UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA

DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Aplicație pentru transferul de stil

Absolvent

Stegarescu Ana-Maria

Coordonator științific

Boriga Radu

București, iunie 2021

Rezumat/Summary:

Deși lucrarea mea însumează mai multe subiecte: Transferul de stil pe imagini, Similaritatea între imagini calculată cu ajutorul unui model AI, și User Interface Design, consider că principalul subiect este Transferul de stil între imagini.

Relevanța temei: Neural Style Transfer (NST), domeniu destul de nou, nu pare să existe în scopul îmbunătățirii confortului vieții umane, așa cum o fac majoritatea produselor/algoritmilor din domeniul digital. Totuși, e un domeniu de studiu din Computer Vision care ar putea aduce roade odată cu evoluția Inteligenței Artificiale și pentru condiția umană. La moment, NST face o legatură între artă și IA, deci scopul este pur divertisment, dar rezultatele sunt de-a dreptul promițătoare. Un model NST procesează o imagine conținut, și reține formele, culorile, textura și alte detalii din imagine. Apoi aplică stilul unei imagini-stil asupra informației procesate din imaginea conținut. Voi explica procesul detaliat în capitolele ce urmează. Modelul NST utilizat în aceasta lucrare este un model licențiat de Apache-2.0[12], pe care am făcut research.

Un alt AI utilizat este cel pentru comparația între imagini. Pe acesta l-am construit singură. Explicațiile și codul, de asemenea, urmează.

Interfața grafică, construită de mine cu ajutorul a PyQt5, plasează utilizatorul într-un experiment NST, unde el își poate alege imaginile conținut și stil, care se aleg random uniform de fiecare dată când userul apasă pe buton pentru a le schimba. Utilizatorul poate încărca o imagine proprie ca încercare de a simula rezultatul unui NST, sau poate desena o imagine. În acest caz, o fereastră nouă va aparea cu aplicația pentru desen. Taskul userului: să deseneze o imagine cu conținutul ales și stilul ales.

Urmează ca modelul NST să fie pus în funcțiune, și fiindcă procesarea informațiilor/detaliilor din imagini durează mult (în jur de 100 de secunde conform experimentelor mele), am decis să creez un nou fir de execuție. Pe un singur fir, aplicația va aștepta până userul termină de desenat pentru a putea începe un task nou (cel de transfer se stil). Pe firul meu nou de execuție, procesul de transfer de stil începe în paralel cu utilizarea aplicației de desen de către user. Sigur, transferul de stil poate fi început înainte ca userul să decidă că va desena, doar în cazul în care e curios să vadă rezultatul modelului

NST. Ultima parte: Comparația între imaginea userului și cea obținută de NST, care oferă o notă de similaritate între 0 și 10.

Obiectivele și rezultatele: O aplicație interactivă, bazată pe subiectele descrise mai sus. (Figura 1 și 2)

Summary:

Even though my project assembles more than one subject: Neural style transfer, Image similarity with Deep Learning, and User Interface Design, I consider that the main subject is Neural Style Transfer (NST).

Subject relevance: At this moment in time, NST does not seem to play a role in making life more comfortable, but it was created with the purpose to inspire. It is a small, quite new domain. A bridge between technology and art. A NST model processes a content image and a style image, applying the style to the content for the resulting image.

The NST model used by me for this project is a model licensed by Apache-2.0[12], which I researched on.

Another AI which I used was one made by me. It is a model that does a comparison between two images, and gives the similarity, a number between 0 and 10.

I made the graphical user interface with PyQt5. I called the resulting app an "Experiment with NST" that lets the user simulate what an NST does. The user picks a content image and a style image randomly (the images change on button click), and then they can either upload their own image, or choose to draw a picture with the said content and style. Upon clicking the "Start to draw" button, a new window will appear with a drawing app I also made with PyQt5.

There is also a button for the image style transfer, that starts the process. I left the button enabled from the start in case the user is curious about the result of the NST model. However, the process may take around 100 seconds (from my experiments), so while the user is busy drawing the picture, the style transfer process will start automatically on a 2nd thread.

The last step is the comparison between the picture the user drew or uploaded, and the NST result.

Results and objectives: An interactive app based on everything described above. (Figure 1 and 2)

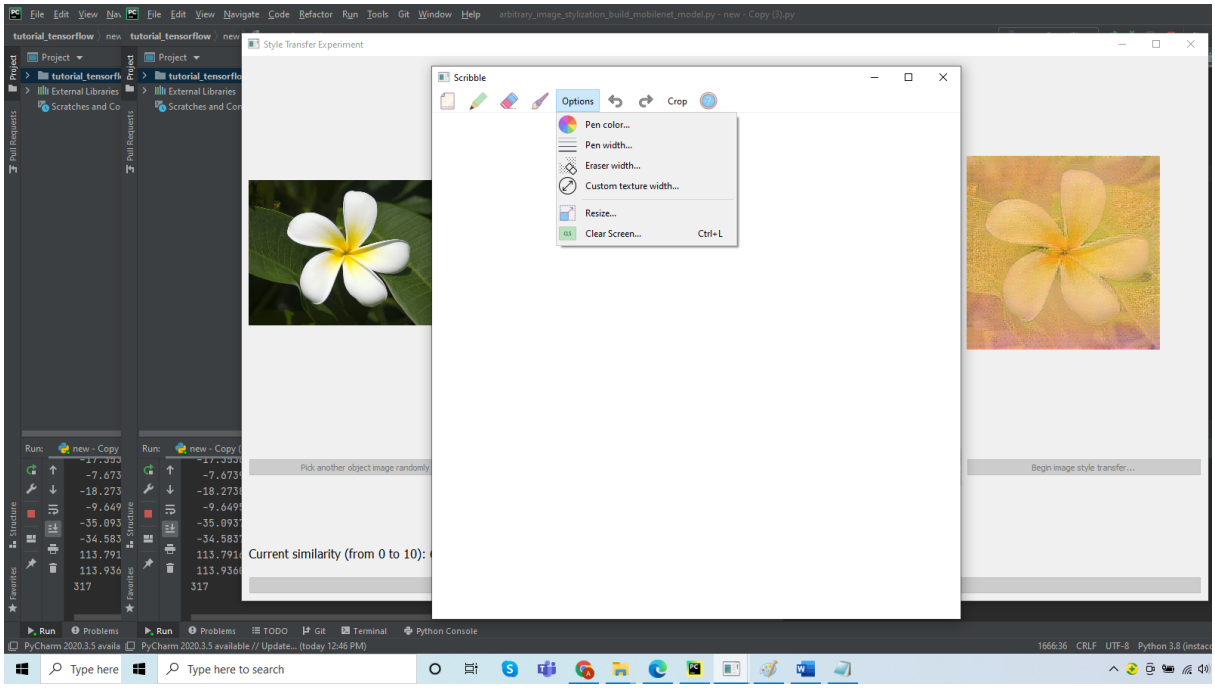

Figura 1: Aplicația pentru desen

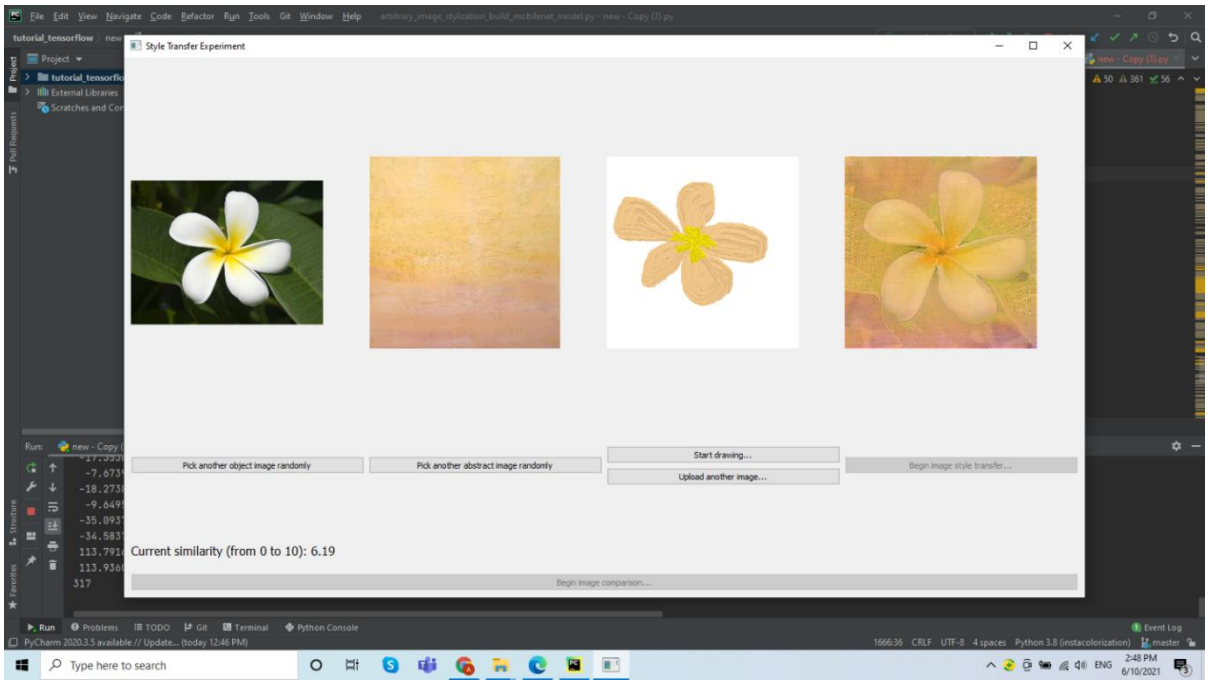

Figura 2: Aplicația interactivă bazată pe NST

Cuprins:

I. Introducere

Istoric NST: După cum am spus în rezumat, NST e un domeniu destul de nou. Prima lucrare a fost publicată în 2015, pe ArXiv.org, "A Neural Algorithm of Artistic Style" de către Leon Gatys [8], conform Wikipedia[25]. A fost acceptat de către Computer Vision and Pattern Recognition (CVPR) în 2016, o conferință anuală din domeniul Computer Vision.

Acest model a folosit o arhitectură tip VGG-19, preantrenată pe task-ul image recognition pe datasetul ImageNet[20]. Aceasta este o arhitectură convoluțională, care apoi a fost transferată pe task-ul de transfer de stil. (Figura 1.1)

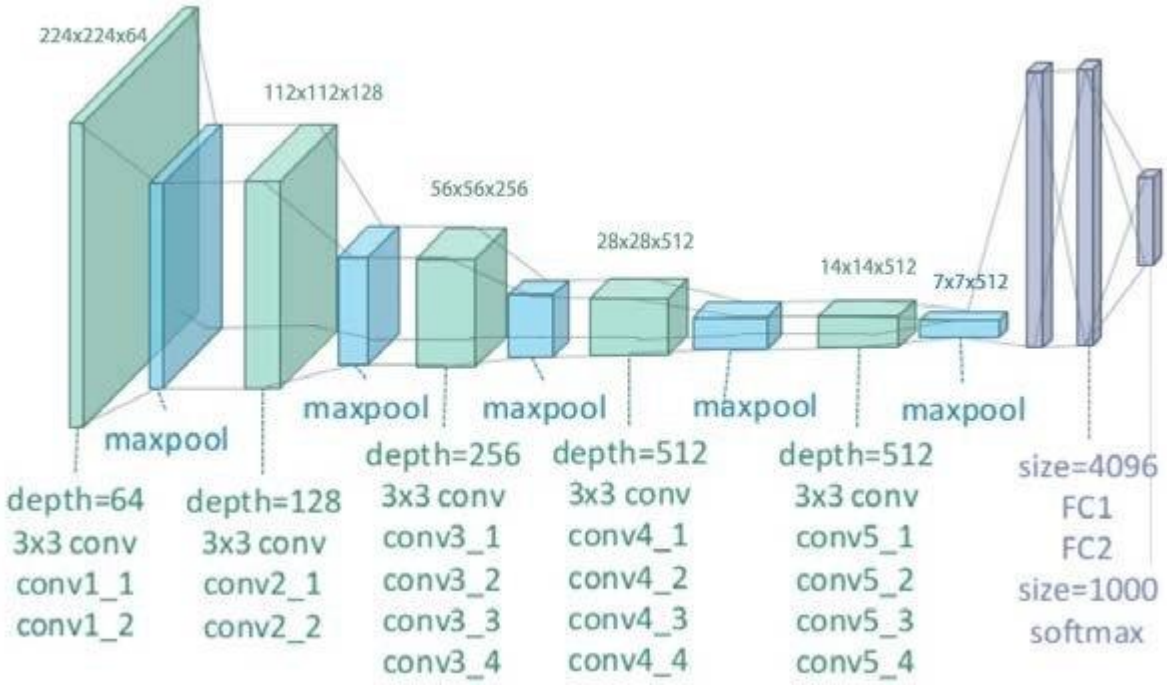

Figura 1.1: Arhitectura convoluțională VGG-19

Voi explica în următoarele capitole arhitecturile convoluționale și cum are loc transferul de task.

NST în prezent: Google AI a creat un nout tip de model care permite unei singuri arhitecturi convoluționale să învețe mai multe stiluri pe care le transferă pe o singură imagine conținut [6].

II. Preliminarii

II.1 Arhitecturile convoluționale - CNN (Convolutional Neural Networks)

O arhitectură CNN este utilizată de obicei pentru procesarea imaginilor, în diverse taskuri.

Ea primește o matrice conținut (feature matrix), de exemplu de o formă [64, 224, 224],

(64 de imagini, cu 1 canal pentru culoare, imagini alb-negru, de dimensiuni 224 x 224 pixeli). Fie o arhitectură de forma celei din figura 3.1. În următorul chenar verde, fie pasul 2, dimensiunea devine (128, 112, 112). O arhitectură CNN micșorează dimensiunea imaginilor cu fiecare pas, dar (în acest caz) mărește dimensiunea de feature. Cu fiecare pas, se procesează ierarhic diverse informații din imagini. Primele layere (de dimensiuni mai mari pentru imagini), procesează cele mai mici detalii, cum ar fi puncte. Următoarele layere – linii, următoarele – forme etc. Într-un sfârșit, informația imaginilor inițială este restrânsă în cel din urmă ”chenar” de 7x7 pixeli, dar 512 feature-uri. Este o matrice 3-dimensională, care, pentru această arhitectură din figură, este redimensionată într-un vector 1D de mărime 4096, care conține toată informația.

Sigur, pentru fiecare tip de task arhitectura diferă. Cum ar fi, dacă vrem să clasificăm imagini (fie câini și pisici – 2 clase), ar trebui să obținem în final o matrice 2-dimensională de tip (nr imagini, 2). Pentru fiecare imagine, vom avea 2 probabilități – prima ne indică probabilitatea de apartenență în prima clasă, a doua – cea de-a doua clasă.

Fiecare arhitectură este antrenată într-un număr fixat de epoci, până la obținerea acurateții dorite. În cadrul antrenamentului, arhitectura se adaptează cerinței/task-ului, și învață din rezultatele sale anterioare greșite. Astfel, un CNN învață ce fel de informație ar trebui să rețină pentru cazul dat (ce fel de detalii din imagine îl interesează).

Despre filtre:

La trecerea informației printre layerele CNN, se aplică un filtru/kernel (Figura 2.1.1)

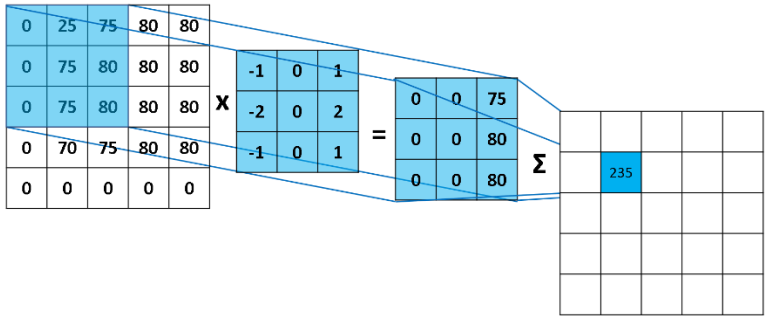

Figura 2.1.1: Aplicarea unui filtru pe o matrice

Un kernel e o matrice de mărime obligator mai mică decât cea de feature. Într-un mod glisant, ea se aplică pe fiecare porțiune din matricea originală, producându-se un produs scalar între filtru și porțiune. Acesta este motivul pentru care un CNN reține detaliile ierarhic din imagini: din porțiunea de 3x3 de mai sus se obține un singur număr -> porțiunea din imagine ar putea conține o linie, și acum toată informația va fi redusă la un singur număr. La trecerea prin următorul layer, numerele ce conțin informații despre linii vor fi din nou reduse la un singur număr -> o figură sau formă. Filtrul glisează cu 1 la dreapta, apoi cu 1 jos când termină rândul, în cazul în care dimensiunea pentru glisare este 1. Aceasta este setată la construcția arhitecturii, și poartă denumirea de ”stride”. Matricea rezultată după aplicarea unui filtru poartă numele de ”feature map”.

