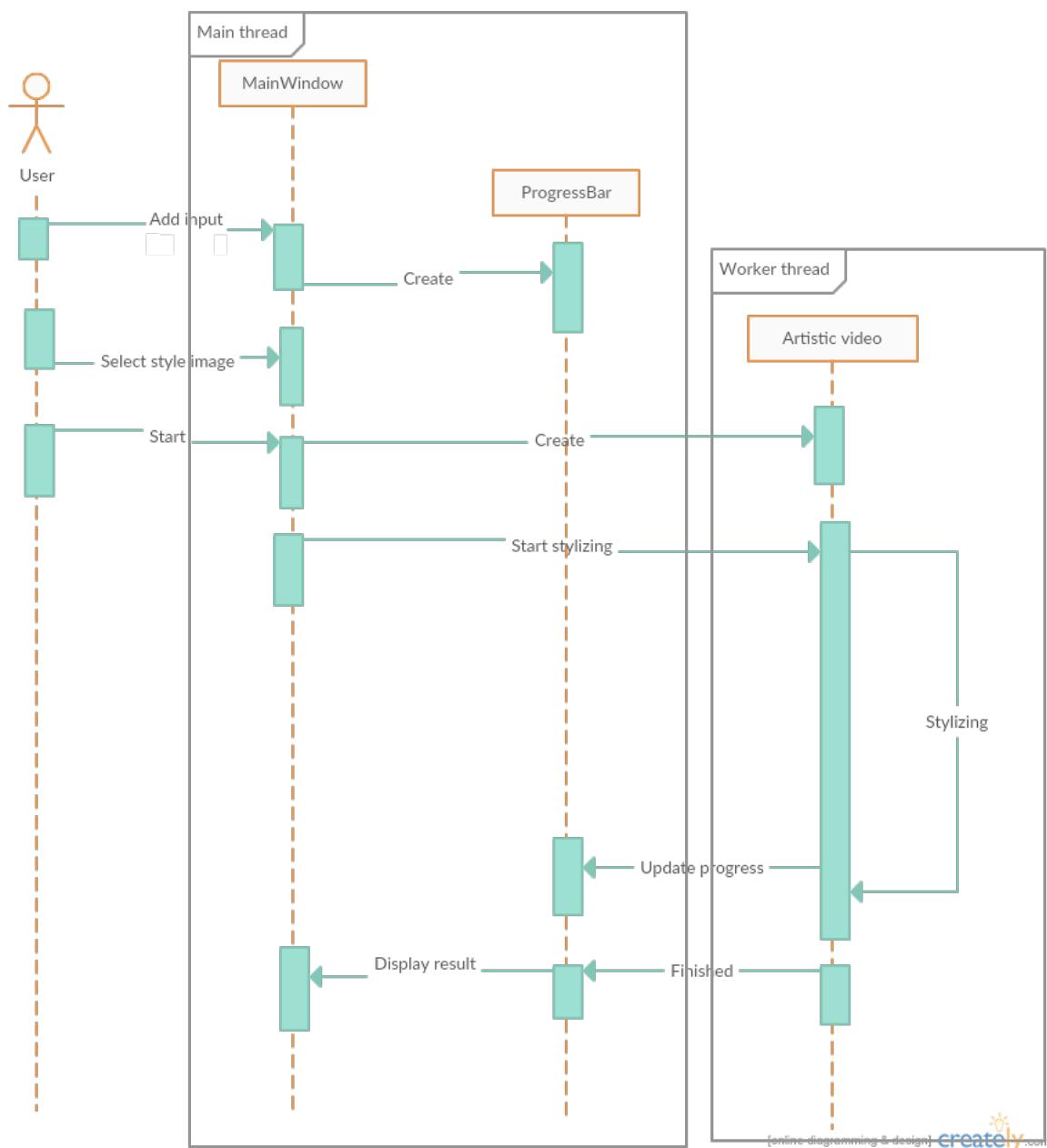
3.10. ábra. A rendszert alkotó komponensek

videókártyán fog elvégződni, szinkron módon történik. Valójában a programunk meghívja az adott függvényt majd addig várakozik, amíg megoldás nem érkezett erre.

Alkalmazásunk esetében megkülönböztetünk egy főszálat valamint egy mellékszálat (3.11. ábra). A főszálon fog futni a Main Window és a köréje csoportosuló felhasználi felülettel rendelkező komponensek, mint például a Settings vagy a ProgressBar. Az alkalmazás indításakor csak a főszál indul el ami létrehozza a Main Window és Settings komponenseket. Ezt követően, ha a felhasználó megadja a bemeneti állományt, létrejön a ProgressBar komponens. A ProgressBar azért jön létre csak ebben a pillanatban, mivel a rendszernek el kell döntenie a bemenet típusát és ennek függvényében egy factory modell segítségével dönti el, hogy milyen típusú komponens jön létre. Ebben a pillanatban azonban a Progressbar komponens még nem látható a felhasználó számára.

Ha felhasználó úgy dönt, hogy a megfelelő bemenetet választotta ki, elindíthatja az átruházási folyamatot. Ekkor egy külön szál jön létre (Worker thread), ami átveszi a bemeneti adatokat és elkezdi a folyamatot. A terhelés ebben a pillanatban átkerül a másodlagos szálra, ennek eredményeképp a főszál nem fog blokkolódni. Így az felhasználói felület aktív marad a felhasználó számára és ki fogja tudni szolgálni annak kéréseit. A

munkafolyamat közben az Artistic Video creator komponens kapcsolatban áll a Progress-Bar komponenssel és jelzéseket küld a munkafolyamat státusáról így ez ki tudja jelezni, hogy mennyi munka van még hátra. A munkafolyamat bármikor leállítható még azelőtt, hogy az átruházás befejeződött volna. Ilyenkor a ProgressBar üzenetet küld az Artistic Video creator komponensnek, ami beállít egy saját leállításra szánt flag-et. A folyamat két iteráció között állítható le, ha az illető flag be volt állítva. Miután a munkafolyamat sikeresen leállt, egy jelzés fog érkezni a ProgressBar komponenshez ami majd nyugtázni fogja. Ha a munkafolyamat bevégeződött, vagy a felhasználó által megszakításra került, a másodlagos szál el fog halni, az ezáltal igényelt erőforrások pedig fel fognak szabadulni.



3.11. ábra. Átruházás szekvencia diagrammja

3.5.4. A rendszer implementálása

A rendszer implementációja Python környezetben történt. Ahogy már a bevezetőben is említésre került, az alábbi könyvtárak kerültek felhasználásra: numpy, Tensorflow, opencv, PyQt. Az elkövetkezőkben ki szeretnék térti a PyQt könyvtár bemutatására ugyanis a szoftver jelentős része ennek a felhasználásával valósult meg és az implementációban használt egyes tervezési minták a PyQt sajátosságait próbálják kiaknázni.

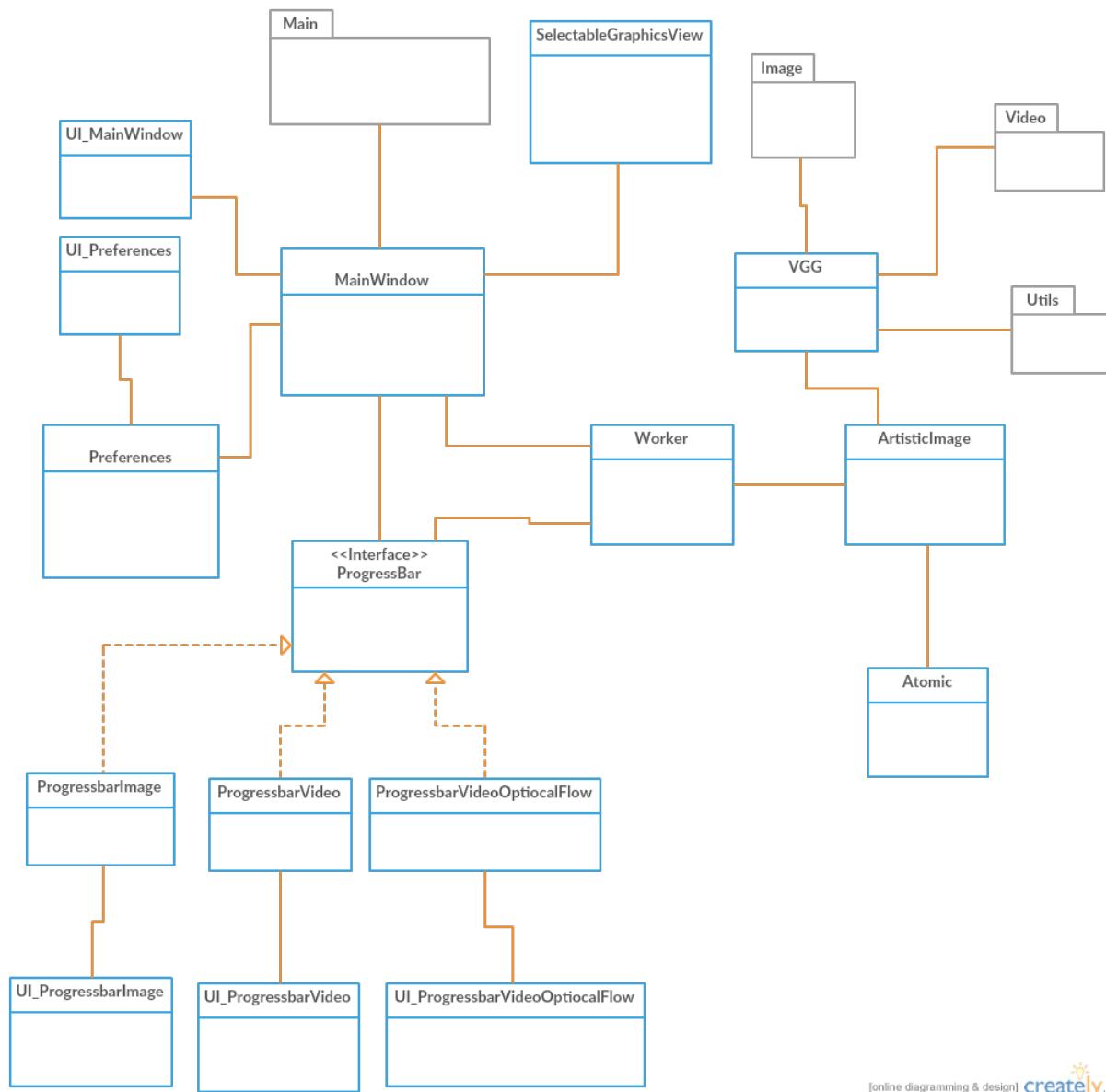
A PyQt a híres Qt könyvtárra épül. A Qt egy C++ könyvtár, aminek célja olyan grafikus szoftverek megvalósítása amik különböző platformokat is támogatnak. A Qt univerzális megoldást ad a grafikus felületek elkészítésére, de emellett számos más komponenseket is implementál, mint például tömbök, listák, konténerek, párhuzamos szálak, smartpointerek, hálózati protokollok, stb. A Qt egyik újítása a SIGNAL-ok és a SLOT-ok bevezetése. Ennek működése egyszerű, egy adott osztály adattagjai és metódusai mellett deklarálhatók SIGNAL-ok és SLOT-ok is. Egy osztály egyes metódusai jelzéseket adhatnak ki SIGNAL-ok segítségével amire a SLOT metódusok csatlakozhatnak. Egy adott SIGNAL kibocsátásakor a rá csatlakozott összes SLOT függvény le fog futni. Például egy grafikus felületen levő gomb lenyomásakor kibocsájtunk egy SIGNAL-t amire a neki megfelelő SLOT függvény meghívódik és lefut. A jelzéskibocsátás működik többszásas program esetében is, az is megoldható, hogy egy adott jelzésre egy más szalon futó metódus válaszoljon. Természetesen a SIGNAL-SLOT megoldás a PyQt esetében is tökéletesen működik és egy nagyon kényelmes megoldást biztosít a grafikus felület és a business logika összekötésében.