Despre pooling:

Layerele de pooling se află între cele convoluționale și nu sunt obligatorii într-un CNN.

Ele de asemenea au un filtru de dimensiune constantă și stride constant. Rolul lor este de a mai reduce din mărimea unui feature map, dacă este necesar pentru adaptarea unui CNN mai ușor la un task. Un **average pooling** calculează media porțiunii curente din feature map, porțiune de mărimea filtrului. Media e noul element din matricea rezultat, micșorată. Un **max pooling** (Figura 2.1.2) ia doar numărul maxim din feature map și îl reține în matricea rezultat. După, glisează pe feature map la fel ca pentru kernelul obișnuit convoluțional.

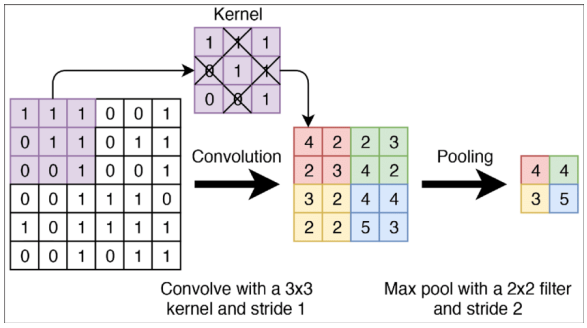

Figura 2.1.2: Aplicarea unei convoluții urmate de max pool

Uneori se dorește micșorarea dimensiunii cu care lucrăm, cu condiția că nu pierdem din informația esențială. Este mai ușor să reținem în memorie mai puțină informație. Un exemplu elegant dintr-un tutorial pe YouTube [3] (Figura 2.1.3):

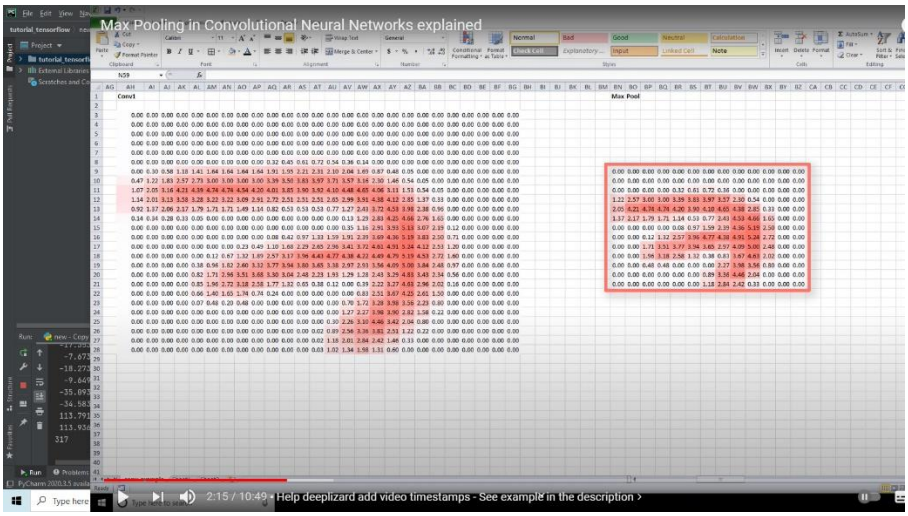

Figura 2.1.3: Max pool

În figura 2.1.3, numerele mai mari din prima matrice indică o culoare mai aprinsă, și cu cât sunt mai aproape de 0, cu atât sunt mai aproape de alb. Matricea 2 este cea rezultată după max pooling, filtru de mărime 2x2. Pentru fiecare porțiune de 2x2 din matricea originală, s-a extras numărul maxim. Matricea rezultată păstrează informația esențială, dar fiind mai mică, ocupă mai puțin loc și e mai ușor de operat cu ea.

Alte tipuri de arhitecturi CNN:

Un exemplu de Fully Convolutional Network (FCN), Figura 2.1.4:

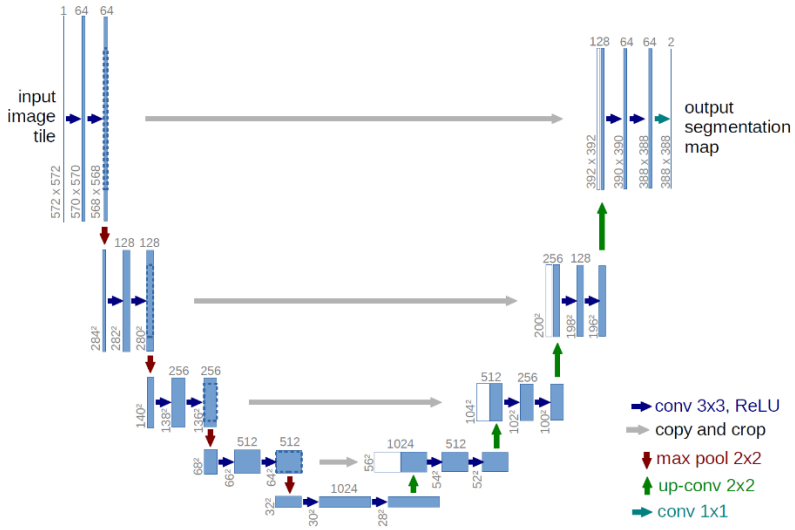

Figura 2.1.4: FCN pentru taskul de segmentation

Această arhitectură se numește FCN, fiindcă are doar layere convoluționale (un CNN care nu e FCN are cel puțin și un layer liniar, cum ar fi pentru un output de dimensiuni 1D sau 2D).

Observați că acest model nu doar micșorează imaginea, dar o și mărește la loc.

Motivul: Taskul este segmentarea imaginilor. Pe parcursul procesării, acest model studiază toate elementele și detaliile imaginii. Restrânge imaginea la cele mai mici mărimi, după care o reconstruiește. Pentru a face o reconstrucție corectă, și pentru că informația se pierde în rețelele adânci cum este aceasta (rețele cu multe layere), informația de la un layer inițial se transmite direct către cel final (săgețile gri), dar și indirect trecând prin layerele ce le despart.

Ce este segmentarea?

Segmentarea este identificarea obiectelor separate din imagine, unde încep și unde se termină (conturul), și ”despărțirea” lor de restul imaginii. Exemplu, figura 2.1.5:

Figura 2.1.5: Image segmentation

Un CNN este perfect pentru determinarea figurilor unei imagini.

Observație: Arhitectura din figura 2.1.4 se mai numește ”U-Shape”, datorită formei sale, și fiindcă transmite informație direct de la layerele low-level la cele high-level.

II.2 Funcții de activare

(Vor fi relevante pentru motivul pentru care am ales ReLU în unele cazuri, și în altele nu pentru modelul meu)

Funcțiile de activare sunt funcții prin care trece informația la ieșirea din fiecare layer. Cea mai simplă și utilizată funcție este ReLU (Rectified Liniar Unit), care este de fapt max(0, x), pentru x matricea rezultată dintr-un layer. Există o multitudine de tipuri, și se aleg în funcție de task, dar și de experiența programatorului.

Alegerea unei funcții de activare conform taskului:

Dacă facem o clasificare, independent de câte clase sunt, am vrea ca la output să avem numere pozitive pentru probabilitate. Dacă aplicăm ReLU, scăpam de numere negative. O idee mai

bună ar fi să utilizăm funcția Softmax, care scalează valorile între 0 și 1. Cum ar fi, pentru vectorul [10, 50, 100], numărul maxim ar fi 1, și restul sunt calculate ca raport după maxim -> [0.1, 0.5, 1]. Această funcție este utilizată atunci când vrem să obținem probabilități.

Atunci putem folosi Softmax doar pentru output, nu și între layere. (Fals)

Motivul pentru care sunt necesare funcțiile de activare între layere:

Layerele într-o arhitectură nu sunt decât niște produse pe matrici aplicate unui input. Cu cât avem mai multe layere, cu atât avem un model mai complex, care se adaptează mai bine taskului (cu condiția că avem grijă la modul cum e procesată informația, sau la comportamentul gradientului la învățare).

Presupunem că nu avem funcții de activare între layere. Fie o matrice de input I. La intrarea în primul layer, se înmulțește cu o matrice A, al doilea layer – se înmulțește cu B.

Faptul că avem 2 layere deja face modelul mai complex decât dacă avem un singur layer. Dar avem cu adevărat 2 layere? Nu.

I * A * B = I * (A*B) = I * C, dacă A*B = C oarecare.

E același efect de parcă am avea un singur layer. Fiindcă înmulțirea e o operație liniară și necesită funcții de activare pentru ca relația de mai sus să nu aibă loc.

Există șansa ca funcțiile de activare să influențeze negativ valorile date ca input?

Nu! Să luăm ca exemplu ReLU. Valorile negative sunt înlocuite cu 0, deci dacă avem o matrice cu valori atât pozitive, cât și negative, se pierde acea parte din informație.

Totuși, modelul nu e afectat. Pe masură ce se antrenează, se adaptează la orice funcție de activare. Totuși, unele sunt mai bune decât altele în funcție de task. Fie cele 2 matrici de mai devreme, A și B, care aparțin primelor 2 layere. La prima epocă, modelul dă rezultate greșite (de obicei știe să ”ghicească” random). Calculează un loss pentru rezultatul lui, o ”diferență” dintre ce a obținut și ce trebuia să obțină, și schimbă puțin valorile din matricile A și B astfel încât la trecerea inputului I prin layere (la înmulțirile cu noul A și B), valorile să fie un pic mai aproape de adevăr. Inițial face schimbări mai mari în A și B, apoi tot mai mici și mai mici cu fiecare pas. Acest proces se numește ”Gradient Descent”, sau Coborârea pe Gradient. Coborârea pe gradient este motivul pentru care modelul își poate adapta valorile din matrici, indiferent de ce schimbări face ReLU.

II.3 Transferul de task în Inteligența Artificială

Transferul de task are loc atunci când antrenăam un model pentru un anumit task, acesta în timpul antrenamentului ”se adaptează” cerinței (face schimbări corespunzătoare la weights și biases, weights fiind matricile A și B utilizate de mine ca exemplu mai sus), și aceleași matrici de weights și biases, sau o parte din ele, sunt utilizate pentru un alt task.

Pentru modelul meu ce face comparație între imagini, am recurs și eu la transferul de task (va urma).

În istoricul NST vorbeam despre arhitectura preantrenată pe image recognition, care a fost apoi utilizată pentru transferul de stil pe imagini. Motivul este faptul că s-au utilizat 2 rețele. Prima procesa imaginea ce cuprine conținutul. Un model antrenat pentru clasificarea obiectelor/recunoașterea de obiecte are layere (cu weights și biases) antrenate pe ”învățarea” conținutului imaginii, ”înțelegerea” figurilor și formelor din imagine. Câteva layere au fost preluate de acolo, pentru a obține o matrice ce conține informația esențială a conținutului / formei obiectului pe care se face transferul de stil. Modul în care se face transferul de stil urmează să fie discutat în subcapitolul ce urmează.

II.4 Transferul de stil pe imagini: informații teoretice

În subcapitolul 3, odată cu transferul de task pentru NST, am explicat și cum se preiau figurile și formele din imaginea conținut. Pentru a explica transferul de stil, voi avea nevoie, din nou, de schema arhitecturii VGG-19, figura 2.4.1:

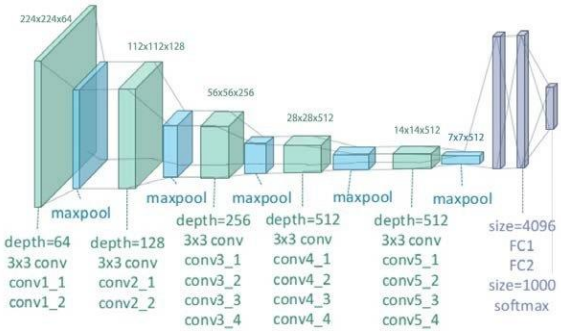

Figura 2.4.1: Arhitectura VGG-19

Observați că, sub fiecare ”chenar” ce reprezintă matricile de feature maps, se află ”subgrupuri”, denumirile layerelor convoluționale. Acestea se numesc blocuri. Fiecare bloc procesează un anumit tip de informație din imagine (ierarhic, cum am mai spus). Primul layer convoluțional procesează culoarea într-o imagine, sau mai bine zis textura. Aici se află ”stilul” imaginii pe care vrem să îl extragem, iar conținutul e procesat de layerele mai adânci din bloc.

Trecem imaginea-stil prin aceeași arhitectură, la fel cum am făcut pentru imaginea conținut. Fie imaginea-stil ”s”, și imaginea conținut ”c”. Efectuăm un produs scalar între ”stil” și ”conținut” astfel <c, conv1_1(s)>.....<conv5_1(s), c>, unde informația privind conținutul, ”c”,

a fost extrasă anterior ierarhic. Aceste produse scalare se mai numesc corelații, sau ”matrici gram”, sau „Gramian Matrix”, care ne dau imaginea rezultat.

Calculul loss-ului în decursul antrenamentului:

Notăm cu ”p” imaginea conținut, ”a” – imaginea stil, ”x” – imaginea generată la fiecare epocă a antrenamentului.

Atunci A_i va fi feature map-ul obținut din imaginea stil pentru layerul i, P_i – feature map-ul pentru imaginea conținut, și F_i va fi feature map-ul pentru conținutul imaginii generate, iar G_i pentru stilul imaginii generate, x.

Loss(x) = |F – P| + k |G – A|, unde k este o constantă care controlează nivelul de stilizare, iar |.| este norma L2.

(informații conform Wikipedia[25] și un tutorial pe YouTube[2])

III. Contribuția / proiectul în sine

III.1 Aplicația pentru desen

Am luat un cod de bază scris în Qt pentru C++ de pe site-ul oficial pentru Qt, oferit ca exemplu pentru utilizarea clasei QPainter[15]. Aplicația de pe website oferea o clasă pentru imagine, alături de pixul digital și radiera, și o clasă pentru fereastra principală, care permitea și salvarea imaginii.