A PyQt egy másik előnye az, hogy elegánsan megoldható a modell-nézet-kontroller szoftverfejlesztési minta. Maga a nézet megvalósítható XML leíró nyelv segítségével. Ebből automatikusan generálható egy Python osztály, ami fel fogja építeni maga a kinézetet. Ezt a kontrollerbe be lehet integrálni kompozícióval majd a SIGNAL-SLOT-ok összekötése után használni lehet azt.

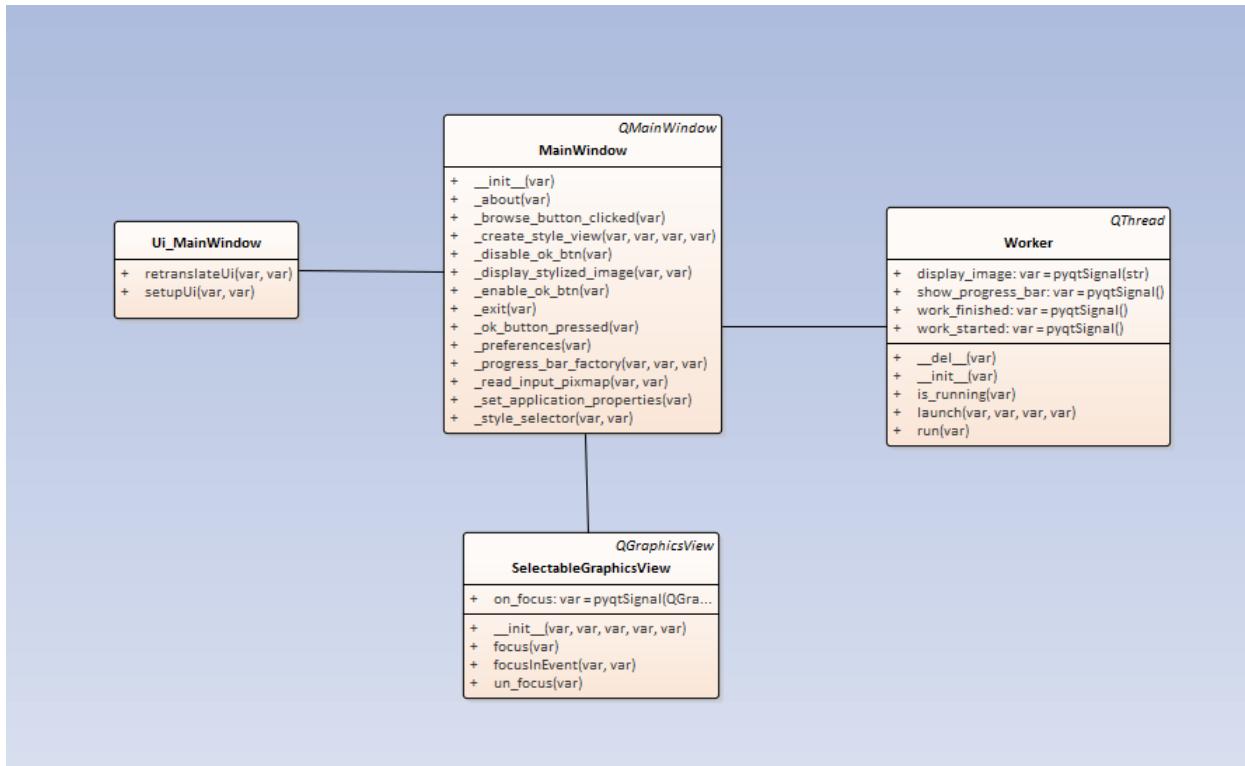
A továbbiakban részletes bemutatásra és elemzésre kerül az applikációnk objektum orientált megvalósítása (3.12. ábra).

3.5.4.1. A főablak (Main Window) komponens implementálásának bemutatása

A MainWindow komponens magába foglalja a fő felhasználói implementációját meg pár ehhez szorosan kapcsolódó segítő osztályt(3.13. ábra). Maga a fő ablak megvalósítását két részre bontottuk: maga a nézettel foglalkozó osztály(Ui_MainWindow) és maga a back-end logikával foglalkozó osztályra (MainWindow). Az Ui_MainWindow egyetlen feladata maga felhasználói felület nézetének definiálása, ezért ezt kompozícióval integráljuk a MainWindow osztályba. A felület (lásd: 3.14. ábra) rendelkezik két grafikus kép kijelzésére szánt ablakkal, amik QGraphicsScene típusúak. Ezek közül a bal ablakba fog



3.12. ábra. A rendszer osztálydiagrammja



3.13. ábra. MainWindow

megjelenni a bemeneti kép amit a felhasználó megad. A QGraphicsScene sajnos nem képes videóanyag beágyazására, ezért az alkalmazás egy előre megadott képet fog ilyenkor kijelezni a felhasználónak.

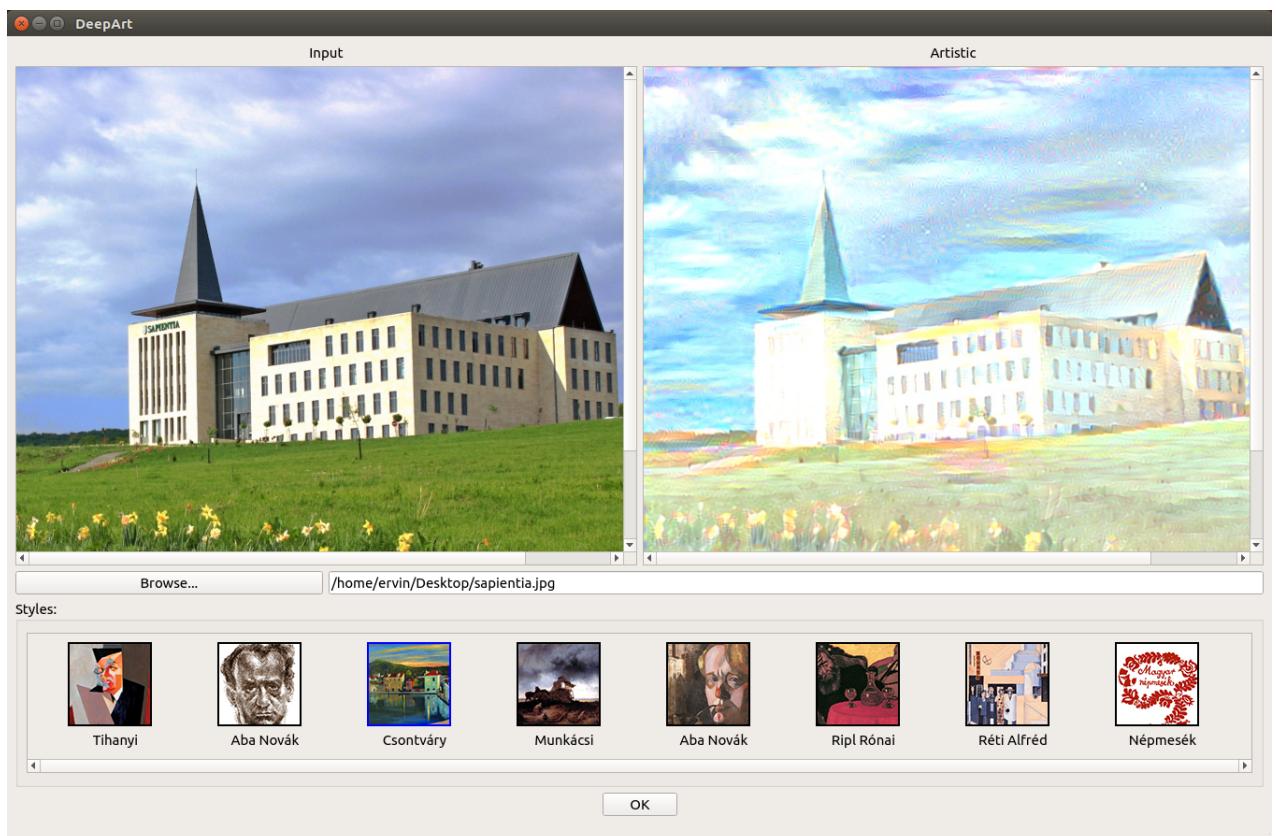
A bemenet magadása a "Browse..." nyomógomb lenyomásával történik, ami jelzést fog küldeni a `_browse_button_clicked` metódusnak. Ez létre fog hozni QFileDialog típusú beépített fájl böngésző ablakot. Ez fel van konfigurálva olyan módon, hogy csak az előre kijelölt típusú fájlkiterjesztéseket lehessen beolvasni ("Images/Videos (*.png *.jpg *.gif *.mp4)"). Ha fájlkiválasztás megtörtént, akkor a fájlnak az útvonala megjelenik a "Browse..." nyomógomb melletti szövegdobozban.