Am tradus în sintaxa Python, pentru PyQt5, și am adăugat multe alte funționalități. Am permis utilizatorului să schimbe diametrul radierei.

```
eraserWidthIcon = QtGui.QIcon(":eraserwidth")
self.eraserWidthAct = QtWidgets.QAction(eraserWidthIcon, self.tr("Eraser
width..."), self)
self.eraserWidthAct.triggered.connect(self.eraserWidth)
```

```
def eraserWidth(self):
    slider = Slider("Select eraser width", 1, 15,
self.scribbleArea.eraserWidth)
    self.scribbleArea.eraserWidth = slider.val
    self.hiddenArea.eraserWidth = slider.val
```

```
class Slider(QtWidgets.QDialog):
    def __init__(self, title, min_val, max_val, current_val):
        super().__init__()
        self.setWindowTitle(self.tr(title))
        self.setWindowModality(QtCore.Qt.WindowModal)
        self.setModal(True)
        self.slider = QtWidgets.QSlider()
        self.slider.setRange(min_val, max_val)
        self.slider.setPageStep(1)
        self.slider.setOrientation(QtCore.Qt.Horizontal)
        self.layout = QtWidgets.QVBoxLayout()
        self.layout.addWidget(self.slider)
        self.button = QtWidgets.QPushButton('Ok', self)
        self.button.resize(150, 50)
        self.button.clicked.connect(self.getVal)
        self.layout.addWidget(self.button)
        self.setLayout(self.layout)
        self.resize(400, 100)
        self.val = None                                          #changes
on button click
        self.slider.setValue(current_val)
        self.exec_()


    def getVal(self):
        self.val = self.slider.value()
        self.close()
```

Am schimbat modul de resize a imaginii (aplicația făcea resize la redimensionarea ferestrei, acum utilizatorul face resize printr-o fereastră de dialog unde scrie lungimea și înalțimea dorită).

```
resizeIcon = QtGui.QIcon(":resize")
self.resizeAct = QtWidgets.QAction(resizeIcon, self.tr("Resize..."), self)
self.resizeAct.triggered.connect(self.resizeImage)
```

```
def resizeImage(self):
    resizeObj = resizeArea(self.scribbleArea.width, self.scribbleArea.height)
    newSize = QtCore.QSize(resizeObj.widthVal, resizeObj.heightVal)
    if self.scribbleArea.image.size() == newSize:
        return
    newImageRGB = QtGui.QImage(newSize, self.scribbleArea.rgb)
    newImageRGBA = QtGui.QImage(newSize, self.scribbleArea.rgba)
    newImageRGBA.fill(QtCore.Qt.transparent)
    newImageRGB.fill(QtGui.QColor(255, 255, 255))

    painterRGB = QtGui.QPainter(newImageRGB)
    painterRGB.drawImage(QtCore.QPoint(0, 0), self.scribbleArea.image)
    self.scribbleArea.image = newImageRGB

    painterRGBA = QtGui.QPainter(newImageRGBA)
    painterRGBA.drawImage(QtCore.QPoint(0, 0), self.hiddenArea.image)
    self.hiddenArea.image = newImageRGBA

    self.scribbleArea.undo_images.append(self.scribbleArea.image.copy())
    self.hiddenArea.undo_images.append(self.hiddenArea.image.copy())
```

```
class resizeArea(QtWidgets.QDialog):
    def __init__(self, width, height):
        super().__init__()
        self.layout1 = QtWidgets.QHBoxLayout()
        self.layout2 = QtWidgets.QHBoxLayout()
        self.layout3 = QtWidgets.QVBoxLayout()
        self.setWindowTitle(self.tr("Change the image width/height"))
        self.widthVal = None
        self.heightVal = None
        self.labelWidth = QtWidgets.QLabel("Set the width: ")
        self.labelHeight = QtWidgets.QLabel("Set the height: ")
        self.widthBox = QtWidgets.QSpinBox()
        self.heightBox = QtWidgets.QSpinBox()
        self.widthBox.setMinimum(1)
        self.heightBox.setMinimum(1)
        self.widthBox.setRange(1, 5000)
        self.heightBox.setRange(1, 5000)
        self.widthBox.setSingleStep(50)
        self.heightBox.setSingleStep(50)
        self.widthBox.setValue(width)
        self.heightBox.setValue(height)
        self.layout1.addWidget(self.labelWidth)
        self.layout1.addWidget(self.widthBox)
        self.layout2.addWidget(self.labelHeight)
        self.layout2.addWidget(self.heightBox)
        self.button = QtWidgets.QPushButton('Set', self)
        self.button.resize(100, 50)
        self.button.clicked.connect(self.getVal)
        self.layout3.addLayout(self.layout1)
        self.layout3.addLayout(self.layout2)
```

```
        self.layout3.addWidget(self.button)
        self.setLayout(self.layout3)
        self.setFixedWidth(300)
        self.setFixedHeight(100)
        self.setWindowModality(QtCore.Qt.WindowModal)
        self.setModal(True)
        self.setWindowFlag(Qt.WindowContextHelpButtonHint, False)
        self.exec_()

    def getVal(self):
        self.widthVal = self.widthBox.value()
        self.heightVal = self.heightBox.value()
        self.close()
```

Am adăugat metode de ”redo” și ”undo”. Reține într-o listă câte o copie a imaginii de fiecare dată când se produce o schimbare. De asemenea, am o variabilă care reține poziția imaginii în listă la momentul curent, dacă utilizatorul a dat ”undo”. Poziția scade de la dreapta la stânga pe cât utilizatorul avansează la primele stadii ale imaginii (primele copii reținute în listă). Variabila e, în mod normal, -1, și își schimbă valoarea la undo. Dacă s-a dat ”CTRL+Z”, și apoi s-a făcut o schimbare în imagine, utilizatorul nu mai poate da ”redo”, pentru că ultimele stări ale imaginii înainte de undo se pierd, și vor fi șterse din listă, iar variabila redevine -1. Am folosit ilustrații (icons) pentru toate opțiunile, incluzând undo și redo, și majoritatea opțiunilor au și scurtături, cum e CTRL+Z pentru undo. Ilustrațiile devin gri/alb-negru, dacă utilizatorul nu are voie să utilizeze acțiunea la stadiul curent (cum ar fi nu mai poate da înainte sau înapoi), și butoanele sau acțiunile în acest caz sunt dezactivate. Am descărcat toate ilustrațiile utilizate în aplicație gratuit de pe icons8.com[19].

```
self.scribbleArea.undoAct = QtWidgets.QAction(self.scribbleArea.undoGrayIcon,
self.tr("Undo"), self)
self.scribbleArea.undoAct.setEnabled(False)
self.scribbleArea.undoAct.triggered.connect(self.undo)
self.scribbleArea.undoAct.setShortcut(self.tr("Ctrl+Z"))

self.scribbleArea.redoAct = QtWidgets.QAction(self.scribbleArea.redoGrayIcon,
self.tr("Redo"), self)
self.scribbleArea.redoAct.setEnabled(False)
self.scribbleArea.redoAct.triggered.connect(self.redo)
self.scribbleArea.redoAct.setShortcut(self.tr("Ctrl+Y"))
```

```
def undo(self):
    lista = [self.scribbleArea, self.hiddenArea]
    for area in lista:
        if area.undo_pos == -1:
            area.undo_pos = len(area.undo_images)-2
            if area.format == area.rgb:
                area.redoAct.setEnabled(True)
                area.redoAct.setIcon(area.redoIcon)
        elif area.undo_pos > 0:
            area.undo_pos -= 1
```

```
            if area.format == area.rgb:
                area.redoAct.setEnabled(True)
                area.redoAct.setIcon(area.redoIcon)
        elif area.format == area.rgb:
#nu mai poate da inapoi
            area.undoAct.setEnabled(False)
            area.undoAct.setIcon(area.undoGrayIcon)
        if area.format == area.rgb and area.undo_pos == 0:
            area.undoAct.setEnabled(False)
            area.undoAct.setIcon(area.undoGrayIcon)
        area.image = area.undo_images[area.undo_pos].copy()
        area.modified = True
        area.update()
```

```
def redo(self):
    lista = [self.scribbleArea, self.hiddenArea]
    for area in lista:
        if area.format == area.rgb:
            area.undoAct.setEnabled(True)
            area.undoAct.setIcon(area.undoIcon)
        if area.undo_pos > -1:
            area.undo_pos += 1
        if area.format == area.rgb and area.undo_pos == (len(area.undo_images)
- 1):
            area.redoAct.setEnabled(False)
            area.redoAct.setIcon(area.redoGrayIcon)
        area.image = area.undo_images[area.undo_pos].copy()
        area.modified = True
        area.update()
```

Observați că în cod am scribbleArea (pentru imaginea văzută de user), și hiddenArea (imaginea ascunsă). Prima e în format RGB și ultima e în RGBA cu fon transparent. Fiecare schimbare care se produce pe scribbleArea, e repetată pe hiddenArea. Totuși, aceasta nu a părut să afecteze performanța aplicației (faptul că am dublat practic toate operațiile de desen), fiindcă operațiile de desen se execută extrem de rapid. Astfel, userul poate salva imaginea în format RGB și RGBA la dorință.

Am re-implementat modul în care funcționează radiera, fiindcă funcția eraseRect() din QPainter, făcută în special pentru a șterge, e numai pentru format RGBA. Pe imaginea principală (voi numi scribbleArea imaginea principală), ea lăsa urme negre pe porțiunile pe care le ștergea. motivul este faptul că funcția nu face decât să înlocuiască porțiunea de o dimensiune setată cu culoare transparentă (da, există așa ceva), care pentru RGB este inexistentă. Pentru imaginea principală am făcut alt tip de radieră – un pix de culoare albă. Pe ambele imagini se șterg aceleași porțiuni de exactă mărime. Aici clasa ScribbleArea cu 2 tipuri de format (RGB și RGBA):

```
class ScribbleArea(QtWidgets.QWidget):
    def __init__(self, format, parent=0):
        super().__init__()
        self.width = 700
```

```
        self.height = 700
        self.parent = parent
        self.modified = False
        self.scribbling = False
        self.myPenColor = QtGui.QColor(0, 0, 255, 255)
#blue
        self.myPenWidth = 1
        self.cap = Qt.RoundCap
        self.join = Qt.RoundJoin
        self.line = Qt.SolidLine
        self.pen = QtGui.QPen(self.myPenColor, self.myPenWidth, self.line,
self.cap, self.join)
        self.hasPen = True                                      #foloseste
creionul la moment (are creionul activ)
        self.hasEraser = False
#foloseste radiera
        self.hasBrush = False
        self.hasBlur = False
        self.blurRadius = 7
        self.hasBucket = False
        self.erasing = False
        self.hasCrop = False
        self.rubberBand = None
        self.croppedRect = None
        self.eraserWidth = 5
        self.lastPoint = None
        self.format = format
        self.rgb = QtGui.QImage.Format_RGB32
        self.rgba = QtGui.QImage.Format_RGBA64
        self.setAttribute(QtCore.Qt.WA_StaticContents)
        self.brush = None
        self.texture = None
        self.texture_name = None
        self.rainbow = False
        self.undoIcon = QtGui.QIcon(":undo")
        self.redoIcon = QtGui.QIcon(":redo")
        self.undoGrayIcon = QtGui.QIcon(":undo_gray")
        self.redoGrayIcon = QtGui.QIcon(":redo_gray")
        self.saveIcon = QtGui.QIcon(":save")
        self.saveGrayIcon = QtGui.QIcon(":save_gray")
        self.saveRGBIcon = QtGui.QIcon(":rgb")
        self.saveRGBGrayIcon = QtGui.QIcon(":rgb_gray")
        self.saveRGBAIcon = QtGui.QIcon(":rgba")
        self.saveRGBAGrayIcon = QtGui.QIcon(":rgba_gray")
        self.cropIcon = QtGui.QIcon(":crop")
        cropSelectIcon = QtGui.QPixmap(":crop_select")
#marime 32 pixeli
        self.cursor_crop = QtGui.QCursor(cropSelectIcon, 0, 0)
        self.last_saved_filename = None
        self.colors = QtGui.QColor.colorNames()
        self.hasSpray = False
        self.texture_size = QtCore.QSize(30, 30)
        self.texture_icons = icon_func()
        self.brush_textures = texture_func(self.texture_size)
        cursor_pixmap = QtGui.QPixmap(":bucket_32")
#marime 32 pixeli
        self.cursor_bucket = QtGui.QCursor(cursor_pixmap, 0, 0)
```

```
        self.undo_pos = -1
        self.image = QtGui.QImage(self.width, self.height, format)
        if self.format == QtGui.QImage.Format_RGB32:
            self.image.fill(QtGui.QColor(255, 255, 255))
            self.file_filter = 'Bitmap (*.bmp);; Cursor (*.cur);; Icon macOS
(*.icns);; Icon Windows (*.ico);; ' \
                               'Joint Photographic Experts Group (*.jpeg);;
Portable Bitmap (*.pbm);; Portable Grayscale (*.pgm);; ' \
                               'Portable Network Graphic (*.png);; Portable
Colored (*.ppm);; Tagged Image Format (*.tif);;' \
                               'Wireless Application Protocol Bitmap
(*.wbmp);; WebP (*.webp);; X Bitmap Graphic (*.xbm);;' \
                               'X PixMap (*.xpm)'
        else:
            self.image.fill(QtCore.Qt.transparent)
            self.file_filter = 'Tagged Image Format (*.tif) ;; Portable
Network Graphic (*.png);;Cursor (*.cur);; ' \
                               'Icon macOS (*.icns);; Icon Windows (*.ico);;
WebP (*.webp);;' \
                               'X PixMap (*.xpm)'
        self.undo_images = [self.image.copy()]
```

Observați `self.image.fill(QtCore.Qt.transparent)` pentru formatul RGBA.

Aici am re-implementat radiera, pentru cele 2 formaturi:

```
def erase(self, endPoint):
    painter = QtGui.QPainter(self.image)
    if self.format == self.rgba:
        r = QtCore.QRect(QtCore.QPoint(), QtCore.QSize(5*self.eraserWidth,
5*self.eraserWidth))
        r.moveCenter(endPoint)
        painter.save()
        painter.setCompositionMode(QtGui.QPainter.CompositionMode_Clear)
        painter.eraseRect(r)
    else:
        painter.setPen(QtGui.QPen(QtGui.QColor(255, 255, 255),
5*self.eraserWidth, QtCore.Qt.SolidLine, QtCore.Qt.RoundCap,
QtCore.Qt.RoundJoin))
        painter.drawLine(self.lastPoint, endPoint)
    self.modified = True
    self.update()
    self.lastPoint = endPoint
```

III.1.1 Clasa QPen[16]

Clasa QPen are implementat în QPainter diverse efecte basic.

a) Line Styles (figura 3.1.1.1):

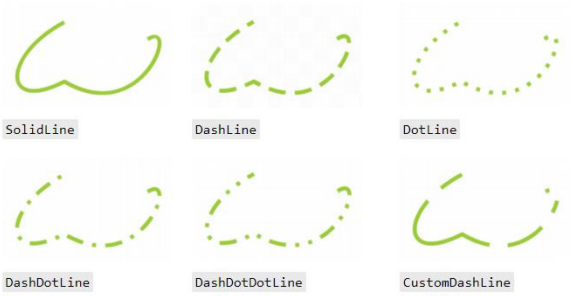

Figura 3.1.1.1: Line Styles pentru QPen

```
self.line_acts = []

dashDotDotLineIcon = QtGui.QIcon(":DashDotDotLine")
dashDotLineIcon = QtGui.QIcon(":DashDotLine")
dashLineIcon = QtGui.QIcon(":DashLine")
dotLineIcon = QtGui.QIcon(":DotLine")
solidLineIcon = QtGui.QIcon(":SolidLine")
dashDotDotAct = QtWidgets.QAction(dashDotDotLineIcon, self.tr("Dash Dot Dot
Line"), self)
dashDotAct = QtWidgets.QAction(dashDotLineIcon, self.tr("Dash Dot Line"),
self)
dashAct = QtWidgets.QAction(dashLineIcon, self.tr("Dash Line"), self)
dotAct = QtWidgets.QAction(dotLineIcon, self.tr("Dot Line"), self)
solidAct = QtWidgets.QAction(solidLineIcon, self.tr("Solid Line"), self)

self.line_acts.append(dashDotDotAct)
self.line_acts.append(dashDotAct)
self.line_acts.append(dashAct)
self.line_acts.append(dotAct)
self.line_acts.append(solidAct)
```

```
for act in self.line_acts:
    act.triggered.connect(lambda ch, x=act.text(): self.setLine(x))
```

```
lineIcon = QtGui.QIcon(":line")
lineMenu = brushMenu.addMenu("Line styles")
lineMenu.setIcon(lineIcon)
```

```
def setLine(self, name):
    self.setCursor(Qt.ArrowCursor)
    line = None
    if name == "Dash Dot Dot Line":
        line = Qt.DashDotDotLine
    elif name == "Dash Dot Line":
        line = Qt.DashDotLine
    elif name == "Dash Line":
```

```
        line = Qt.DashLine
    elif name == "Dot Line":
        line = Qt.DotLine
    elif name == "Solid Line":
        line = Qt.SolidLine
    self.scribbleArea.line = line
    self.hiddenArea.line = line
```

b) Cap Styles (figura 3.1.1.2):

Figura 3.1.1.2: Cap Styles pentru QPen

```
self.cap_acts = []

flatCapIcon = QtGui.QIcon(":FlatCap")
roundCapIcon = QtGui.QIcon(":RoundCap")
squareCapIcon = QtGui.QIcon(":SquareCap")
flatCapAct = QtWidgets.QAction(flatCapIcon, self.tr("Flat Cap"), self)
roundCapAct = QtWidgets.QAction(roundCapIcon, self.tr("Round Cap"), self)
squareCapAct = QtWidgets.QAction(squareCapIcon, self.tr("Square Cap"), self)

self.cap_acts.append(flatCapAct)
self.cap_acts.append(roundCapAct)
self.cap_acts.append(squareCapAct)
```

```
for act in self.cap_acts:
    act.triggered.connect(lambda ch, x=act.text(): self.setCap(x))
```

```
def setCap(self, name):
    self.setCursor(Qt.ArrowCursor)
    cap = None
    if name == "Flat Cap":
        cap = Qt.FlatCap
    elif name == "Round Cap":
        cap = Qt.RoundCap
    else:
        cap = Qt.SquareCap

    self.scribbleArea.cap = cap
    self.hiddenArea.cap = cap
```

c) Join Styles (figura 3.1.1.3 și 3.1.1.4):