A MainWindow belső osztályként tartalmazza a SelectableGraphicsView osztályt. Ennek célja a stílusnépek közötti váltogatás megoldása(3.13. ábra). Az alkalmazás összes előre definiált stílusnépe látható a 3.15. ábrán. Alapértelmezetten a PyQt nem tartalmaz ilyen típusú előredefiniált osztályt, ezért ezt külön le kellett implementálni. A SelectableGraphicsView a QGraphicsView beépített PyQt osztályból származik. Ez azért került választásra mivel a QGraphicsView alkalmas bármiféle statikus grafikus elem beágyazására és ugyanakkor egy olyan területet kínál ami, ha nem fér ki teljesen a képernyőn, akkor scroll-ozni lehet. A SelectableGraphicsView biztosítja, hogy az alkalmazás futásának minden pillanatában egy stílus ki van jelölve. A felhasználó váltogathat ezen stílusok között. A SelectableGraphicsView típusú objektum minden váltogatás esetében jelzést fog küldeni a MainWindow objektumnak, ami lementi azt, hogy az épp kijelölt stílus bemeneti

stílus képek a merevlemezen hol található.

A főablak egyik fontos objektuma a Worker típusú objektum. Ez a típus a QThread osztály leszármazottja, ami a párhuzamosításért felelős. A Worker típusú objektum felüldefiniálja a QThread run metódusát ami külön szálon fog majd lefutni. Ebben a metódusban lesz létrehozva az ArtisticVideo objektum ami a stílusátruházást végzi. Kezdetben a Worker objektumot a MainWindow inicializálja, az inicializálás még a főszálon történik. Miután ez megtörtént, a főablakon levő "Start" nyomógomb lenyomásával, meghívódik a run metódus. A Worker objektum kapcsolatban áll a MainWindow és a ProgressBar típusú objektumokkal is, amiknek különféle jelzéseket tud küldeni (work_started, work_finised, show_progress, display_image).

Ugyancsak a MainWindow tartalmazza a ProgressBar létrehozásával felelős factory típusú metódust. Amint említettem, a felhasználó "Browse..." lenyomásával beolvashat statikus képet vagy mozgó kép típusú fájlt. Ezekre az átruházás különböző folyamatokat vesznek igénybe. Ahhoz, hogy a felhasználó részletesebb tájékoztatást kapjon arról, hogy mi folyik a háttérben, ezért három típusú progress sávval rendelkező dialógus ablakot implementáltunk le, amik egy közös ősosztállyal rendelkeznek.



3.14. ábra. A rendszer felhasználói felülete



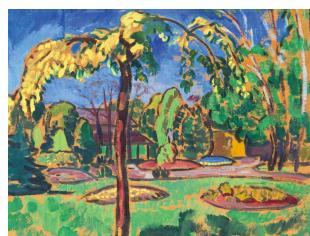
(a) Aba Novak Vilmos - Ön-



(b) Aba Novak Vilmos - Selfportrait



(c) Csontvary Kosztka Tivadar - Traui tájkép naplemente idején



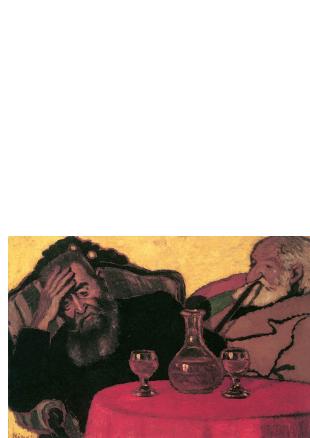
(d) Iványi Grunwald Béla - Parkrészlet Kecskeméten



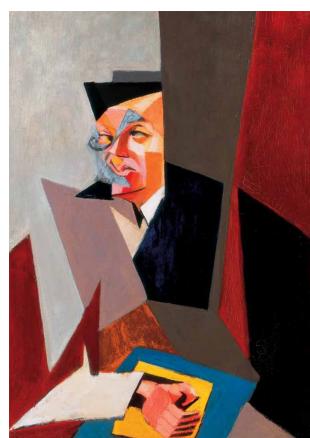
(e) Munkácsi Mihály - Vihar a pusztán



(f) Réti Alfréd



(g) Ripl-Rónai József - Apám és Piacsek bácsi vörösbor mellett

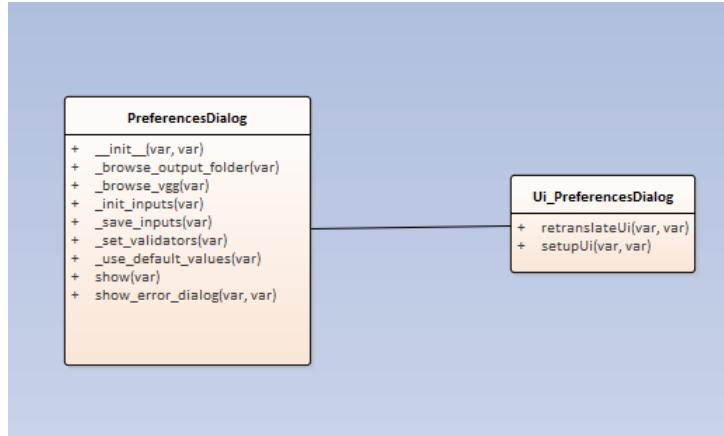


(h) Tihanyi Tzara



(i) Magyar Népmesék

3.15. ábra. Stílusképek



3.16. ábra. A beállítások komponens osztálydiagrammja

3.5.4.2. A beállítások (Settings) komponens implementálása

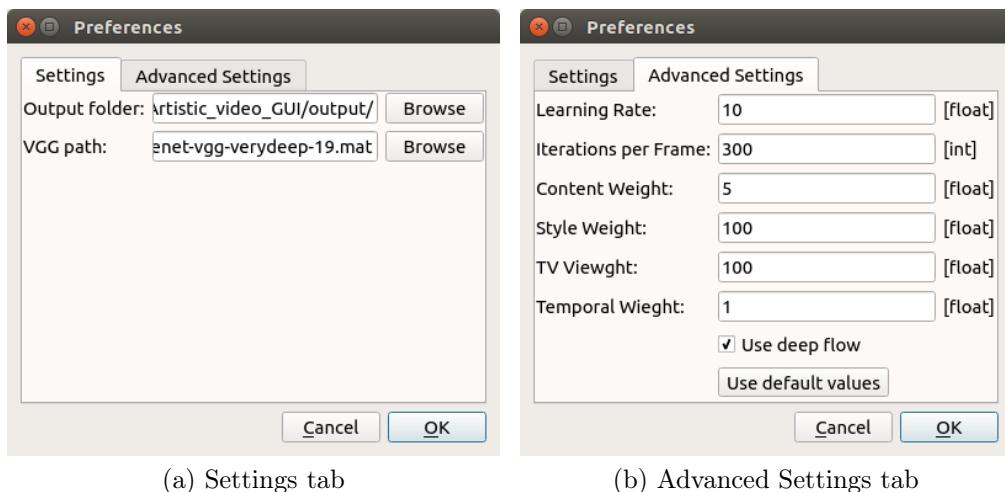
A beállítások komponens két objektumot tartalmaz, egyik a PreferencesDialog típusú objektum és az ebbe beágyazott nézetért felelős UiPreferencesDialog objektum. Értelemszerűen a PreferencesDialog tartalmazza a működési logikát.

A PreferencesDialog a beépített QDialog osztály leszármazottja, örökli ennek tulajdonságait. Szülő objektumként természetesen a MainWindow objektumot kapja meg, ami Qt-ben annyit jelent, hogy ha a szülő objektum elhal, akkor fel fogja szabadítani a gyerek objektumot is, tehát valójában a szülő objektum felelős a gyerek objektum életciklusáért. A beállítások dialógus két tab-ot tartalmaz:

- Settings: itt meg lehet adni a foldert ahova a kimenet automatikus mentve lesz. Ugyanitt meg kell adni az előre betanított háló útvonalát.
- Advanced Settings: itt a tanítással kapcsolatos paramétereket lehet szabályozni. Ilyen paraméterek:
 - Learning rate - tanítási együttható
 - Iteration per Frame - egy adott képkocka esetében a maximális tanítási iterációk száma;
 - Content Weight - súlyzó, a bemeneti kép veszteségfüggvénye esetében használatos;
 - Style Weight - súlyzó, a stílus veszteségfüggvényének az együtthatója;
 - Temporal Weight - súlyzó, a mozgóképek esetében bevezetett veszteségfüggvény együtthatója.
 - Use deep flow - egy flag-et állít, ezzel kikapcsolható az időigényes deep flow módszer használata, cserébe gyengébb minőségű videó lesz az eredmény.