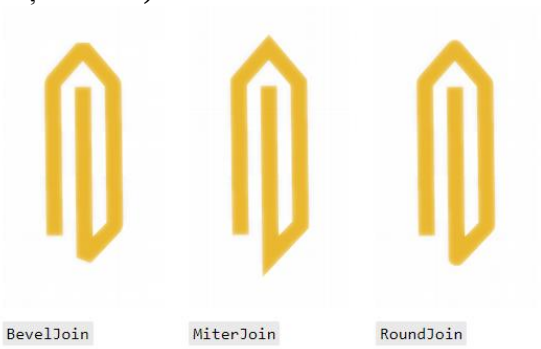

Figura 3.1.1.3: Join Styles pentru QPen

Join este modul în care se unesc liniile în timpul desenului. Default este BevelJoin (figura 5.1.1.4):

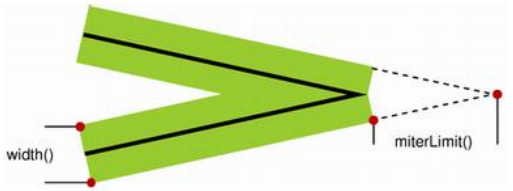

Figura 3.1.1.4: BevelJoin

```
self.join_acts = []

bevelJoinIcon = QtGui.QIcon(":BevelJoin")
miterJoinIcon = QtGui.QIcon(":MiterJoin")
roundJoinIcon = QtGui.QIcon(":RoundJoin")
bevelJoinAct = QtWidgets.QAction(bevelJoinIcon, self.tr("Bevel Join"), self)
miterJoinAct = QtWidgets.QAction(miterJoinIcon, self.tr("Miter Join"), self)
roundJoinAct = QtWidgets.QAction(roundJoinIcon, self.tr("Round Join"), self)

self.join_acts.append(bevelJoinAct)
self.join_acts.append(miterJoinAct)
self.join_acts.append(roundJoinAct)
```

```
for act in self.join_acts:
    act.triggered.connect(lambda ch, x=act.text(): self.setJoin(x))
```

```
def setJoin(self, name):
    self.setCursor(Qt.ArrowCursor)
    join = None
    if name == "Bevel Join":
        join = Qt.BevelJoin
    elif name == "Miter Join":
        join = Qt.MiterJoin
    else:
        join = Qt.RoundJoin
    self.scribbleArea.join = join
    self.hiddenArea.join = join
```

III.1.2 Clasa QBrush[17]

Clasa QBrush, de asemenea, are câteva chestii implementate:

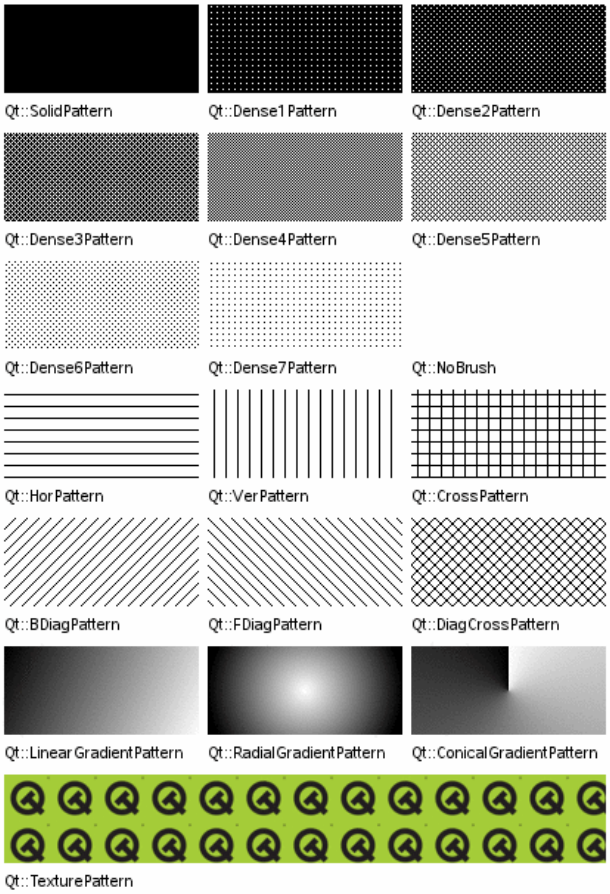

Figura 3.1.2.1: Texturi și Gradienți existenți în clasa QBrush

a) Patterns:

Texturile sunt primele 15 din figura 3.1.2.1 (până la gradienți), plus textura custom (ultima).

Eu mi-am consturuit câteva texturi proprii pentru pensule (va urma), întrucât acestea nu mi s-au părut la fel cu texturile pe care te aștepți să le vezi într-o aplicație pentru desen (voi explica de ce aceste texturi m-au dezamăgit, inclusiv textura ”custom”).

Texturile la care mă aștept într-o aplicație pentru desen (figura 3.1.2.2):

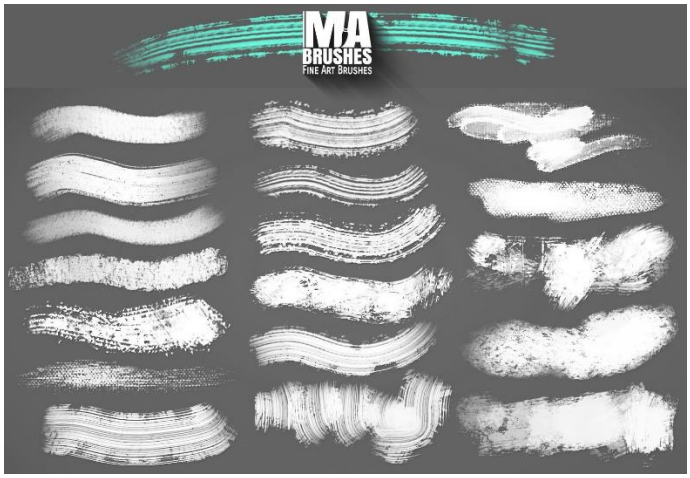

Figura 3.1.2.2: Texturile dorite de mine

Motivul pentru care astfel de texturi sunt necesare pentru aplicația mea: Userul trebuie să poată simula stilul unor imagini abstracte pe care le am la îndemână[22](am luat desene abstracte de pe un site care le vinde în format fizic) – în acril și acuarelă (Exemplu, figura 3.1.2.3):

Figura 3.1.2.3: H-Nguyen - Green Mantra [5]

Diferența dintre texturile propuse de clasa QBrush și de ce nu se potrivesc cu scopul aplicației mele:

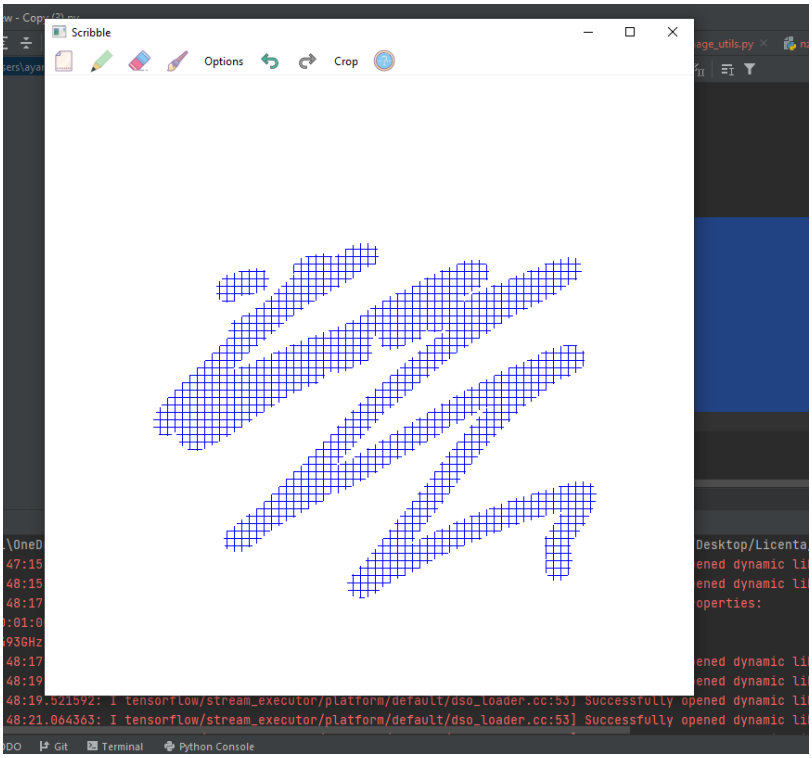

Figura 3.1.2.4: ”Dezamăgire”

După cum observați în figura 3.1.2.4, texturile din clasa QBrush se setează ca background ascuns al imaginii, care ”iese la iveală” dacă se desenează deasupra lui. Textura ”custom”, ultima din listă, face același lucru, doar cu o imagine aleasă de user. Mai mult de atât, printre ele nu există efect de textură de acril sau acuarelă, sau ceva asemănător.

Cazul în care descarc o ”mostră”(figura 3.1.2.5) de textură și o setez ca textură ”custom”:

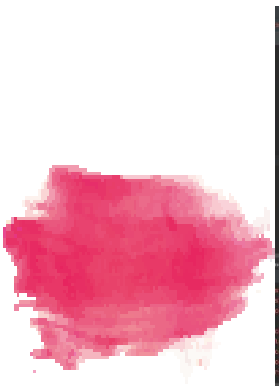

Figura 3.1.2.5: ”mostră”

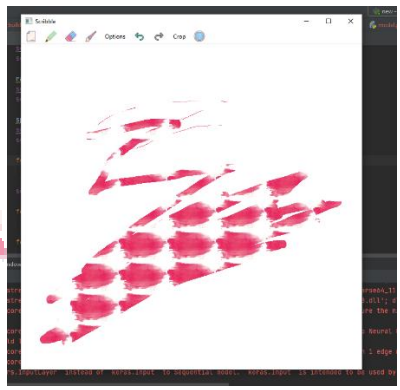

Figura 3.1.2.6: ”Dezamăgire II”

Voi explica cum am trecut de această problemă și mi-am făcut propriile texturi pentru pensule în următorul subcapitol.

b) Gradients:

Aici voi arăta implementările pentru texturile deja existente, și gradienți. Gradienții, totuși, au necesitat inputuri de la user. Cum ar fi, gradientul liniar – o trecere de la o culoare la alta, în mod liniar, necesită cel puțin 2 culori, și pozițiile focarelor lor (să spunem că am putea avea o trecere a culorilor de la stânga la dreapta sau de sus în jos, depinde de poziția culorilor). Voi cere de la user numărul de culori, și voi prezenta, pentru fiecare, o fereastră pop-up, copie a imaginii pe care desenează, pe care să dea click pentru a indica poziția culorii. Fereastra primește inputul și se închide.

La fel, gradientul radial – trece prin diverse culori, și are un singur focar. Prima culoare e cercul/centrul gradientului. Aici de asemenea cer poziția focarului, o singură poziție.

Pentru gradientul conic cer poziția focarului și unghiul în grade, până la 360 grade.

```
def setBrush(self, name):
    self.setCursor(Qt.ArrowCursor)
    self.scribbleArea.hasBrush = True
    self.scribbleArea.hasPen = False
    self.scribbleArea.hasEraser = False
    self.scribbleArea.hasBucket = False
    self.scribbleArea.texture = None

    self.hiddenArea.hasBrush = True
    self.hiddenArea.hasPen = False
    self.hiddenArea.hasEraser = False
    self.hiddenArea.hasBucket = False
    self.hiddenArea.texture = None                                  #custom
texture

    pattern = None
    grad = None
    if name == "Dense 1 Pattern":
        pattern = QtCore.Qt.Dense1Pattern
    elif name == "Dense 2 Pattern":
        pattern = QtCore.Qt.Dense2Pattern
    elif name == "Dense 3 Pattern":
        pattern = QtCore.Qt.Dense3Pattern
    elif name == "Dense 4 Pattern":
        pattern = QtCore.Qt.Dense4Pattern
    elif name == "Dense 5 Pattern":
        pattern = QtCore.Qt.Dense5Pattern
    elif name == "Dense 6 Pattern":
        pattern = QtCore.Qt.Dense6Pattern
    elif name == "Dense 7 Pattern":
        pattern = QtCore.Qt.Dense7Pattern
    elif name == "Horizontal Pattern":
        pattern = QtCore.Qt.HorPattern
    elif name == "Vertical Pattern":
        pattern = QtCore.Qt.VerPattern
```

```
    elif name == "Cross Pattern":
        pattern = QtCore.Qt.CrossPattern
    elif name == "B Diagonal Pattern":
        pattern = QtCore.Qt.BDiagPattern
    elif name == "F Diagonal Pattern":
        pattern = QtCore.Qt.FDiagPattern
    elif name == "Diagonal Cross Pattern":
        pattern = QtCore.Qt.DiagCrossPattern
    elif name == "Linear Gradient Pattern":
        self.scribbleArea.rainbow = False
        self.hiddenArea.rainbow = False
        nrCol, ok = QtWidgets.QInputDialog.getInt(self, self.tr("Gradient"),
self.tr("Select number of colors for the linear gradient: "),
                                                  2, 2, 10, 1)
        lista = []
        if ok:
            pos1 = Coord(self.scribbleArea.image, "Select first focal point
position", self.scribbleArea.width, self.scribbleArea.height)
            pos2 = Coord(self.scribbleArea.image, "Select second focal point
position", self.scribbleArea.width, self.scribbleArea.height)
            for i in range(nrCol):
                newColor =
QtWidgets.QColorDialog.getColor(self.scribbleArea.myPenColor)
                slider = Slider("Select a position for the color", 0, 10, 0)
                if newColor and slider.val is not None:
                    lista.append((newColor, slider.val))
            if pos1.area.coord and pos2.area.coord and lista:
                grad = QtGui.QLinearGradient(pos1.area.coord, pos2.area.coord)

                for i in range(nrCol):
                    grad.setColorAt(lista[i][1]/10, lista[i][0])
        else:
            x = self.scribbleArea.geometry().x()
            y = self.scribbleArea.geometry().y()
            grad = QtGui.QLinearGradient(x, y,
self.scribbleArea.geometry().width() + x,

self.scribbleArea.geometry().height() + y)
            grad.setColorAt(0.0, self.scribbleArea.myPenColor)
            grad.setColorAt(0.5, QtCore.Qt.black)

    elif name == "Radial Gradient Pattern":
        self.scribbleArea.rainbow = False
        self.hiddenArea.rainbow = False
        nrCol, ok = QtWidgets.QInputDialog.getInt(self, self.tr("Gradient"),
                                                  self.tr("Select number of
colors for the radial gradient: "),
                                                  2, 2, 10, 1)
        if ok:
            print("ok")
            lista = []
            pos1 = Coord(self.scribbleArea.image, "Select center position",
                         self.scribbleArea.width, self.scribbleArea.height)
            slider1 = Slider("Select gradient radius:", 1, 1000, 1)
            for i in range(nrCol):
                newColor =
QtWidgets.QColorDialog.getColor(self.scribbleArea.myPenColor)
```

```
                slider2 = Slider("Select a position for the color", 0, 10, 0)
                if newColor and slider2.val is not None:
                    lista.append((newColor, slider2.val))
            if pos1.area.coord and lista and slider1.val:
                grad = QtGui.QRadialGradient(pos1.area.coord, slider1.val)

                for i in range(nrCol):
                    grad.setColorAt(lista[i][1]/10, lista[i][0])
        else:
            x = self.scribbleArea.geometry().x()
            y = self.scribbleArea.geometry().y()
            width = self.scribbleArea.geometry().width()
            height = self.scribbleArea.geometry().height()
            grad = QtGui.QRadialGradient(x + width/2, y + height/2, min(x +
width/2, y + height/2))
            grad.setColorAt(0.0, self.scribbleArea.myPenColor)
            grad.setColorAt(0.5, QtCore.Qt.black)

    elif name == "Conical Gradient Pattern":
        self.scribbleArea.rainbow = False
        self.hiddenArea.rainbow = False
        nrCol, ok = QtWidgets.QInputDialog.getInt(self, self.tr("Gradient"),
                                                  self.tr("Select number of
colors for the conical gradient: "),
                                                  2, 2, 10, 1)
        if ok:
            lista = []
            pos1 = Coord(self.scribbleArea.image, "Select center position",
                         self.scribbleArea.width, self.scribbleArea.height)
            slider1 = Slider("Select Select angle (1° - 359°) :", 1, 359, 1)
            for i in range(nrCol):
                newColor =
QtWidgets.QColorDialog.getColor(self.scribbleArea.myPenColor)
                slider2 = Slider("Select a position for the color", 0, 10, 0)
                if newColor and slider2.val is not None:
                    lista.append((newColor, slider2.val))
            if pos1.area.coord and lista and slider1.val:
                grad = QtGui.QConicalGradient(pos1.area.coord, slider1.val)

                for i in range(nrCol):
                    grad.setColorAt(lista[i][1] / 10, lista[i][0])
        else:
            x = self.scribbleArea.geometry().x()
            y = self.scribbleArea.geometry().y()
            width = self.scribbleArea.geometry().width()
            height = self.scribbleArea.geometry().height()
            grad = QtGui.QConicalGradient(x + width / 2, y + height / 2, 30)
            grad.setColorAt(0.0, self.scribbleArea.myPenColor)
            grad.setColorAt(0.5, QtCore.Qt.black)
    else:
        fileName, _filter = QtWidgets.QFileDialog.getOpenFileName(self,
self.tr("Open File"), QtCore.QDir.currentPath())
        if fileName:
            image = QtGui.QImage(fileName)
            self.scribbleArea.brush = QtGui.QBrush(image)
            self.hiddenArea.brush = QtGui.QBrush(image)
    if name != "Texture Pattern..." and not grad:
```

```
        self.scribbleArea.brush = QtGui.QBrush(self.scribbleArea.myPenColor,
pattern)
        self.hiddenArea.brush = QtGui.QBrush(self.hiddenArea.myPenColor,
pattern)
    elif grad:
        self.scribbleArea.brush = QtGui.QBrush(grad)
        self.hiddenArea.brush = QtGui.QBrush(grad)
```

Exemplu a cum arată (figura 3.1.2.7):

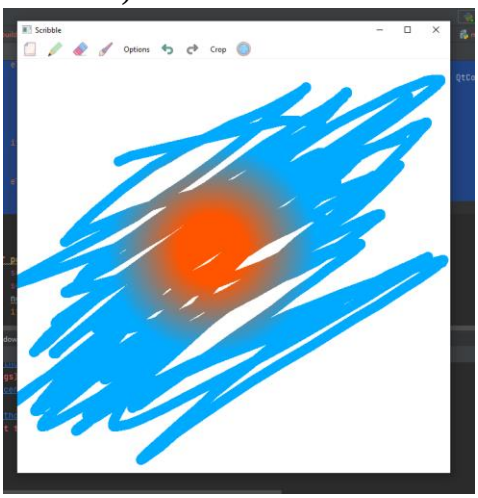

Figura 3.1.2.7: Radial Gradient cu 2 culori random alese de user

III.1.3 Implementări custom (inclusiv texturi)

a) Texturi pentru pensule:

Am descărcat ”mostre” de texturi de pe un website[18] care oferă o gamă largă de imagini cu background transparent, câteva exemple dintre cele pe care le utilizez în aplicație (figura 3.1.3.1):

Figura 3.1.3.1: Texturile custom utilizate de mine

Mi-a venit ideea după un post văzut pe StackOverflow[14], care e implementat în C# (vedeți un gif în post). Simulez același lucru, practic după ce userul alege una din texturi/imaginile de mai sus, o redimensionez la un size (o variabila care poate fi modificată la dorința userului), și o ”desenez” pe canvas pentru fiecare poziție/pixel parcurs de user cu ”pixul”.

Pentru a desena o linie, cu QPen sau QBrush, există funcția drawLine. Iată cum arată funcția pentru desen în mod obișnuit:

```
def drawLineTo(self, endPoint):
    painter = QtGui.QPainter(self.image)
    if not self.rainbow and self.texture is None:     #scribbling, pen or
```

```
brush
        if self.hasBrush:
            painter.setPen(QtGui.QPen(self.brush, self.myPenWidth, self.line,
                                      self.cap, self.join))
        elif self.hasPen:
            painter.setPen(QtGui.QPen(self.myPenColor, self.myPenWidth,
self.line,
                                      self.cap, self.join))
        if not self.hasSpray and self.texture is None:
        painter.drawLine(self.lastPoint, endPoint)
        self.update()
        self.update()
    self.modified = True
    self.lastPoint = endPoint
```

chemată la fiecare mișcare cu mouse-ul în timpul click-ului. Nu am indicat funcția completă, pentru că mai conține ustensile custom implementate despre care urmează să vorbesc.

Acum, în loc de a desena linia cu painter.drawLine(pos1, pos2), trebuie doar să identific care e linia, care sunt pixelii ei, și să desenez textura descărcată de mine per fiecare pixel.

Funcția QtCore.QLineF(pos1, pos2) returnează o linie, dar nu o desenează.
Am aflat cum determin pixelii ei, cu ajutorul unui post pe StackOverflow[1].

```
elif self.texture is not None:
    line = QtCore.QLineF(self.lastPoint, endPoint)
    lista = []
    for i in range(int(line.length())):
        x = line.x1() + i*math.cos(math.radians(line.angle()))
        y = line.y1() - i*math.sin(math.radians(line.angle()))
        point = QtCore.QPoint(int(x), int(y))
        lista.append(point)
    for pos in lista:
        painter.drawImage(pos, self.texture)
        self.update()
self.update()
self.modified = True
self.lastPoint = endPoint
```

(parte din funcția drawLineTo).

A funcționat (figura 3.1.3.2):

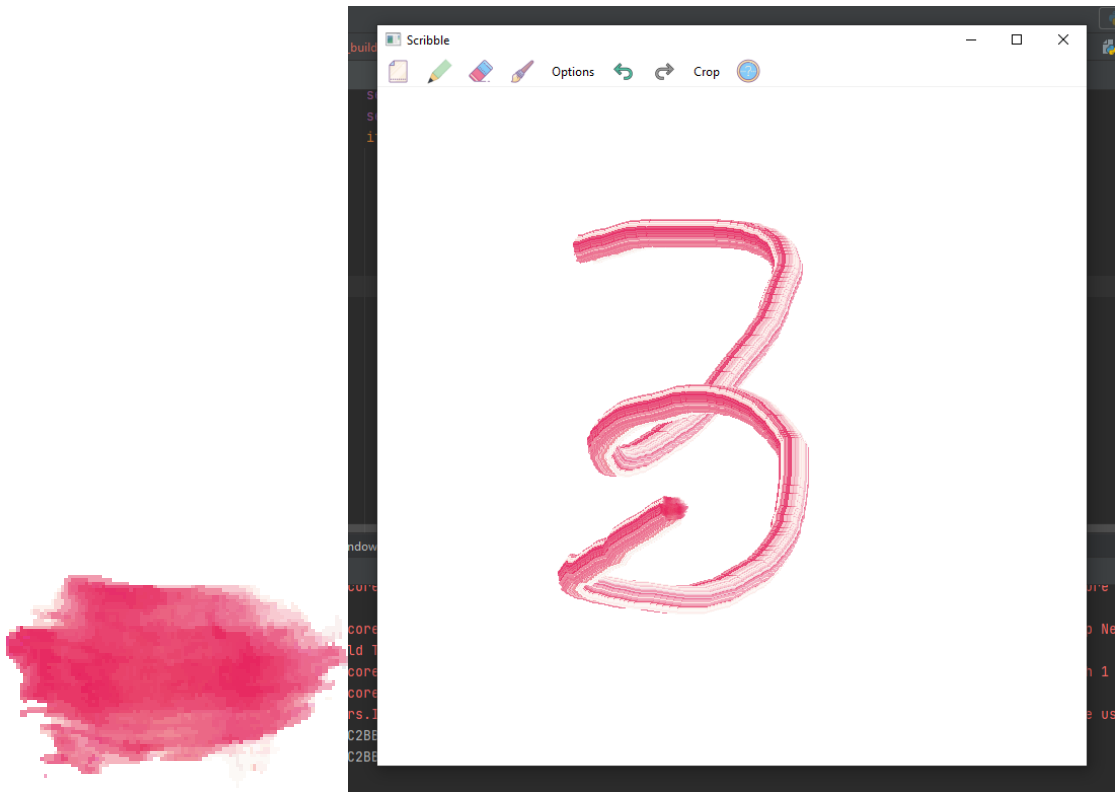

Figura 3.1.3.2: ”A funcționat!” Mostra și rezultatul

```
if "Brush" in name:
    idx = int(name[6:])
    image = QtGui.QImage("./resources/textures/brush"+str(idx)+".png")

    new = image.scaled(self.hiddenArea.texture_size)
    self.hiddenArea.texture = new
    self.scribbleArea.texture = new
```

Textura este setată la dimensiunea 30px (width and height), userul poate schimba.

Rămâne o problemă: **Vreau să pot utiliza texturile cu orice culoare aleasă de user.**

Am dat întrebarea pe StackOverflow[13] și am primit o soluție. Să convertesc imaginea în grayscale cu valori float între 0 și 1, și să înmulțesc fiecare pixel din imaginea grayscale cu fiecare canal de culoare RGB din culoarea pe care vreau să o obțin.

Fie culoarea rgba(0, 0, 255, 255), care e default pentru aplicația mea (figura 3.1.3.3):

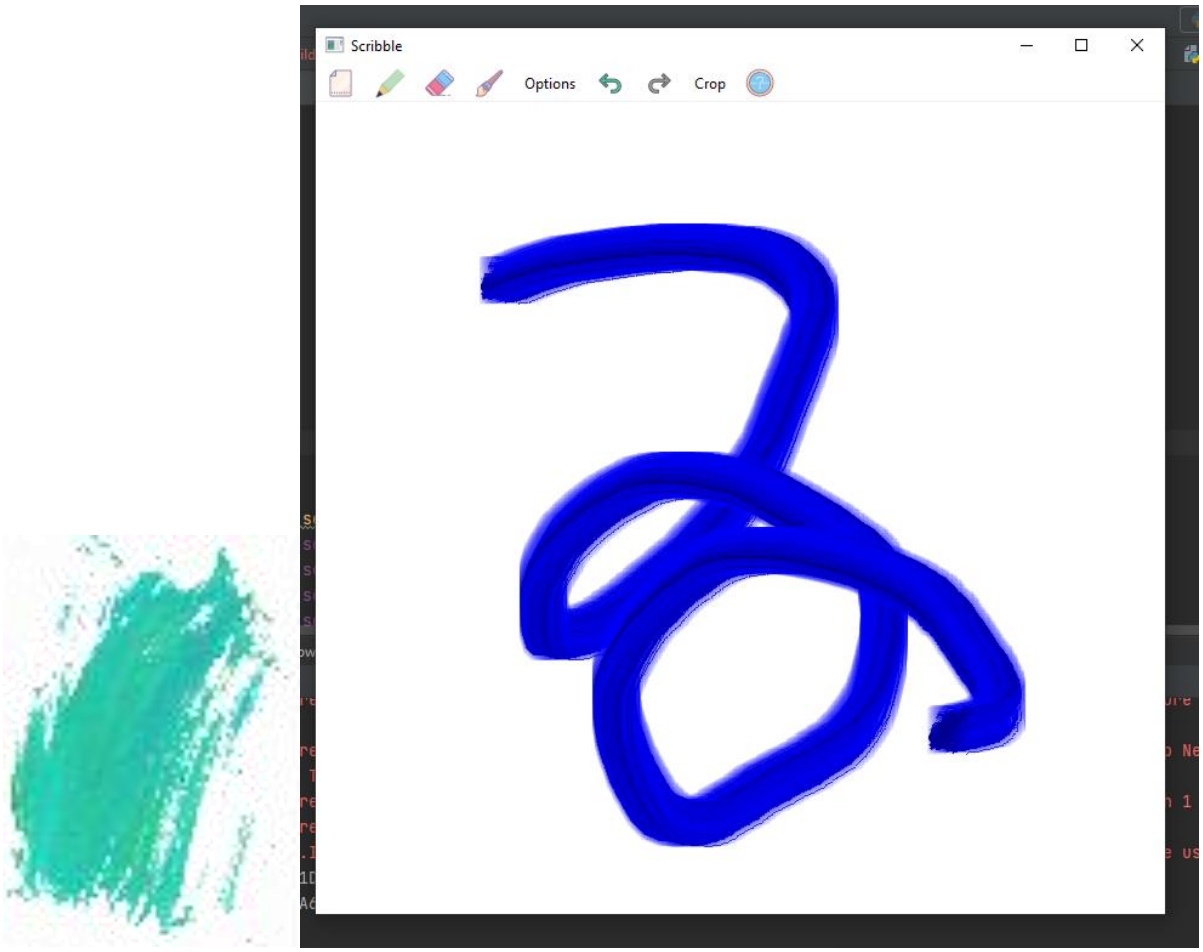

Figura 3.1.3.3: ”Evrica” Mostra și rezultatul

```
def setTexture(self, name):
    self.setCursor(Qt.ArrowCursor)

    self.hiddenArea.hasBrush = True
    self.hiddenArea.hasPen = False
    self.hiddenArea.hasEraser = False
    self.hiddenArea.hasBucket = False
    self.hiddenArea.hasSpray = False
    self.hiddenArea.rainbow = False
    self.hiddenArea.texture_name = name

    self.scribbleArea.hasBrush = True
    self.scribbleArea.hasPen = False
    self.scribbleArea.hasEraser = False
    self.scribbleArea.hasBucket = False
    self.scribbleArea.hasSpray = False
    self.scribbleArea.rainbow = False
    self.scribbleArea.texture_name = name
    if "Brush" in name:
```

```
        idx = int(name[6:])
        image = img.open("./resources/textures/brush"+str(idx)+".png",
formats=None)
        target_col = self.hiddenArea.myPenColor
        my_blue = target_col.blue()
        my_red = target_col.red()
        my_green = target_col.green()
        target_col = np.array((my_red, my_green, my_blue))
        image_np = np.asarray(image)
        alpha = image_np[:, :, -1:]
        image_rgb = image_np[:, :, :-1]
        imgGray = image.convert('L')
        image_array = np.asarray(imgGray)
        image_float = image_array / 255

        new_img = np.zeros(image_rgb.shape)

        for i in range(len(image_float)):
            for j in range(len(image_float[0])):
                new_img[i][j] = image_float[i][j] * target_col

        new_img = np.concatenate((new_img, alpha), axis=2)
        output_image = img.fromarray(new_img.astype(np.uint8))
        output_image.save("output.png")

        new = QtGui.QImage("output.png")

        new = new.scaled(self.hiddenArea.texture_size)
        self.hiddenArea.texture = new
        self.scribbleArea.texture = new
```

b) ”Bucket” Tool:

Am împrumutat codul de pe internet[10]. Acest tool colorează întreg spațiul alb din jurul poziției pe care s-a dat click și se oprește când întâlnește o linie/obstacol.

Algoritmul este simplu, la click se pune în funcțiune un ”event”, care mereu deține poziția clickului. Având poziția, un tuplu (x,y), îl punem într-o coadă. Pentru toate pozițiile din coadă, pe rând: le scoatem din coadă, identificăm toți pixelii din jurul acestui tuplu, dacă sunt albi îi colorăm și îi punem în coadă. Ne oprim până coada nu e goală. Algoritmul are o problemă: e departe de a fi optim, dar nu am identificat o metodă mai bună.

```
def getCardinalPoints(self, haveSeen, centerPos, w, h):
    points = []
    cx, cy = centerPos
    for x, y in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
        xx, yy = cx + x, cy + y
        if (xx >= 0 and xx < w and yy >= 0 and yy < h and (xx, yy) not in
haveSeen):
            points.append((xx, yy))
            haveSeen.add((xx, yy))
    return points
```

în funcția mousePressEvent(self, event):

```
painter.setPen(QtGui.QPen(self.myPenColor, self.myPenWidth, self.line,
                          self.cap, self.join))
while queue:
    x, y = queue.pop()
    if self.image.pixel(x, y) == target_color:
        painter.drawPoint(QtCore.QPoint(x, y))
        queue[0:0] = self.getCardinalPoints(haveSeen, (x, y), w, h)
self.update()
```

c) ”Spray” Tool:

De asemenea, am împrumutat de pe internet[11], dar am făcut schimbări:

în funcția drawLineTo():

```
elif self.hasSpray:
#hasSpray
    pen = painter.pen()
    pen.setWidth(1)
    painter.setPen(pen)
    for n in range(self.myPenWidth*10 + 5):
        xo = random.gauss(0, 4 + self.myPenWidth)
        yo = random.gauss(0, 4 + self.myPenWidth)
        painter.drawPoint(int(self.lastPoint.x() + xo), int(self.lastPoint.y()
+ yo))
```

Acest algoritm plasează puncte random, de culoare curentă, într-un cerc cu diametru dependent de grosimea pixului/pensulei, și în număr de asemenea dependent de grosime. Am adus doar aceste schimbări. Astfel, punctele nu sunt nici prea abundente, nici prea departe unul de altul.