Az beállítások komponens tárolja az előre megadott (default) értékeket a paraméterek esetében, ugyanakkor az is el lesz tárolva amit felhasználó módosít ezeken. A módosított értékek megmaradnak akkor is ha az alkalmazást bezárjuk majd újraindítjuk. Erre beépített QSettings objektum használtuk, ami megoldja az beállítások tárolását és visszaolvasását. A QSettings kulcs-érték táblákba tárolja az paramétereket. Ezek majd az applikációnk bármelyik objektumából elérhetőek, ha példányosítjuk a QSettings osztályt és megadjuk az adatbázisunk nevét ahova a paraméterek mentve lettek.

Az alapértelmezett beállítások bármikor visszállíthatók a "Use default values" gomb lenyomásával ugyanis ezek a paraméterek hardcode-olva vannak az applikáción belül.

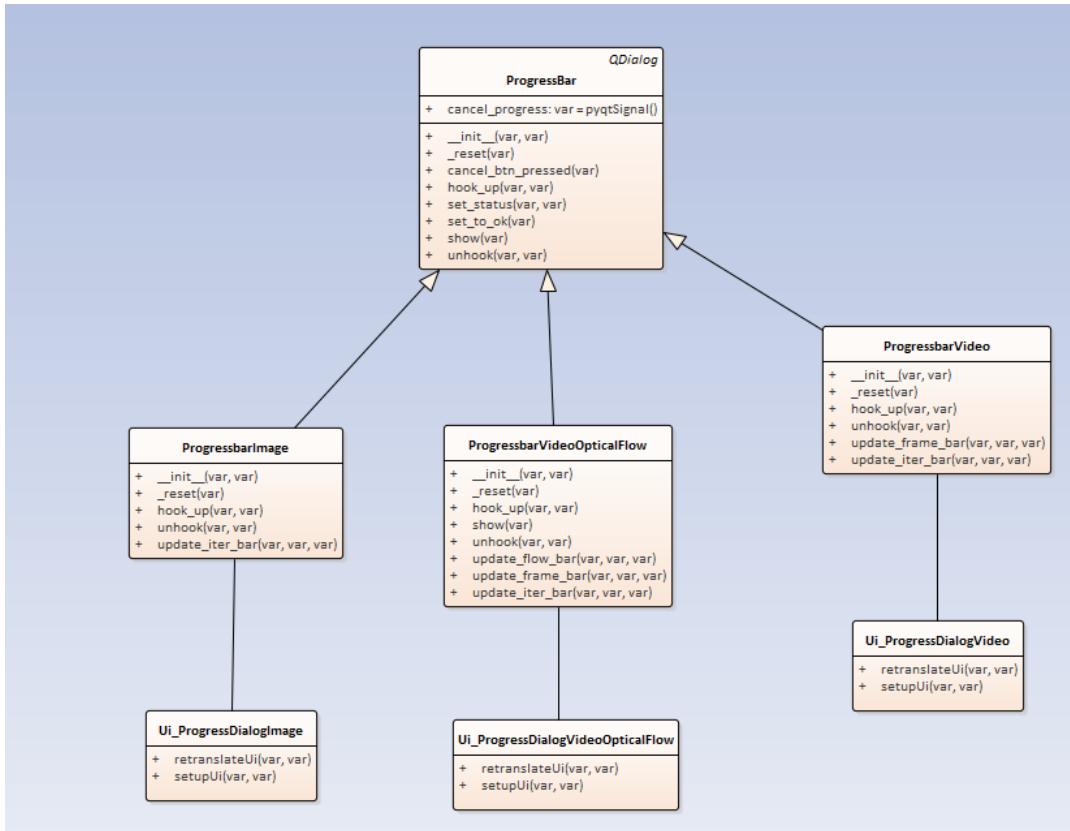


3.17. ábra. Beállítások dialogus ablak

3.5.4.3. A progress sáv (Progress bar) komponens implementálása

A progress sáv komponens feladata kimutatni a felhasználó számára a hátralevő munkamennyiséget. Statikus képek esetében kimutatásra kerül az, hogy az össziterációk számának hány százaléka volt elvégezve. Mozgóképek esetén emellett kimutatásra kerül, hogy a képkockák közül hány százalékre volt stílus átruházva az összes képkockákból. Ha deep flow metódus is használatra kerül, akkor ennek is egy külön sáv fogja valós időben kijelezni, hogy hány képkocka van még hátra, amire alkalmazni kell a deep flow metódust. Ezekből következtethető, hogy három típusú ProgressBar osztály példányosítható, amelyek az alábbiak:

- ProgressBarImage(3.19. ábra)
- ProgressBarVideo(3.20. ábra)
- ProgressBarVideoOpticalFlow (3.21. ábra)



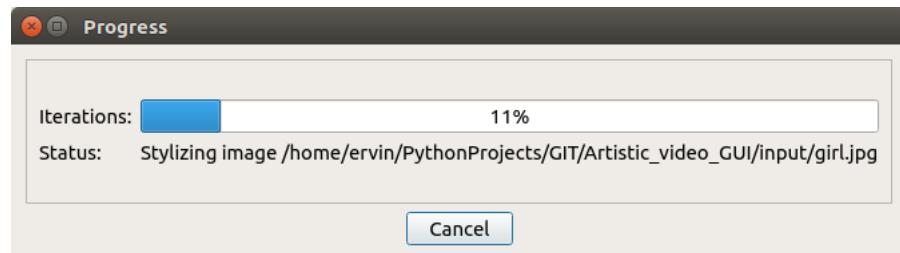
3.18. ábra. A progress sáv komponens osztálydiagrammja

Mind a három egy közös ősosztályból öröklődik, ez a sima ProgressBar osztály. A ProgressBar osztály a beépített QDialog-ból öröklődik. A QDialog előnye, hogy be meg lehet határozni, hogy amikor a dialógus ablak fókuszban van, akkor ne lehessen interakcionálni a szűlő osztály interfészével. Ez fontos aspektus, mivel amikor maga a stílus átruházás történik, akkor nem szeretnénk, ha a felhasználó a fő ablakkal interakciónáljon, esetlegesen egy másik átruházási folyamatot indítson. Ez az applikáció összeomlásához vezetne.

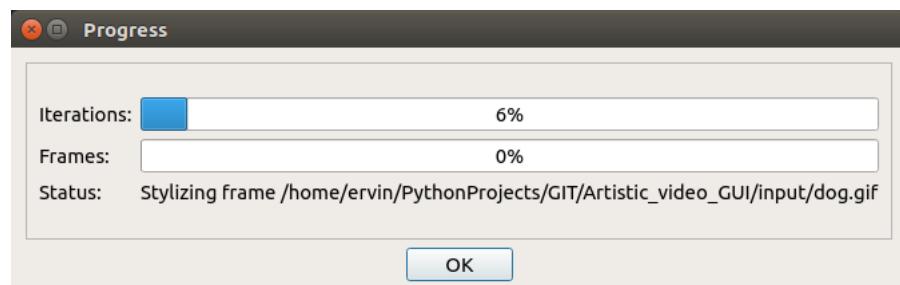
Ahogyan már a MainWindow esetében is bemutatásra került, a ProgressBar típusú osztályok létrehozása a factory modell segítségével történik. Az hogy melyiket fogjuk példányosítani, az a bemeneti állomány típusától, valamint a beállításoktól függ. A ProgressBar ősosztály SLOT metódust tartalmat a "Cancel\OK" nyomógomb lekezelésére, és a státus üzenet kiíratására, amit a többi származtatott osztály is örökölni fog. A videóval kapcsolatos ablakok emellett SLOT függvényeket definiálnak annak érdekében, hogy ki tudják jelezni hány képkocka van még hátra.

3.5.4.4. A stílusátruházó (Artistic video creator) komponens implementálása

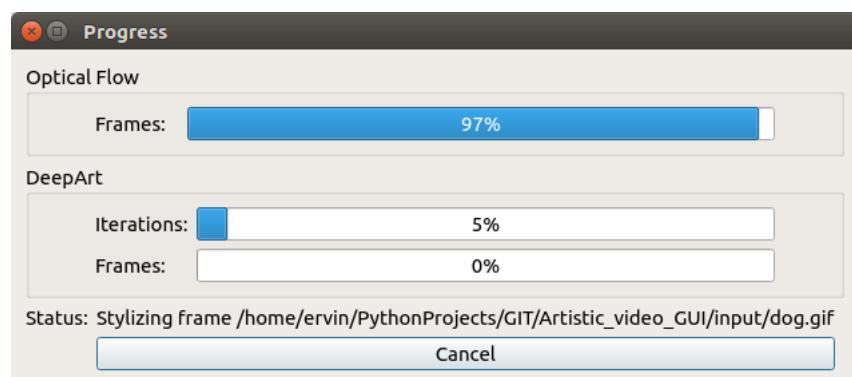
A stílusátruházó komponenst az ArtisticVideo, VGG19 és AtomBoolean osztályok alkotják (3.22. ábra). Ezek körül az ArtisticVideo osztály felelől a stílus átruházásával, a