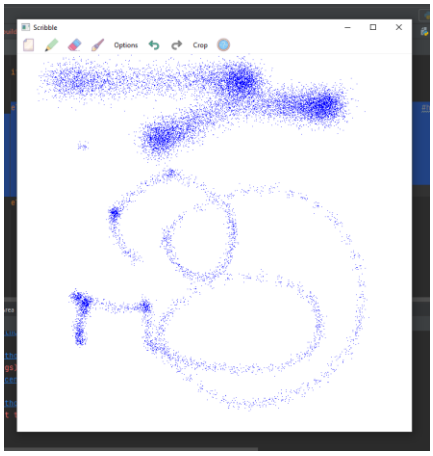

Figura 3.1.3.4: Spray

În figura 3.1.3.4 jos, varianta de spray pentru grosime minimă a pixului (1px), și sus pentru grosime 5px.

d)”Rainbow” Effect:

Am preluat ideea din aceeași sursă[11]. În timpul desenului, de fiecare dată cand se mișcă mouse-ul în timpul desenului, se alege o culoare random uniform. Acest efect este valabil atât pentru pix, cât și pentru spray (figura 3.1.3.5).

în funcția drawLineTo():

```
elif self.rainbow:
#rainbow
    color = QtGui.QColor(choice(self.colors))
    if self.hasBrush:
        self.brush.setColor(color)
        painter.setPen(QtGui.QPen(self.brush, self.myPenWidth, self.line,
                                  self.cap, self.join))
    elif self.hasPen:
        painter.setPen(QtGui.QPen(color, self.myPenWidth, self.line,
                                  self.cap, self.join))
if not self.hasSpray and self.texture is None:
    painter.drawLine(self.lastPoint, endPoint)
    self.update()
```

```
self.colors = QtGui.QColor.colorNames()
```

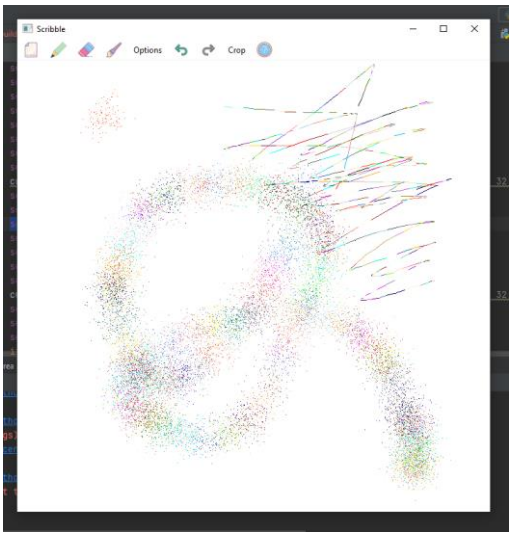

Figura 3.1.3.5: Rainbow Effect pentru pix și spray

Ca o ultimă observație pentru aplicația pentru desen, am refăcut funcțiile inițiale pentru salvarea imaginii și print:

```
def Print(self):
    try:
        from PyQt5.QtPrintSupport import QPrinter, QPrintDialog
    except ImportError as e:
        print("not print support")
    else:
        if self.image is None:
```

```
            return
        printer = QPrinter()
        printer.setPageSize(QPrinter.A4)
        printDialog = QPrintDialog(printer, self)
        if printDialog.exec_() == QtWidgets.QDialog.Accepted:
            painter = QtGui.QPainter(printer)
            rect = painter.viewport()
            image_aux = self.image.copy().scaled(rect.size(),
QtCore.Qt.KeepAspectRatio, Qt.SmoothTransformation)
#A4
            painter.drawImage(0, 0, image_aux)
            del painter
```

```
def saveFile(self, format):
    if format == self.scribbleArea.rgb:
        fileName, _ = QtWidgets.QFileDialog.getSaveFileName(
            parent=self,
            caption=self.tr("Save As"),
            directory=os.getcwd()+"/NewFile",
            filter=self.scribbleArea.file_filter
        )
        if not fileName:
            return False
        else:
            self.scribbleArea.last_saved_filename = fileName
            self.saveRGBAct.setEnabled(True)
            self.saveRGBAct.setIcon(self.scribbleArea.saveRGBIcon)
            self.saveMenu.setIcon(self.scribbleArea.saveIcon)
            return self.scribbleArea.saveImage(fileName)
    else:
        fileName, _ = QtWidgets.QFileDialog.getSaveFileName(
            parent=self,
            caption=self.tr("Save As"),
            directory=os.getcwd() + "/NewFile",
            filter=self.hiddenArea.file_filter
        )
        if not fileName:
            return False
        else:
            self.hiddenArea.last_saved_filename = fileName
            self.saveRGBAAct.setEnabled(True)
            self.saveMenu.setIcon(self.scribbleArea.saveIcon)
            self.saveRGBAAct.setIcon(self.scribbleArea.saveRGBAIcon)
            return self.hiddenArea.saveImage(fileName)
```

și am schimbat mărimea ilustrațiilor icons, custom, după o postare de pe StackOverflow[7]

```
class MyProxyStyle(QtWidgets.QProxyStyle):
    pass
    def pixelMetric(self, QStyle_PixelMetric, option=None, widget=None):

        if QStyle_PixelMetric == QtWidgets.QStyle.PM_SmallIconSize:
            return 25
        else:
```

```
            return QtWidgets.QProxyStyle.pixelMetric(self, QStyle_PixelMetric,
option, widget)
```

```
app = QtWidgets.QApplication(sys.argv)
myStyle = MyProxyStyle('Plastique')
app.setStyle(myStyle)
window = App()
window.show()
app.exec()
```

III.2 Modelul NST

După cum am spus în rezumat, am făcut research pe un model NST deja existent[12].

La fel ca primul model din 2015 [8], pentru partea de procesare a imaginii conținut s-a utilizat un model preantrenat pe image recognition, pe datasetul ImageNet[20]. Procesul de ”extragere” a conținutului dintr-o imagine a fost explicat în preliminarii în subcapitolul Rețelelor Convoluționale și subcapitolul NST. Modelul preantrenat poartă numele de ”InceptionV3”[23], și acesta este unul din modelele oficiale din biblioteca TensorFlow. Orice layer preantrenat InceptionV3 poate fi accesat din tensorflow și utilizat în propriul model, ca transfer de task, ceea ce s-a întâmplat și aici.

Pentru partea de transfer de stil s-a utilizat un alt model [4]. Un model din ”magenta”, un proiect inițiat de Google Brain Team[21]. Magenta conține modele AI pe partea de artă, ce procesează imagini și muzică.

Revenind la modelul NST, arhitectura 2 pentru acest model utilizează rețele convoluționale, dar printre ele mai există și câteva blocuri mai interesante – **Residual Blocks** (figura 3.2.1).

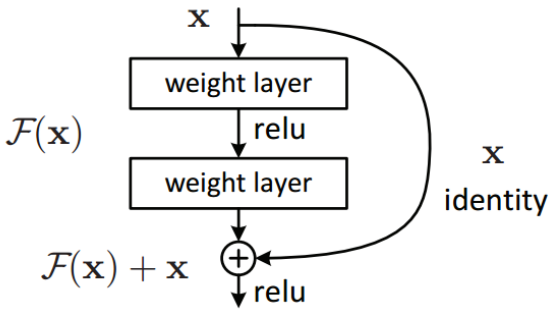

Figura 3.2.1: Residual Block

Într-o rețea adâncă (după cum spuneam, una cu multe layere), poate apărea problema de **vanishing gradient**. Însemnând că loss-ul scade treptat, dar la un moment dat se blochează și nu mai continuă să scadă spre rezultatul dorit, sau chiar începe să crească din nou. Cu cât are mai multe rețele, cu atât modelul învață mai greu. Un residual block se mai numește ”skip-

connection”, el transmite informația de la un lower-layer direct la un layer mai sus, deși ele sunt despărțite de 1 sau mai multe alte layere.

Motivul pentru care un model cu mai multe layere învață mai greu:

Trainingul are loc prin **backpropagation**. Fiecare layer are o matrice de weight și bias, care pentru un input x, dă un output w*x+b. Prin backpropagation, noi editām la fiecare epocă w și b, calculând loss-ul. Loss-ul, după cum spuneam, este ”diferența” dintre ce am obținut noi la epoca curentă și rezultatul dorit. De ce **”diferența”** și nu diferența propriu-zisă? Fiindcă funcțiile de loss diferă și pot fi alese, la fel ca funcțiile de activare, conform taskului. Un tip de funcție loss e chiar media diferenței în modul, MAE (Mean Absolute Error). Funcția de loss e doar pentru a compara rezultatele.

Prin backpropagation, se calculează derivata loss-ului (sau erorii) față de w și b al fiecărui layer. Pornind de la rezultatul final, derivatele se calculează prin **chain rule** (figura 3.2.2):

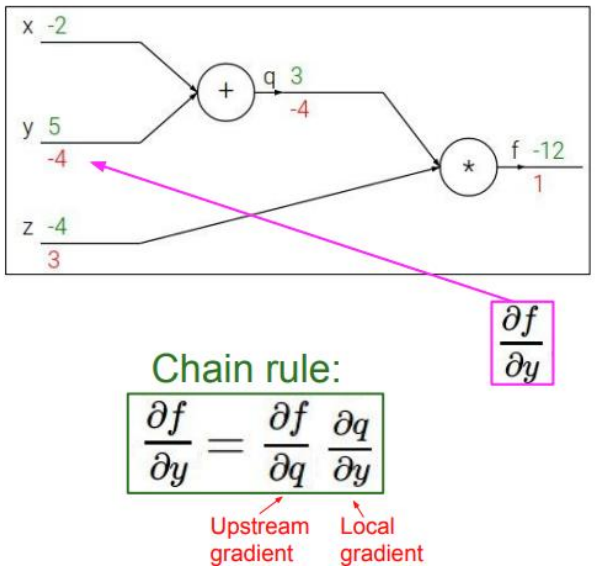

Figura 3.2.2: Chain Rule in Backpropagation

Backpropagation înseamnă calcularea **în direcție inversă** pornind de la rezultatul final la layerul ultim la cel penultim prin înmulțiri succesive. Până ajunge la primele layere, valoarea derivatei devine tot mai eronată.

Motivul pentru care avem nevoie de derivatele în raport cu w și b:

Trainingul presupune w_nou = w – derivată(w), la fel pentru b. Derivata compusă cu ajutorul loss-ului schimbă w și b inițiale pentru ca la următoarea trecere a inputului prin layere, rezultatul să devină cu un pas mai aproape de adevăr. Pașii cu care se schimbă devin tot mai mici. Acesta este gradientul.

Skip connection rezolvă problema de vanishing gradient. În figura 5.2.1, fără de skip connection, valoarea adevărată a lui x ar fi rămas cu 2 layere în urmă, unde era locul său. Se

face o sumă dintre noua valoare și x, cel de acum 2 layere, salvând valoarea lui pentru un backpropagation mai funcțional.

Cum are loc transferul de stil într-o arhitectură CNN cu ajutorul informației din imaginea conținut a fost explicat în preliminarii: NST, noțiuni teoretice.

Un alt exemplu al acestui NST (figura 3.2.3):

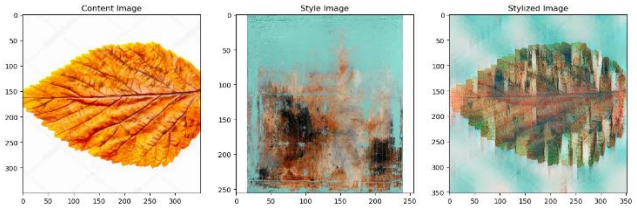

Figura 3.2.3: Style Transfer

III.3 AI-ul pentru comparații pe imagini

Mi-am construit un model destul de simplu, făcând un transfer de task pe parcurs. Am decis să antrenez un **Autoencoder.**

Un Autoencoder folosește o rețea convoluțională pentru a micșora informația (să spunem, dintr-o imagine) până la cele mai mici dimensiuni posibile, și apoi o reconstruiește la loc treptat fără a primi informații despre cum arăta informația înainte de a fi micșorată, și o reconstruiește cu scopul de a pierde sau deteriora cât mai puțin din informația originală.

Un astfel de model este folositor din diverse pretexte:

Un Autoencoder în sine nu ar folosi la nimic (inputul și outputul nu diferă), decât dacă s-ar face un transfer de task. Un Autoencode e format din 2 părți: encoderul (micșorează), decoderul (mărește).

Un autoencoder întreg nu e util, dar una din părți poate fi utilizată la ceva, cum ar fi informația poate fi encodată (pentru a păstra memorie/pentru portabilitate etc), și apoi decodată fără a pierde din ea (observați că și aici autoencoderul funcționează separat).

Pentru cazul meu, nu e util să compar 2 imagini pixel cu pixel, întrucât rezultatul nu va fi valid (părțile similare din imagini se pot afla pe poziții diferite etc, multe dintre forme și detalii pot să nu coincidă exact).

Știm deja, din preliminarii, că o arhitectură CNN poate să rețină informații privind conținutul unei imagini, și o micșorează pe parcurs. Dar există o condiție – weight-urile și bias-urile trebuie să fie antrenate corect. Am antrenat un autoencoder, și m-am condus după un exemplu

existent[24], doar că am layerele un pic diferit. În mijlocul arhitecturii, feature map-ul rezultat din encoder va fi transformat în 1D, pentru a putea compara mai ușor imaginile.

Am lucrat în Google Colab Pro și am utilizat un dataset de câini și pisici[9] pentru Autoencoder.

```
from PIL import Image as img
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras import layers, losses
```

```
class Autoencoder(Model):
  def __init__(self):
    super(Autoencoder, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(1, 256, 256, 4)),
      layers.Conv2D(256, (3, 3), activation='relu', padding='same', stride
s=2, input_shape=(1,256, 256, 4)),
      layers.Conv2D(128, (3, 3), activation='relu', padding='same', stride
s=2),
      layers.Conv2D(64, (3, 3), activation='relu', padding='same', strides
=2),
      layers.Conv2D(16, (3, 3), activation='relu', padding='same', strides
=2),
      layers.Flatten(),
      layers.Dense(512, activation='relu')
      ])

    self.decoder = tf.keras.Sequential([
      layers.Dense(512, activation='sigmoid'),
      layers.Reshape((16, 16, 16)),
      layers.Conv2DTranspose(16, kernel_size=3, strides=2, activation='rel
u', padding='same'),
      layers.Conv2DTranspose(64, kernel_size=3, strides=2, activation='rel
u', padding='same'),
      layers.Conv2DTranspose(128, kernel_size=3, strides=2, activation='re
lu', padding='same'),
      layers.Conv2DTranspose(256, kernel_size=3, strides=2, activation='re
lu', padding='same'),
      layers.Conv2D(4, kernel_size=(3, 3), activation='sigmoid', padding='
same')])

  def call(self, x):
    encoded = self.encoder(x)
```

```
    print(encoded.shape)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = Autoencoder()
```

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

Am creat un checkpoint în Google Drive, pentru a-mi salva weight-urile și bias-urile.

```
checkpoint_path = "/content/drive/My Drive/check/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)
```

Procesarea datelor înainte de train:

```
import os
path = "/content/drive/My Drive/dataset/training_set"
images = os.listdir(path)

lista_im = []
for image in images:
  lista_im.append(os.path.join(path, image))

array_images = []
for image in lista_im:
  array_images.append(np.array(img.open(image)))
array_preprocessed = []

for image in array_images:
  array_preprocessed.append(preprocess_image(tf.convert_to_tensor(image),
256))
```

În cazul în care unele imagini au 4 canale de culoare (format rgba), am adăugat un canal în plus pentru toate imaginile cu 3 canale, cu toți pixelii setați pe 255 (opac).

```
alpha_channels = np.empty((256, 256, 1))
alpha_channels.fill(255)
alpha_torch = tf.convert_to_tensor(alpha_channels)
images_torch = []
```

```
for image in array_preprocessed:
  if image.shape[-1] == 3:
    image = tf.concat([tf.cast(image, dtype=tf.float32), tf.cast(alpha_tor
ch, tf.float32)], axis=2)
  image = tf.expand_dims(image, axis=0)
  images_torch.append(image)
```

```
x_train = images_torch
path_test = "/content/drive/My Drive/dataset/test_set"
```

```
images_test = os.listdir(path_test)
lista_im_test = []
for image in images_test:
  lista_im_test.append(os.path.join(path_test, image))

array_images_test = []
for image in lista_im_test:
  array_images_test.append(np.array(img.open(image)))

array_preprocessed_test = []
for image in array_images_test:
  array_preprocessed_test.append(preprocess_image(tf.convert_to_tensor(ima
ge), 256))


images_torch_test = []
for image in array_preprocessed_test:
  if image.shape[-1] == 3:
    image = tf.concat([tf.cast(image, dtype=tf.float32), tf.cast(alpha_tor
ch, tf.float32)], axis=2)
  image = tf.expand_dims(image, axis=0)
  images_torch_test.append(image)

x_test = images_torch_test
```

Normalizarea datelor (aducerea valorilor între 0 și 1):

```
x_train = tf.stack(x_train, axis=0)
x_test = tf.stack(x_test, axis=0)
x_train = tf.cast(x_train, tf.float32) / 255.
x_test = tf.cast(x_test, tf.float32) / 255.
```

Antrenarea pentru 10 epoci:

```
autoencoder.fit(x_train, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[cp_callback])
```

Construcția unui nou model pentru a putea utiliza valorile celui vechi:

Din pretextul că Autoencoderul meu, primind un input, întoarce același input, cauzat de construcția sa în funcția ”call”. Mai mult de atât, vreau să scap de funcția de activare ReLU pentru ultimul layer din encoder (voi explica de ce).

Modelul din aplicația mea:

```
class ModelEncoder(Model):
    def __init__(self):
        super(ModelEncoder, self).__init__()
        self.encoder = tf.keras.Sequential([
            layers.Input(shape=(1, 256, 256, 4)),
            layers.Conv2D(256, (3, 3), activation='relu', padding='same',
strides=2, input_shape=(1, 256, 256, 4)),
            layers.Conv2D(128, (3, 3), activation='relu', padding='same',
strides=2),
            layers.Conv2D(64, (3, 3), activation='relu', padding='same',
strides=2),
            layers.Conv2D(16, (3, 3), activation='relu', padding='same',
strides=2),
            layers.Conv2D(8, (3, 3), activation='relu', padding='same',
strides=2),
            layers.Flatten(),
            layers.Dense(512)])

        self.decoder = tf.keras.Sequential([
            layers.Dense(512, activation='sigmoid'),
            layers.Reshape((8, 8, 8)),
            layers.Conv2DTranspose(8, kernel_size=3, strides=2,
activation='relu', padding='same'),
            layers.Conv2DTranspose(16, kernel_size=3, strides=2,
activation='relu', padding='same'),
            layers.Conv2DTranspose(64, kernel_size=3, strides=2,
activation='relu', padding='same'),
            layers.Conv2DTranspose(128, kernel_size=3, strides=2,
activation='relu', padding='same'),
            layers.Conv2DTranspose(256, kernel_size=3, strides=2,
activation='relu', padding='same'),
            layers.Conv2D(4, kernel_size=(3, 3), activation='sigmoid',
padding='same')])


    def call(self, x):
        encoded = self.encoder(x)
```

```
        #print(encoded.shape)
        return encoded
```

Vreau să elimin ReLU pentru ultimul layer al encoderului, fiindcă voi compara rezultatul pentru 2 imagini. Voi avea un output de mărime 512 (imaginile micșorate). Outputul va avea atât valori pozitive, cât și negative, care vor fi înlocuite cu 0 în urma ReLU. Voi număra câte valori din 512 pentru cele 2 imagini sunt asemănătoare (fie voi calcula diferența, fie altă metodă). Astfel, se vor număra și valorile de 0, deși e foarte posibil ca valorile negative înlocuite cu 0 să nu fie apropiate. Deci, voi obține o notă mai mare de similaritate, o notă mai eronată.

Calculez similaritatea astfel:

```
def compare(self):
    self.button_compare.setEnabled(False)
    self.your_image.save("my_drawing.png")
    your_drawing = img.open("my_drawing.png")
    your_drawing_np = np.array(your_drawing)
    stylized_image = img.open("stylized_image.png")
    stylized_image_np = np.array(stylized_image)

    your_drawing_rgb =
self.preprocess_image(tf.convert_to_tensor(your_drawing_np), 256)
    stylized_image_rgb =
self.preprocess_image(tf.convert_to_tensor(stylized_image_np), 256)

    your_drawing = your_drawing_rgb.numpy()
    stylized_image = stylized_image_rgb.numpy()

    alpha_channels = np.empty((256, 256, 1))
    alpha_channels.fill(255)

    if your_drawing.shape[-1] == 3:
        your_drawing = np.concatenate((your_drawing, alpha_channels), axis=2)

    if stylized_image.shape[-1] == 3:
        stylized_image = np.concatenate((stylized_image, alpha_channels),
axis=2)

    your_drawing = np.expand_dims(your_drawing, axis=0)
    stylized_image = np.expand_dims(stylized_image, axis=0)

    your_drawing = tf.cast(your_drawing, tf.float32) / 255.
    stylized_image = tf.cast(stylized_image, tf.float32) / 255.

    your_drawing_info = self.compare_model(your_drawing)
    stylized_image_info = self.compare_model(stylized_image)

    sim = self.similarity(your_drawing_info, stylized_image_info)


    sim = sim/512 * 10
    self.label_comparison.setText("Current similarity (from 0 to 10):
```

```
"+str(round(sim, 2)))
    self.button_static.setEnabled(True)
    self.button_abstract.setEnabled(True)



def similarity(self, image1, image2, threshold=5):

    image1 = tf.squeeze(image1)
    image2 = tf.squeeze(image2)

    suma = 0
    for i in range(512):
        suma += 1 if abs(image1[i] - image2[i]) < threshold else 0
    return suma
```

Adun câte cupluri de pixeli sunt asemănătoare dintre cei 512, cu un threshold setat. Am imaginile pentru a primi valori între 0 și 1. Diferența maximă dintre un cuplu de valori poate fi 1. Thresholdul e 0.025 pentru ca să consider pixelii asemănători. La sfârșit fac o medie și înmulțesc cu 10 pentru a avea o valoare între 0 și 10.

III.4 Construcția propriului meu dataset pentru transferul de stil

Am căutat online un dataset de obiecte statice care ar putea fi desenate de către user, care să aibă în același timp o formă ușor de simulat. Un obiect prea detaliat poate deveni frustrant pentru un astfel de experiment. Datasetul meu de obiecte statice include 161 imagini, luate din google Images. Sunt din categoriile: mere, struguri, căni/vase, vaze cu sau fără flori (puține flori), flori în sine (dar imaginile conțin o singură floare care este focusul principal), figurine de porțelan, frunze, sculpturi simple (figura 3.4.1).

Datasetul pentru imagini abstracte, după cum am menționat anterior, a fost cules de pe un website pentru vânzarea picturilor abstracte[22], la moment nu este foarte vast (figura 3.4.2).

Figura 3.4.1: Câteva exemple din datasetul de imagini statice

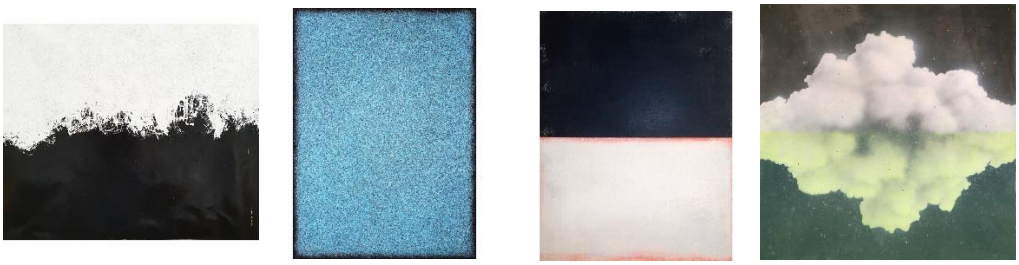

Figura 3.4.2: Câteva exemple din datasetul de imagini abstracte

III.5 Aplicația finală

Aplicația GUI finală este foarte simplistă. Are câteva butoane care inițiază procesul pentru desen, schimbare de imagini (se alege o imagine random la fiecare click, din cele existente în folder), procesul de transfer de stil și cel de comparație între rezultate. Când se dă startul unui proces (cum ar fi transferul de stil), se dezactivează butoanele de schimbare ale imaginilor, întrucât vreau ca aplicația să prezinte imaginile corespunzătoare alături. De asemenea, butonul pentru comparație nu e activ până nu a avut loc transferul și nu s-a desenat o imagine, fie adăugat una din calculator.

Întrucât transferul de stil durează în jur de 100 secunde, atunci când userul începe să deseneze, pornesc și NST-ul.