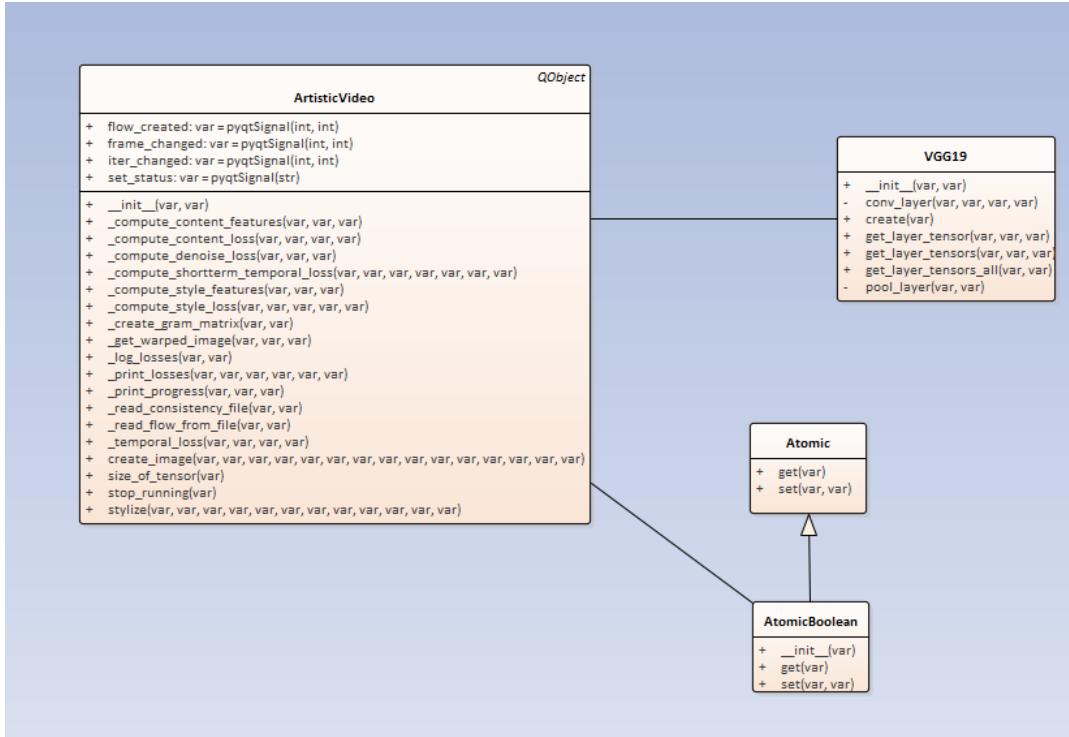
3.19. ábra. Progress sáv statikus kép esetében



3.20. ábra. Progress sáv videó esetében (deepflow kikapcsolva)



3.21. ábra. Progress sáv videó esetében (deepflow bekapcsolva)



3.22. ábra. A stílusátruházó (Artistic video creator) komponens osztálydiagrammja

VGG19 osztály felelős az előredefiniált neuronháló beolvasásával. Az AtomBoolean egy segítő osztály, több szálról is módosítható flag-et definiál. Ezek mellett komponens még használ két csoport segítőfüggvényt amikor az Image meg a Video állományokba csoporthosztottuk.

Az ArtisticVideo osztályt a Worker objektum példányosítja. Mivel a Worker objektumunk a mellákszálóból fut, ezzért az ArtisticVideo objektumunk metódusai is a mellékszálon fognak elvégződni. Az ArtisticVideo-nak két fontos metódusa a create_image és a stylize. A create_image megkapja a bemenet útvonalát, amit az Image vagy Video segítő függvénycsomagok segítségével beolvas és feldolgoz. Amiután ez megtörtént, egyenként minden képkockára meghívódik a stylize metódus. A stylize metódus tartalmazza a tanítási algoritmus. Ennek a függvény eredménye minden esetben egy stilizált kép, kivételes eset, ha a tanítás megszakad, ekkor a visszatérített érték a null (None). Ugyancsak a create_image metódusra hárul a képkockákból videót alkotni.

Az ArtisticVideo jelzéseket küld a főablak és a progress dialógus komponenseknek. A Qt kimondja, hogy a felhasználói felület mindenkorban a fő szalon kell fusson ezért az ArtisticVideo nem oldhatja meg az, hogy függvényhívással módosítja a felhasználói felületet. Ez mindenkorban jelzéssel kell megoldani, kikötve azt, hogy a jelzésre válaszoló függvény a fő szalon fut. Az ArtisticVideo osztály által definiált jelzések:

- flow_created: akkor bocsájtódik ki amikor egy adott képkocka esetében a deep flow eljárás lefutott. Csak mozgóképek esetében bocsájtódik ki;

- frame_changed: akkor küldődik, amikor egy képkockára sikeresen át lett ruházva a stílus;
- iter_changed: egy kép esetében minden eltelt tanítási iteráció esetén kibocsájtódik;
- set_status: a progress dialógus ablakon levő státus üzenetet lehet állítani ezzel.

A VGG19 osztály a bemeneti hálót parszolja és felépíti az alkalmazásban használatos hálót. Mivel a háló mérete nagy (549MB), ezért a merevlemezről történő beolvasás egyszer történik. A háló a memóriában marad amíg szükség van erre. Ennek hátránya a memória foglalása, de ekkora memóriahasználat modern számítógépek esetében nem jelent gondot.

Ugyanehhez a komponenshez sorolhatunk két segítő (utility) függvényeket tartalmazza csomagot: az egyik csoport az Image, a másik a Video névre hallgat.

Az Image csomag tartalmaz egy imread függvényt statikus kép beolvasására és egy imwrite függvényt statikus kép mentésére. Beolvasáskor preprocesszálás történik a bemeneti adaton, ami azt jelenti, hogy a színcsatornákból kivonódnak a következő középértékek: [123.68, 116.779, 103.939]. Ez arra szolgál, hogy a pixelértékek 0-ás körüli értékeket vegyenek fel. Mentéskör ezek az értékeket hozzáadjuk a kimeneti kép színcsatornához.

A Video csomag tartalmaz egy convert_to_frames függvényt ami a megadott útvonalon levő videót képkockákká alakítja. Ezeket a képkockákat majd az imread-del fogjuk feldolgozni. Emellett tartalmazza a convert_to_video függvényt, ami a kimentett képsorozatot fogja videóvá alakítani. Továbbá tartalmazza a make_opt_flow és _run_consistency_check függvényeket, amik az optical flow és képkockák mozgási határait eredményezik. Mindkét esetben a függvény egy külső bináris állomány futtatásával végzi el a feladatot. Ezek a binárisok nem saját programkódok a készítők lefordított állományait használtuk fel[34].

3.6. Limitációk

Az implementáláshoz használt deeplmatching és deeplflow algoritmusok esetében a készítők által kiadt binárisokat használtuk. Ezek tökéletesen működnek Linux alapú rendszereken. Mivel a forráskódjuk nyílt, ezért megpróbáltuk portálni azokat Windows környezet alá, sajnos többszörös próbálkozás után kénytelenek voltunk feladni. A forráskód elég komplex és nem dokumentált. Mivel a dolgozat célja nem ez a két algoritmus köré alapul, ezért inkább abbahagyutuk azoknak az átírását. Ennek eredménye az, hogy a deepmatching és deepflow által biztosított funkcionálitások nem elérhetők, ha az alkalmazás Windows operációs rendszer alatt fut.

A deepmatching algoritmus is mesterséges intelligenciát használ, ezért itt is ajánlatos volna VGA megoldást használni. Sajnos a dolgozat elkészítésekor működő megoldás nem volt erre. A VGA-ás megoldás jelentőset gyorsítana az algoritmusokon.

4. fejezet

A rendszer tesztelése

A rendszer tesztelése alatt fel szerettük volna mérni, hogy a rendszer hogyan viselkedik a minden nap használatban. Többek között választ szerettünk volna kapni olyan kérdésekre, hogy a rendszer mennyi idő alatt ruházza át a stílust egy statikus képre, mekkora a memória igénye, mekkora a maximális képméret amire még működőképes.

4.1. A tesztrendszer ismertetése

A rendszer tesztelésére a következő konfigurációt használtuk:

- **Alaplap:** MSI Z170A-G45 GAMING
- **Processzor:** Intel(R) Core(TM) Skylake i7-6700 ("non-K") CPU @ 3.40GHz (Turbo Boost: 3.90GHz)
- **Videokártya:** GIGABYTE GeForce GTX 1080 G1 GAMING 8GB DDR5X 256-bit
- **RAM Memória:** Corsair Vengeance LPX Black 32GB DDR4 3000MHz
- **Merev lemez:** HyperX Savage SSD, 240GB, 2.5", SATA III
- **Operációs rendszer:** Ubuntu 17.04 LTS

A használt videókártya egy nVidia GeForce GTX 1080 volt, ennek is a Gigabyte által kiadott G1 GAMING változata. A kártya főleg játékra van tervezve, de kiemelkedő hardver tulajdonságainak megfelelően tökéletesen használható párhuzam gépi tanulásra is. A videókártya a Pascal architektúrával rendelkezik. A kártya alapfrekvenciája 1695 MHz, terhelés alatt felugorhat 1860MHz-re. A kártya 2560 CUDA maggal rendelkezik valamint 8GB DDR5X típusú memóriával, amiből gépi tanulásra használható 7860MB.

A rendszer a legújabb Linux Debian alapú Ubuntu operációs rendszeren futott. A rendszer egyes limitációkkal támogatja a Microsoft Windows operációs rendszert is, viszont a limitációk miatt az összes teszt nem végezhető el azalatt, ezért az Ubuntu-ra esett

választás. A operációs rendszerünk az ekkori legújabb nVidia driver-ekkel volt ellátva ami a 375.66 változat. A Pascal architektúrának támogató CUDA 8.0-ás videókártya fejlesztő könyvtárat használtuk. Emellett a Tensorflow igényeli a cudnn videó kártya memóriájával foglalkozó könyvtárat is.

4.2. Egy képkockára történő stílusátruházási idő és memória igénye

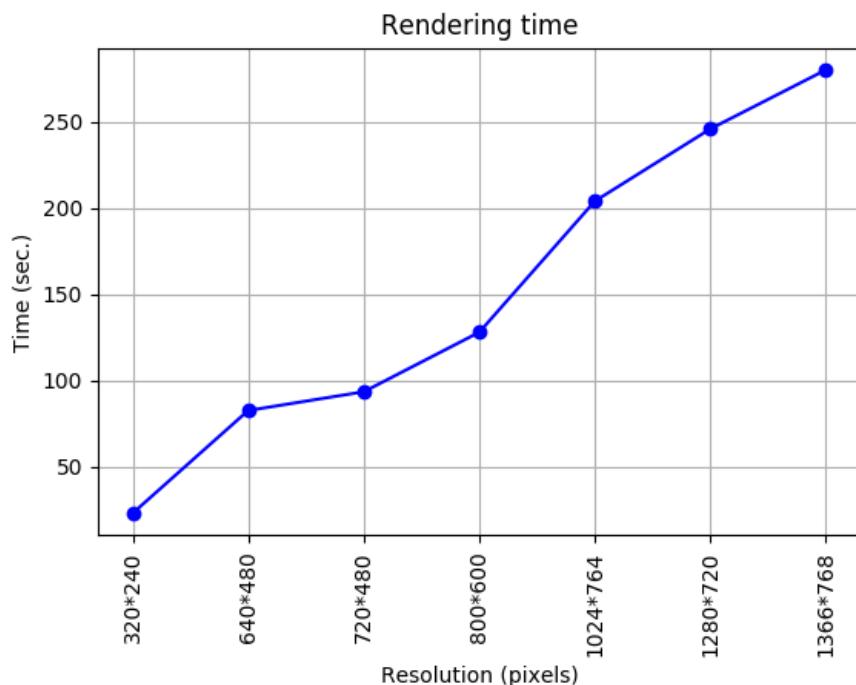
Első tesztelési esetként az szerettük volna lemérni, hogy mennyi idő szükséges egy statikus képre történő stílusátruházáshoz. A művelethez a videókártya párhuzamos számítási képességeit is igénybe vesszük. Maga a stílusátruhuzás a videókártyán fog történni. Ehhez a teszthez kiválasztottunk pár ismertebb felbontással rendelkező bemeneti képet és mindenik esetben lemértük az átruházáshoz szükséges időt. Az átruházás 300 iteráció alatt történt. Ugyanazt a stílusképet használtunk mindenik bemenet esetében. A stílus kép felbontása 600×855 pixel volt. A következő eredményeket kaptuk (4.1. ábra):

| Felbontás(pixel) | Idő(másodperc) |
|-------------------|----------------|
| 320×240 | 23.126104 |
| 640×480 | 82.640347 |
| 720×480 | 93.487573 |
| 800×600 | 127.973041 |
| 1024×764 | 203.892024 |
| 1280×720 | 245.769916 |
| 1366×768 | 279.945757 |

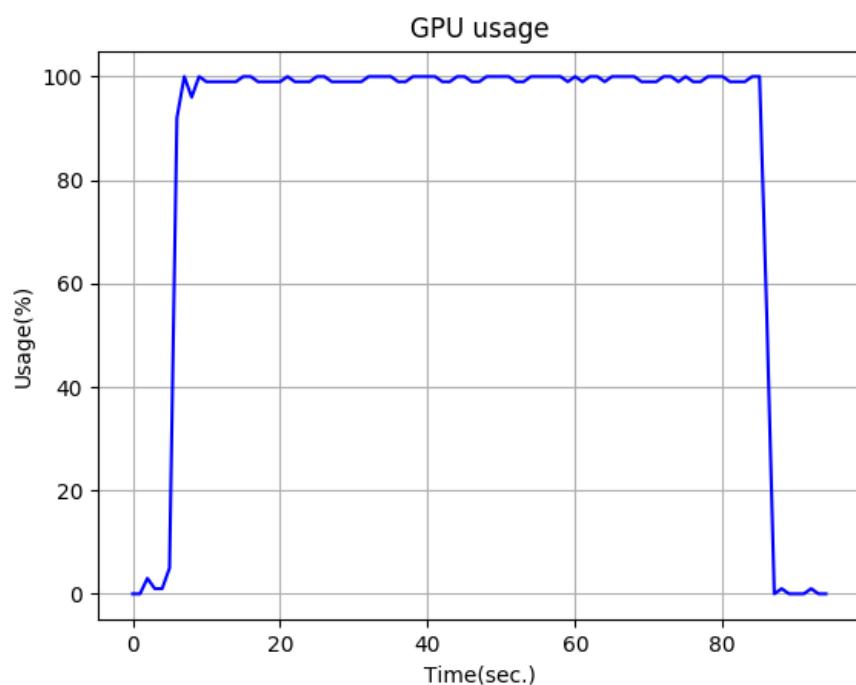
A 4.1. diagrammból lekövetkeztethetjük az, hogy az igényelt idő lineárisan nő a felbontás növekedésének megfelelően ugyanaz a stílus kép használatával. Az is egyértelművé válik, hogy a stílusátruházás high definition(HD) felbontású képek esetében elég időigényessé válik, ugyanis közel 5 percre van szükség egyetlen bemenet elvégzésére.

Az előbbi bemeneti képek esetében mérni szerettük volna a videókártya kihasználtságát is. Amint észlelhettük, még az alacsony felbontású képek esetében is elég időigényes a folyamat. Feltételeztük, hogy a Tensorflow könyvtár a kártyát maximálisan ki tudja használni. A tesztelő folyamathoz egy teljesen egyedülálló programot készítettünk. Ezt a stílusátruházás előtt el kell indítani. A program a gpustats[36] könyvtárat használja a videókártya működési paramétereinek a lekérdezésére. A programunk minden másodpercben lekéri egyszer a videókártya terheltségi százalékát és kimenti az egy fájlba amit később fel lehet használni.

A 4.2. diagramm igazolta előrejelzésünket. A videókártya használati százaléka az művelet elején felugrik szinte maximálisra és azon marad addig amíg be nem fejezte azt.



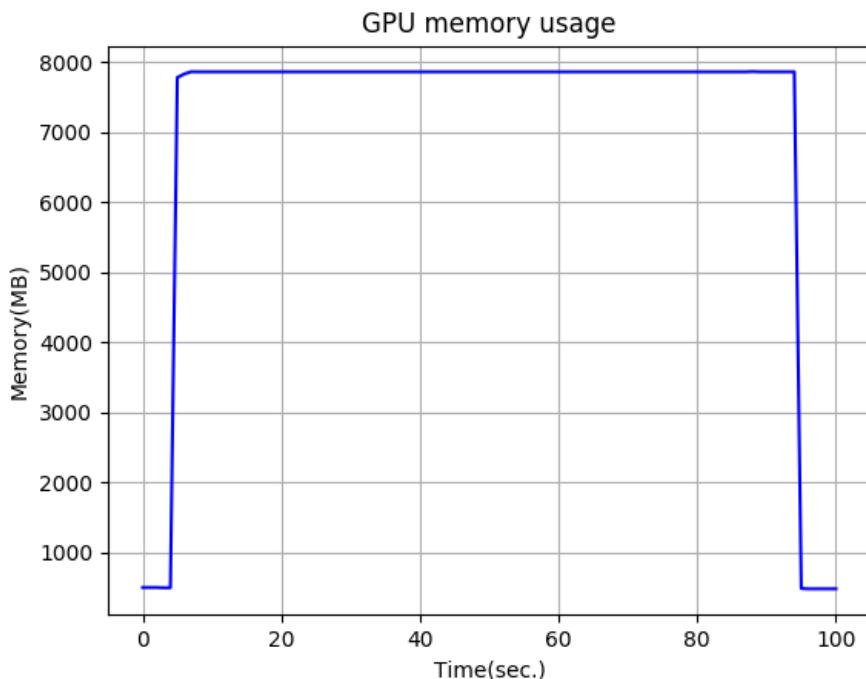
4.1. ábra. Stílusátruházási idő különböző felbontásokra



4.2. ábra. Videókártya terhelése átruházás közben

A 4.2. ábrán látható diagramm 640×480 -as felbontásra készült, de hasonló eredményeket kapunk bármely más felbontás esetében is.

Továbbá azt szerettük volna mérni, hogy mekkora a memória használat egy különböző felbontások esetében. Itt a logika azt sugallná, hogy a felbontásokkal arányosan lineárisan nőni fog. Ezzel ellenben a ábra azt igazolja, hogy a Tensorflow működése nem egyezik meg a előrejelzésünkkel. A mérés hasonlóan mint a kihasználtság esetében egy külön program méri ami egy külön processzen fut. A mérésre ugyancsak a gpustats könyvtárat használtuk.



4.3. ábra. Videókártya memóriájának a kihasználtsága

A 4.3. diagramm 640×480 felbontású kép esetében mért értékeket ábrázolja. A videókártyán maximálisan 7860MB volt használható a Tensorflow által. Amint látható, a Tensorflow a kezdetben lefoglalja az összes használható videómemóriát, majd ennek a felhasználásával végzi el a feladatát. A memória használat ugyanakkor 1366×768 felbontás esetében is. Mindez felvet egy másik kérdést, azt hogy mekkora az a maximális felbontás ami még elfér egy GTX 1080 típusú kártya memóriájában?