Am reușit să fac Python să conducă 2 operații în același timp, creând un al doilea thread. La sfârșit, am grijă să îl ”omor” cu nume_thread.join(). Am utilizat biblioteca ”threading”.

```
class Widget(QtWidgets.QWidget):
    def __init__(self, images_static, images_abstract):
        super().__init__()
        self.button_static = QtWidgets.QPushButton("Pick another object image
randomly", self)
        self.button_abstract = QtWidgets.QPushButton("Pick another abstract
image randomly", self)
        self.button_static.resize(100, 50)
        self.button_abstract.resize(100, 50)
        self.size = QtCore.QSize(256, 256)
        self.images_static = images_static
        self.images_abstract = images_abstract

        self.static_image_label = QtWidgets.QLabel()
        self.static_choice = "./static_art/flower10.jpg"
        self.static_image = QtGui.QPixmap("./static_art/flower10.jpg")
        self.static_image = self.static_image.scaled(self.size,
QtCore.Qt.KeepAspectRatio)
        self.static_image_label.setPixmap(self.static_image)

        self.abstract_image_label = QtWidgets.QLabel()
        self.abstract_choice = "./abstract_art/Ivana Olbright_Desert
Roses.jpg"
        self.abstract_image = QtGui.QPixmap("./abstract_art/Ivana
Olbright_Desert Roses.jpg")
```

```
        self.abstract_image = self.abstract_image.scaled(self.size,
QtCore.Qt.KeepAspectRatio)
        self.abstract_image_label.setPixmap(self.abstract_image)


        self.your_image = QtGui.QImage(256, 256, QtGui.QImage.Format_RGB32)
        self.your_image.fill(QtGui.QColor(255, 255, 255))

        self.your_image = QtGui.QPixmap.fromImage(self.your_image)
        self.your_image_label = QtWidgets.QLabel()
        self.your_image_label.setPixmap(self.your_image)

        self.button_draw = QtWidgets.QPushButton("Start drawing...", self)
        self.button_draw.clicked.connect(self.draw)
        self.button_draw.resize(100, 50)

        self.button_upload = QtWidgets.QPushButton("Upload image...", self)
        self.button_upload.clicked.connect(self.upload)
        self.button_upload.resize(100, 50)

        self.stylized_image = QtGui.QImage(256, 256,
QtGui.QImage.Format_RGB32)
        self.stylized_image.fill(QtGui.QColor(255, 255, 255))
        self.stylized_image = QtGui.QPixmap.fromImage(self.stylized_image)
        self.stylized_image_label = QtWidgets.QLabel()
        self.stylized_image_label.setPixmap(self.stylized_image)

        self.button_stylize = QtWidgets.QPushButton("Begin image style
transfer...", self)
        self.button_stylize.clicked.connect(self.style_transfer_aux)
        self.button_stylize.resize(100, 50)

        self.layout1 = QtWidgets.QHBoxLayout()
        self.layout1.addWidget(self.static_image_label)
        self.layout1.addWidget(self.abstract_image_label)
        self.layout1.addWidget(self.your_image_label)
        self.layout1.addWidget(self.stylized_image_label)

        self.layout2 = QtWidgets.QVBoxLayout()
        self.layout2.addWidget(self.button_draw)
        self.layout2.addWidget(self.button_upload)

        self.layout3 = QtWidgets.QHBoxLayout()
        self.layout3.addWidget(self.button_static)
        self.layout3.addWidget(self.button_abstract)
        self.layout3.addLayout(self.layout2)
        self.layout3.addWidget(self.button_stylize)

        self.label_message = QtWidgets.QLabel()
        self.label_message.setStyleSheet(" font-size: 16px;")
        self.label_message.setFixedHeight(50)
        self.layout4 = QtWidgets.QVBoxLayout()
        self.button_compare = QtWidgets.QPushButton("Begin image
comparison...", self)
        self.button_compare.clicked.connect(self.compare)
        self.button_compare.setEnabled(False)
        self.label_comparison = QtWidgets.QLabel("Current similarity (from 0
```

```
to 10): Empty")
        self.label_comparison.setFixedHeight(50)
        self.label_comparison.setStyleSheet(" font-size: 16px;")
        self.layout4.addWidget(self.label_message)
        self.layout4.addWidget(self.label_comparison)
        self.layout4.addWidget(self.button_compare)

        self.drawed = False
        self.transfered = False
        self.compare_model = ModelEncoder()
        checkpoint = tf.train.Checkpoint(self.compare_model)
        checkpoint.restore("./check1/cp.ckpt")
        self.layout = QtWidgets.QVBoxLayout()
        self.layout.addLayout(self.layout1)
        self.layout.addLayout(self.layout3)
        self.layout.addLayout(self.layout4)
        self.setLayout(self.layout)
        self.button_static.clicked.connect(self.changeStatic)
        self.button_abstract.clicked.connect(self.changeAbstract)
        self.setAttribute(QtCore.Qt.WA_StaticContents)
        self.resize(600, 300)


    def changeStatic(self):
        img_choice = choice(self.images_static)
        self.static_choice = img_choice
        image = QtGui.QPixmap(img_choice)
        image = image.scaled(self.size, QtCore.Qt.KeepAspectRatio)
        self.static_image_label.setPixmap(image)
        self.button_stylize.setEnabled(True)
        self.drawed = False
        self.transfered = False
        self.button_compare.setEnabled(False)
        self.label_comparison.setText("Current similarity (from 0 to 10):
Empty")


    def changeAbstract(self):
        img_choice = choice(self.images_abstract)
        self.abstract_choice = img_choice
        image = QtGui.QPixmap(img_choice)
        image = image.scaled(self.size, QtCore.Qt.KeepAspectRatio)
        self.abstract_image_label.setPixmap(image)
        self.button_stylize.setEnabled(True)
        self.drawed = False
        self.transfered = False
        self.button_compare.setEnabled(False)
        self.label_comparison.setText("Current similarity (from 0 to 10):
Empty")


    def compare(self):
        self.button_compare.setEnabled(False)
        self.your_image.save("my_drawing.png")
        your_drawing = img.open("my_drawing.png")
        your_drawing_np = np.array(your_drawing)
        stylized_image = img.open("stylized_image.png")
```

```
        stylized_image_np = np.array(stylized_image)

        your_drawing_rgb =
self.process_image(tf.convert_to_tensor(your_drawing_np), 256)
        stylized_image_rgb =
self.process_image(tf.convert_to_tensor(stylized_image_np), 256)

        your_drawing = your_drawing_rgb.numpy()
        stylized_image = stylized_image_rgb.numpy()

        alpha_channels = np.empty((256, 256, 1))
        alpha_channels.fill(255)

        if your_drawing.shape[-1] == 3:
            your_drawing = np.concatenate((your_drawing, alpha_channels),
axis=2)

        if stylized_image.shape[-1] == 3:
            stylized_image = np.concatenate((stylized_image, alpha_channels),
axis=2)

        your_drawing = np.expand_dims(your_drawing, axis=0)
        stylized_image = np.expand_dims(stylized_image, axis=0)

        your_drawing = tf.cast(your_drawing, tf.float32) / 255.
        stylized_image = tf.cast(stylized_image, tf.float32) / 255.

        your_drawing_info = self.compare_model(your_drawing)
        stylized_image_info = self.compare_model(stylized_image)

        sim = self.similarity(your_drawing_info, stylized_image_info)
        sim = sim/512 * 10
        self.label_comparison.setText("Current similarity (from 0 to 10):
"+str(round(sim, 2)))
        self.button_static.setEnabled(True)
        self.button_abstract.setEnabled(True)



    def similarity(self, image1, image2, threshold=5):

        image1 = tf.squeeze(image1)
        image2 = tf.squeeze(image2)

        suma = 0
        for i in range(512):
            suma += 1 if abs(image1[i] - image2[i]) < threshold else 0
        return suma


    def draw(self):
        self.button_upload.setEnabled(False)
        self.button_static.setEnabled(False)
        self.button_abstract.setEnabled(False)
        self.button_stylize.setEnabled(False)
        self.window = MainWindow(self)
        self.window.show()
```

```
        if not self.transfered:
            x = threading.Thread(target=self.style_transfer, args=("thread",))
            x.start()


    def style_transfer_aux(self):
        x = threading.Thread(target=self.style_transfer, args=("thread",))
        x.start()


    def style_transfer(self, thread=None):
        if self.drawed:
            self.button_static.setEnabled(False)
            self.button_abstract.setEnabled(False)
        self.button_stylize.setEnabled(False)
        self.label_message.setText("Currently style transfering...You may
draw/re-draw or upload an image")
        now = time.time()
        content_image = self.load_image(self.static_choice)
        content_image = self.load_content_image(content_image)

        style_image = self.load_image(self.abstract_choice)
        style_predict_path = tf.keras.utils.get_file('style_predict.tflite',

'https://tfhub.dev/sayakpaul/lite-model/arbitrary-image-stylization-
inceptionv3-dynamic-shapes/int8/predict/1?lite-format=tflite')
        style_transform_path =
tf.keras.utils.get_file('style_transform.tflite',

'https://tfhub.dev/sayakpaul/lite-model/arbitrary-image-stylization-
inceptionv3-dynamic-shapes/int8/transfer/1?lite-format=tflite')
        content_image_size = 256.0

        if len(content_image.shape) > 3:
            content_image = tf.squeeze(content_image)

        if len(style_image.shape) > 3:
            style_image = tf.squeeze(style_image)

        preprocessed_content_image = self.process_image(content_image,
tf.constant([content_image_size]))
        preprocessed_style_image = self.process_image(style_image, 256)
        preprocessed_style_image = tf.expand_dims(preprocessed_style_image, 0)
        style_bottleneck = self.style_predict(preprocessed_style_image,
style_predict_path)

        stylized_image = self.style_transform(style_bottleneck,
tf.expand_dims(preprocessed_content_image, 0),
                                              style_transform_path,
content_image_size)
        if len(stylized_image.shape) > 3:
            stylized_image = tf.squeeze(stylized_image, axis=0)
        image = tf.keras.preprocessing.image.array_to_img(stylized_image)
        image.save("stylized_image.png")

self.stylized_image_label.setPixmap(QtGui.QPixmap("stylized_image.png"))
        now = time.time() - now
```

```
        print(now)
        self.label_comparison.setText("Current similarity (from 0 to 10):
Empty")
        self.label_message.setText("")
        self.transfered = True
        if self.drawed:
            self.button_compare.setEnabled(True)
        if thread is not None:
            threading.currentThread().join()


    def style_predict(self, processed_style_image, style_predict_path):
        interpreter = tf.lite.Interpreter(model_path=style_predict_path)

        interpreter.allocate_tensors()
        input_details = interpreter.get_input_details()
        interpreter.set_tensor(input_details[0]["index"],
processed_style_image)

        interpreter.invoke()
        style_bottleneck =
interpreter.tensor(interpreter.get_output_details()[0]["index"])()
        return style_bottleneck


    def style_transform(self, style_bottleneck, processed_content_image,
style_transform_path, content_image_size):
        interpreter = tf.lite.Interpreter(model_path=style_transform_path)

        input_det = interpreter.get_input_details()
        for index in range(len(input_det)):
            if input_det[index]["name"] == 'content_image':
                index = input_det[index]["index"]
                interpreter.resize_tensor_input(index, [1, content_image_size,
content_image_size, 3])
        interpreter.allocate_tensors()

        for index in range(len(input_det)):
            if input_det[index]["name"] == 'Conv/BiasAdd':
                interpreter.set_tensor(input_det[index]["index"],
style_bottleneck)
            elif input_det[index]["name"] == 'content_image':
                interpreter.set_tensor(input_det[index]["index"],
processed_content_image)
        interpreter.invoke()

        stylized_image =
interpreter.tensor(interpreter.get_output_details()[0]["index"])()
        return stylized_image


    def load_image(self, path_to_img):
        img = tf.io.read_file(path_to_img)
        img = tf.io.decode_image(img, channels=3)
        img = tf.image.convert_image_dtype(img, tf.float32)
        img = img[tf.newaxis, :]
        return img
```

```
    def process_image(self, image, target_dim):
        shape = tf.cast(tf.shape(image)[0:-1], tf.float32)
        short_dim = min(shape)
        scale = target_dim / short_dim
        new_shape = tf.cast(shape * scale, tf.int32)
        image = tf.image.resize(image, new_shape)
        image = tf.image.resize_with_crop_or_pad(image, int(target_dim),
int(target_dim))
        return image


    def load_content_image(self, image_pix):
        if image_pix.shape[-1] == 4:
            image_pixels = Image.fromarray(image_pix)
            image = image_pixels.convert('RGB')
            image = np.array(image)
            image = tf.convert_to_tensor(image)
            image = tf.image.convert_image_dtype(image, tf.float32)
            image = image[tf.newaxis, :]
            return image
        elif image_pix.shape[-1] == 3:
            image = tf.convert_to_tensor(image_pix)
            image = tf.image.convert_image_dtype(image, tf.float32)
            image = image[tf.newaxis, :]
            return image
        elif image_pix.shape[-1] == 1:
            raise TypeError('Grayscale images not supported! Please try with
RGB or RGBA images.')
        print('Exception not thrown')


    def upload(self):
        self.button_static.setEnabled(False)
        self.button_abstract.setEnabled(False)
        fileName, _filter = QtWidgets.QFileDialog.getOpenFileName(self,
self.tr("Open File"), QtCore.QDir.currentPath())
        loadedImage = QtGui.QPixmap(fileName)
        loadedImage = loadedImage.scaled(self.size, QtCore.Qt.KeepAspectRatio)
        self.your_image = loadedImage
        self.your_image_label.setPixmap(loadedImage)
        self.button_upload.setText("Upload another image...")
        self.button_static.setEnabled(True)
        self.button_abstract.setEnabled(True)
        self.drawed = True
        if self.transfered:
            self.button_compare.setEnabled(True)
        self.label_comparison.setText("Current similarity (from 0 to 10):
Empty")

class App(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle(self.tr("Style Transfer Experiment"))

        static_art = os.listdir("./static_art")
```

```
        abstract_art = os.listdir("./abstract_art")
        self.static_images = [os.path.join("./static_art", e) for e in
static_art]
        self.abstract_images = [os.path.join("./abstract_art", e) for e in
abstract_art]
        self.image_widget = Widget(self.static_images, self.abstract_images)
        self.setCentralWidget(self.image_widget)

        self.resize(1280, 720)
```

(Această aplicație nu conține și aplicația pentru desen).

Toate funcțiile care permit transferul de stil au fost preluate din NoteBook-ul oferit împreună cu modelul NST[12]

IV .Concluzii

Lucrând la acest proiect, am învățat cum funcționează un model de transfer de stil între imagini, și cum are loc acest transfer în sine (partea de gram matrix), și cunoașterea acestui lucru este foarte importantă, din moment ce vreau să îmi dezvolt cariera în domeniul IA.

Am reușit să implementez propriile texturi pentru pensule, care comparativ cu pixul simplu oferit de program, arată destul de profesionist. Sunt sigură că nu sunt prima care a obținut acest lucru, însă pe internet nu există implementări în Python pentru astfel de efect, sau orice alt limbaj din care poate fi adaptat pentru QPainter. De acum înainte, pe internet vor exista aceste texturi, fiindcă proiectul meu va fi public.

V. Direcții de îmbunătățire

V.1 Construcția unui propriu NST sau un research mai detaliat pe modelul existent

Prin research mai detaliat, mă refer la faptul că se pot aduce schimbări la modelul existent, apoi reantrena și experimenta pe el.

V.2 Îmbunătățiri în aplicația pentru desen

a) Arată prea simplist
b) Bucket Tool este departe de a fi performant (durează mult)
c) Texturile custom pentru pensule, făcute de mine, nu par să aibă culoarea potrivită dacă imaginea inițială a texturii e de culoare închisă. Pot explica de ce: Pentru a schimba culoarea texturii, imaginea se transformă în grayscale cu valori între 0 și 1, apoi se înmulțește cu valorile RGB a culorii dorite. Cu cât imaginea grayscale e mai aproape de negru, cu atât dă o culoare mai diferită de cea dorită. (figura 5.2.2.1):

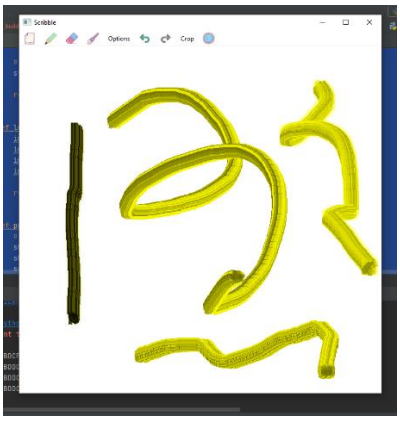

Figura 5.2.2.1: ”Oops”

Observați figura 5.2.2.1. Culoarea dorită era rgb(255,255,0), galben. Textura care era inițial pe negru nu dă un rezultat bun. De ce am păstrat texturile negre? Fiindcă arată bine pe culori închise, dar rămâne un bug. În figură, toate cele 4 linii sunt 4 texturi diferite.

d) Deși QPainter oferă funcții pentru a desena forme, precum elipse, pătrate, cercuri etc, nu am reușit să le implementez la timp. Aceasta ar fi o posibilă dezvoltare ulterioară. Ar ușura munca userului în desen.
e) Am început un tool pentru ”crop”, pe care nu l-am dus la capăt. Vreau ca userul să selecteze, într-un chenar, o porțiune dorită, și să o mute, redimensioneze etc La moment, userul poate doar desena un chenar care dispare la mouseReleaseEvent (figura 5.2.2.2).

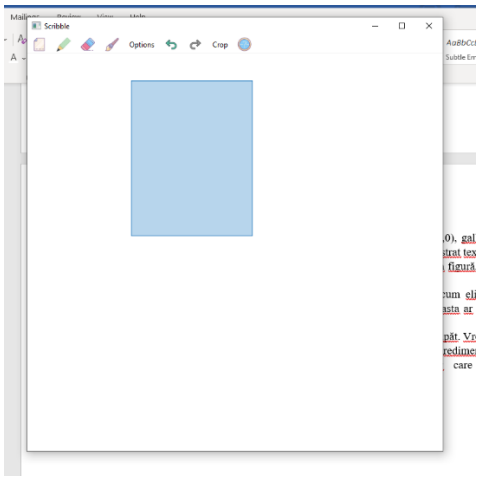

Figura 5.2.2.2: ”Crop?”

V.3 Îmbunătățirea AI-ului de comparație între imagini

La moment, modelul nu dă o similaritate mare între o imagine desenată cu aplicația mea pentru desen și cea stilizată. Motivul notei aprox. 6 din figura 2 (rezumat), este faptul că am setat un threshold destul de mare. Totuși, modelul funcționează pe poze cu obiecte asemănătoare. Voi demonstra în experimentul de mai jos, și voi seta un threshold potrivit de mic:

```
def similarity(image1, image2, threshold = 2):
  suma = 0
  for i in range(512):
    suma += 1 if abs(image1[i] - image2[i]) < threshold else 0
  return suma
```

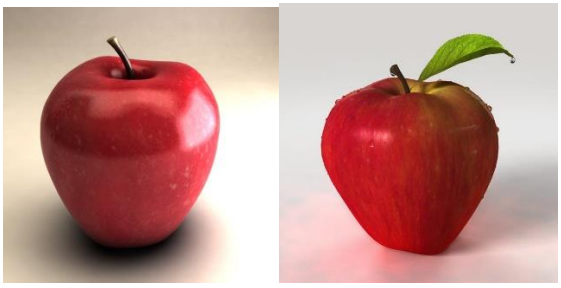

Figura 5.3.1: Primul set de imagini

Figura 5.3.2: Al doilea set de imagini

Pentru primul set de imagini (figura 5.3.1), similaritatea este 4.39 cu un threshold de 2.

Pentru al doilea set de imagini (figura 5.3.2), similaritatea este 1.66 cu același threshold.

În aplicație utilizez un threshold de 5 sau mai sus.

Data ultimelor accesări, pentru toate linkurile, 10-11.06.2021

Bibliografie:

[1] 501 – not implemented, *stackoverflow.com:* https://stackoverflow.com/questions/22860777/how-to-acces-all-pixels-under-a-qpainterpath

[2] A. Chaurasia, *youtube.com: Explained – Neural Sttyle Transfer Research Paper* https://www.youtube.com/watch?v=0tTRA3emrr4

[3] deeplizard, *youtube.com: Max Pooling in Convolutional Neural Networks explained* https://www.youtube.com/watch?v=ZjM_XQa5s6s

[4] Google Brain Team, *github.com: magenta, nza_model.py* https://github.com/magenta/magenta/blob/master/magenta/models/arbitrary_image_stylization/nza_model.py

[5] *H. Nguyen, theartling.com: Green Mantra* https://theartling.com/en/artwork/h-nguyen-green-mantra/

[6] J. Shelns, *arxiv.org: A learned Representation For Artistic Style* https://arxiv.org/abs/1610.07629

[7] K. Mulier, *stackoverflow.com:* https://stackoverflow.com/questions/39396707/how-to-make-icon-in-qmenu-larger-pyqt

[8] L. Gatys, *arxiv.rg: A Neural Algoritm of Artistic Style* https://arxiv.org/abs/1508.06576

[9] lucif3r, *kaggle.com: Cat & Dogs* https://www.kaggle.com/d4rklucif3r/cat-and-dogs

[10] M. Fitzpatrick, *mfitzp.com: Implementing QPainter flood fill in PyQt5/PySide* https://www.mfitzp.com/qna/implementing-qpainter-flood-fill-pyqt5pyside/

[11] M. Fitzpatrick, *mfitzip.com: QPainter and Bitmap Graphics* https://www.mfitzp.com/bitmap-graphics/

[12] S.Paul, *tfhub.dev: Arbitrary Image Stylization* https://tfhub.dev/sayakpaul/lite-model/arbitrary-image-stylization-inceptionv3-dynamic-shapes/dr/transfer/1

[13] somename.py, *stackoverflow.com:* https://stackoverflow.com/questions/67837060/python-how-to-change-color-palette-of-an-image-based-on-a-single-color-picked

[14] TaW, *stackoverflow.com:* https://stackoverflow.com/questions/49290951/creating-different-brush-patterns-in-c-sharp/49298313#49298313

[15] *doc.qt.io: Scribble Example* https://doc.qt.io/qt-5/qtwidgets-widgets-scribble-example.html

[16] *doc.qt.io: QPen* https://doc.qt.io/qtforpython-5/PySide2/QtGui/QPen.html

[17] *doc.qt.io: QBrush* https://doc.qt.io/qtforpython-5/PySide2/QtGui/QBrush.html

[18] *cleanpng.com:* https://www.cleanpng.com/

[19] *icons8.com:* https://icons8.com/

[20] *imagenet.org:* https://www.image-net.org/

[21] *research.google:* https://research.google/teams/brain/

[22] *theartling.com:* https://theartling.com/en/art/abstract/

[23] *tensorflow.org:* https://www.tensorflow.org/api_docs/python/tf/keras/applications/InceptionV3

[24] *tensorfow.org: Intro to Autoencoders* https://www.tensorflow.org/tutorials/generative/autoencoder

[25] *wikipedia.org, Neural Style Transfer:* https://en.wikipedia.org/wiki/Neural_Style_Transfer