A kérdésre a válasz egyszerűen úgy adható meg, hogy vesszük az ismertebb felbontásokat és próbáljuk addig, amíg a Tensorflow nem dob egy memóriával kapcsolatos kivételt. Tovább próbálkoztunk a következő felbontásokkal: 1440×900 , 1600×900 és 1920×1080 . Ezennel elértek a fullHD minőséget. Statikus kép esetében egy GTX 1080-as kártyával ez meg elfogadható bemenet, viszont itt meg kell említeni, hogy az alkalmazás a HD minőséget célozta. Ebből kifolyólag a stílus képek mérete a HD felbontás környékén mozog.

Ez fullHD felbontás esetében bevezethet látható torzulásokat, ami abból adódik, hogy a stílus kép át fel lesz skálázva. Tovább haladva 2560×1440 felbontással próbálkoztunk, ami esetében jelentkezett a várt memória kivétel. Tehát a maximális felbontás amit lehetséges használni statikus képes esetében, az a 1920×1080 felbontás.

4.3. CPU-GPU futásidő összehasonlítása

Tisztába vagyunk azzal, hogy a deep learning tanítási módszer alkalmas az adatpárhuzamosításra. Érthető módon elvárt, hogy a processzoron történő futtatás jóval alul kellene maradjon a GPU-un történő futtatással szemben. Ez igazolják a következő mérések is. Ugyanúgy mint az előző mérések esetében 300 iteráció alatt történt a stílusátruházás. A bemeneti teszt képek is az előző teszthalmazból vannak kölcsönvéve, ebben az esetben három felbontásra futtattuk a tesztet. Ez elegendő kellene legyen arra, hogy lássuk a GPU használata által bevezetett gyorsulást. minden teszt esetben ugyanazt a stílusképet alkalmaztuk, aminek felbontása 600×855 pixel. Az mért eredmények a következők:

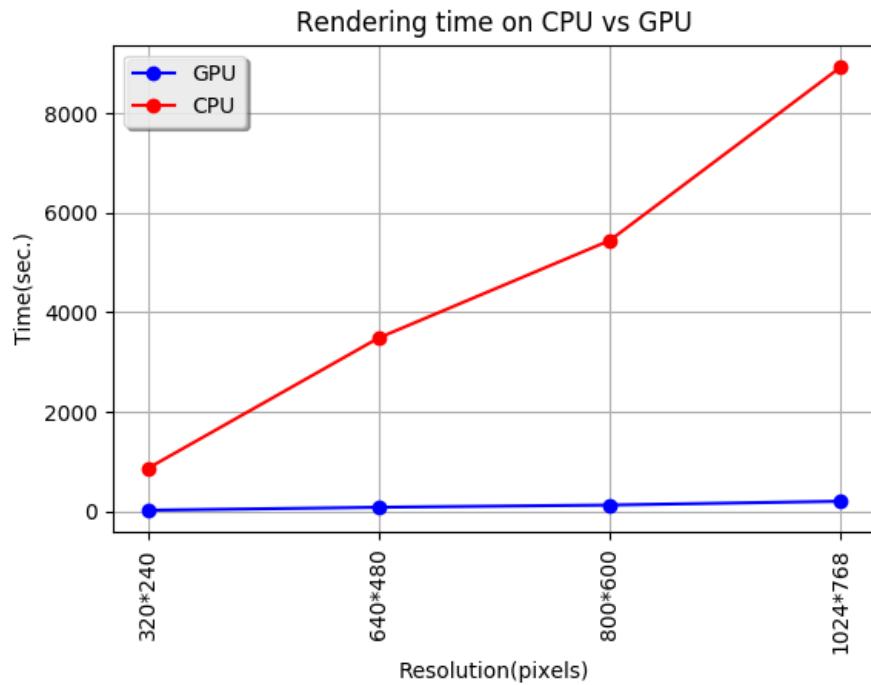
| Felbontás(pixel) | GPU Idő(másodperc) | CPU Idő(másodperc) | Gyorsulás |
|-------------------|--------------------|--------------------|-----------|
| 320×240 | 23.126104 | 862.980290 | 37.316285 |
| 640×480 | 82.640347 | 3487.931886 | 42.206162 |
| 800×600 | 127.973041 | 5446.849902 | 42.562479 |
| 1024×768 | 203.892024 | 8935.804122 | 43.826158 |

Előrejelzés szerint a párhuzamosított megoldás sokkal gyorsabb futásidőt igényel. Ahogyan a GPU esetében is láttuk, itt is a felbontás növekedésével lineárisan növekedik a futási idő is. Ami viszont megfigyelhető, hogy alacsonyabb felbontás esetében is a gyorsulás jelentős, 320×240 -es felbontás esetében 37-szeres gyorsulásról beszélünk. Ez, ugyan nem ennyire jelentősen, de észlelhetően növekedik a felbontás növekedésével egyaránt (4.5. ábra), 1024×768 felbontásnál, közel 44-es gyorsuláról beszélhetünk.

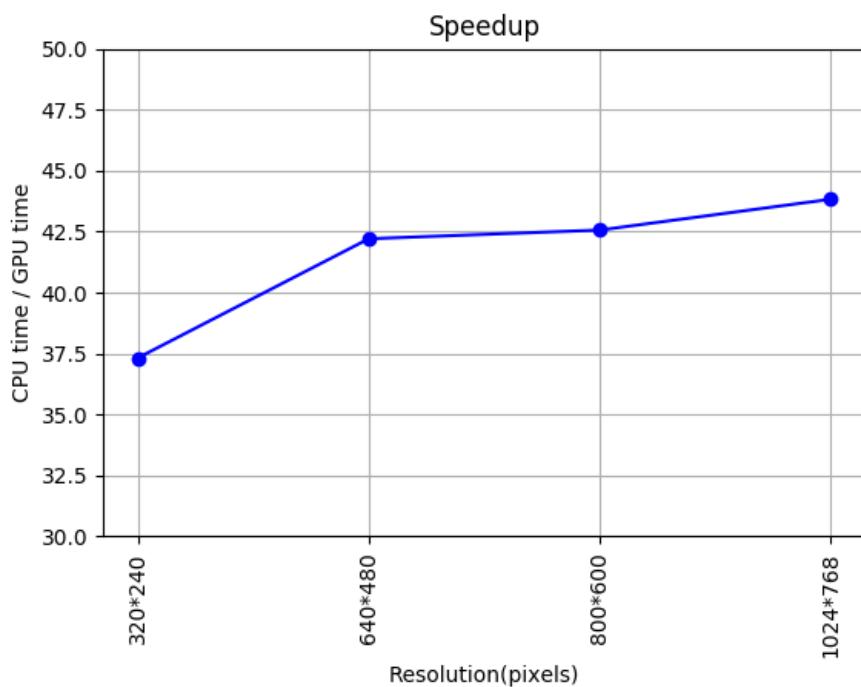
Következtetésképpen kijelenthető, hogy ha rendelkezünk megfelelő videókártyával, akkor mindenkorban azt ajánlott használni. Az előbbiekben bemutatott gyorsulás a CPU és GPU között egyetlen képkockára voltak mérve. Összehasonlításképp, egy 1024×768 felbontású képkocka GPU használatával 300 iterációs átruházás mellett körülbelül 3.4 perc alatt történik, míg CPU használatával körülbelül 149 perc alatt történik. Itt már órás különbségekről beszélünk, tehát mozgókép esetében semmiképp sem ajánlott CPU-t használni.

4.4. Videóra történő stílusátruházás időigénye

Mindenekelőtt le szerettük volna mérni az optical flow algoritmus futási idejét. Itt megjegyezzük, hogy a programban az deepmatching - deepflow algoritmust használtuk.



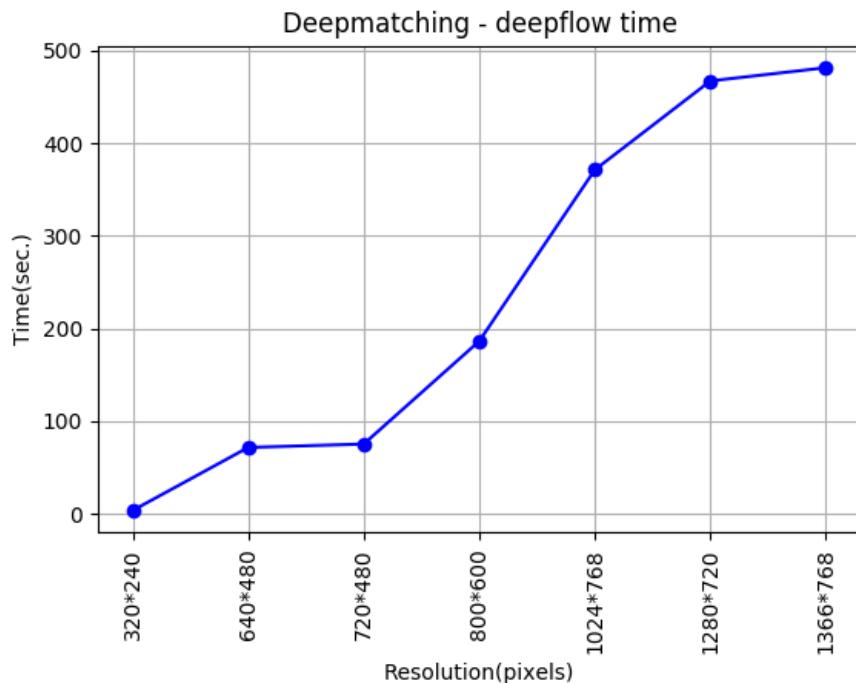
4.4. ábra. CPU futásideje összehasonlítva GPU futásidejével



4.5. ábra. Gyorsulás

A készítők által szolgáltatott nyílt források binárisok lettek integrálva az alkalmazásba. A binárisok a számítógép processzorát használják, hivatalos GPU-ra készített bináris a dolgozat írása idejében nem volt kiadva.

| Felbontás(pixel) | Idő(másodperc) |
|------------------|----------------|
| 320×240 | 3.700949 |
| 640×480 | 71.693549 |
| 720×480 | 75.388477 |
| 800×600 | 186.4160516 |
| 1024×764 | 371.140060 |
| 1280×720 | 466.996901 |
| 1366×768 | 481.25628 |

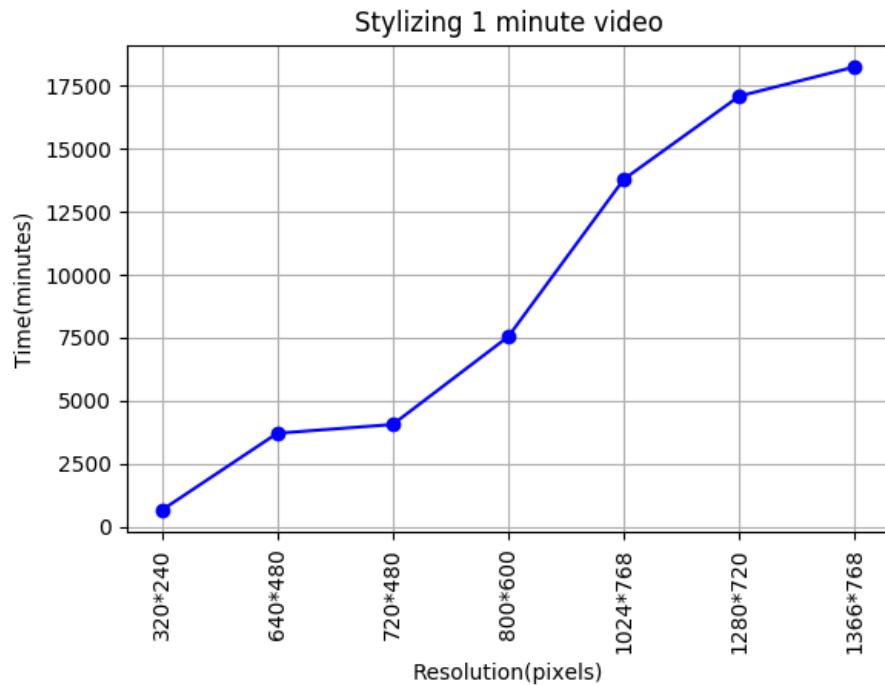


4.6. ábra. Optical flow futási ideje

Az optical flow minden két egymást követő képkocka között ki van számolva minden két irányba. Ez azt jelenti, hogy ha nekünk például 10 képkockánk van, akkor $9 + 9$ eredményt kapunk. Amint a 4.6. diagramm kimutatja, az optical flow eléggye költséges művelet, legtöbb esetben több időbe kerül, mint maga a stílusátvitel.

Továbbá egy 1 perces videóra történő stílusátruházás idejét szeretnénk lemerni. A videó 24 FPS (képkocka/másodperc) paraméterrel rendelkezik, ami azt jelenti, hogy 60 másodperces videó esetében 1440 képkockára kell átruházni a stílust.

A 4.7. ábrán látható diagramm tartalmazza az optical flow használatával történő átruházást, míg a 4.8. ábrán látható diagramm az optical flow nélküli átruházási időt



4.7. ábra. Egy perces videóra történő stílusátvitel optical flow használatával



4.8. ábra. Egy perces videóra történő stílusátvitel optical flow használata nélkül

szemlélteti. Mindkét esetben nagyon időigényes műveletről beszélünk, az első esetében HD felbontás esetében 12 napot futna a program. Ebből azt következtethetjük le, hogy a művelete természetesen kivitelezhető, de annak az időigényével számolni kell. 1 perc videóhoz 12 napos futási idő nem biztos, hogy megéri ez bárkinek a futtatást. Viszont ahogy már láttuk, hardver szempontjából a Tensorflow nagyon skálázható, ebből a kifolyóla a futási idő simán csökkenthető egy újabb videókártya hozzáadásával. Sajnos erre a dolgozat nem tartalmaz átfogóbb méréseket és tesztelést.

5. fejezet

Összefoglaló

összefoglaló

Ábrák jegyzéke

| | | |
|-------|--|----|
| 3.1. | A rendszer működése felülnézetből | 20 |
| 3.2. | Tensorflow számítási gráf[35] | 21 |
| 3.3. | Egyetlen procesz összehasonlítása több proceszes működéssel[35] | 23 |
| 3.4. | Adatpárhuzamos szinkron és aszinkron csomópont kiértékelés[35] | 23 |
| 3.5. | Feladatpárhuzamos tanítás[35] | 24 |
| 3.6. | VGG-19 (Model E) háló topológiája[29] | 26 |
| 3.7. | A stílus súlyzó módosításának eredményei | 28 |
| 3.8. | Tanítási függvény optimalizálása különböző tanítási ráták esetében | 31 |
| 3.9. | A rendszer viselkedési modellje | 34 |
| 3.10. | A rendszert alkotó komponensek | 36 |
| 3.11. | Átruházás szekvencia diagramma | 37 |
| 3.12. | A rendszer osztálydiagramma | 39 |
| 3.13. | MainWindow | 40 |
| 3.14. | A rendszer felhasználói felülete | 41 |
| 3.15. | Stílusnépek | 42 |
| 3.16. | A beállítások komponens osztálydiagramma | 43 |
| 3.17. | Beállítások dialogus ablak | 44 |
| 3.18. | A progress sáv komponens osztálydiagramma | 45 |
| 3.19. | Progress sáv statikus kép esetében | 46 |
| 3.20. | Progress sáv videó esetében (deepflow kikapcsolva) | 46 |
| 3.21. | Progress sáv videó esetében (deepflow bekapcsolva) | 46 |
| 3.22. | A stílusátruházó (Artistic video creator) komponens osztálydiagramma . . | 47 |
| 4.1. | Stílusátruházási idő különböző felbontásokra | 51 |
| 4.2. | Videókártya terhelése átruházás közben | 51 |
| 4.3. | Videókártya memóriájának a kihasználtsága | 52 |
| 4.4. | CPU futásideje összehasonlítva GPU futásidejével | 54 |
| 4.5. | Gyorsulás | 54 |
| 4.6. | Optical flow futási ideje | 55 |
| 4.7. | Egy perces videóra történő stílusátvitel optical flow használatával | 56 |
| 4.8. | Egy perces videóra történő stílusátvitel optical flow használata nélkül . . | 56 |

Irodalomjegyzék

- [1] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks (2012)
- [2] Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks (2013)
- [3] <https://en.wikipedia.org/wiki/CUDA> (2017.04.24)
- [4] <http://caffe.berkeleyvision.org> (2017.04.24)
- [5] <https://keras.io/> (2017.04.24)
- [6] <http://deeplearning.net/software/theano/> (2017.04.24)
- [7] <https://www.tensorflow.org/> (2017.04.24)
- [8] <http://torch.ch/> (2017.04.24)
- [9] <https://computerstories.net/microsoft-computer-outperforms-human-image-recognition> (2017.04.29)
- [10] Kevin Alfianto, Mei-Chen Yeh, Kai-Lung Hua - Artist-based Classification via Deep Learning with Multi-scale Weighted Pooling (2016)
- [11] Rosenblatt F. - The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain (1958)
- [12] Werbos, P.J. - Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences (1975)
- [13] LeCun, Yann, Léon Bottou, Yoshua Bengio, Patrick Haffner - Gradient-based learning applied to document recognition (1998)
- [14] Dave Steinkraus, Patrice Simard, Ian Buck - Using GPUs for Machine Learning Algorithms (2005)
- [15] Gatys, L. A., Ecker, A. S., Bethge - A neural algorithm of artistic style (2015)

- [16] Yaroslav Nikulin, Roman Novak - Exploring the Neural Algorithm of Artistic Style (2016)
- [17] Justin Johnson, Alexandre Alahi, Li Fei-Fei - Perceptual Losses for Real-Time Style Transfer and Super-Resolution
- [18] Ulyanov, D., Lebedev, V., Vedaldi, A., and Lempitsky - Texture networks: Feed-forward synthesis of textures and stylized images
- [19] Ulyanov, D., Lebedev, V., Vedaldi, A., and Lempitsky - Instance Normalization: The Missing Ingredient for Fast Stylization
- [20] [https://en.wikipedia.org/wiki/Prisma_\(app\)](https://en.wikipedia.org/wiki/Prisma_(app)) (2017.04.29)
- [21] Manuel Ruder, Alexey Dosovitskiy, Thomas Brox - Artistic style transfer for videos (2016)
- [22] -
- [23] <https://www.python.org/> (2017.04.30)
- [24] <http://www.numpy.org/> (2017.04.30)
- [25] <https://www.riverbankcomputing.com/software/pyqt/intro> (2017.04.30)
- [26] <http://opencv.org/> (2017.04.30)
- [27] Gatys, L. A., Ecker, A. S., Bethge, M. Texture synthesis and the controlled generation of natural stimuli using convolutional neural networks (2015)
- [28] <https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html> (2017.05.01)
- [29] Simonyan, K., Zisserman, A. - Very Deep Convolutional Networks for Large-Scale ImageRecognition (2015)
- [30] <http://mathworld.wolfram.com/GramMatrix.html> (2017.05.02)
- [31] https://en.wikipedia.org/wiki/Total_variation_denoising (2017.05.03)
- [32] Kingma D. P., Lei Ba, J. - ADAM: A method for stochastic optimization (2015)
- [33] <http://caffe.berkeleyvision.org/tutorial/solver.html> (2017.05.14)
- [34] <https://www.mathworks.com/discovery/optical-flow.html> (2017.05.18)

- [35] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng - TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems (2015)
- [36] gpustats library: <https://github.com/dukestats/gpustats> (2017.06.04)