Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Московский политехнический университет»

**факультет информационных технологий**

**Кафедра СМАРТ-технологий**

Дисциплина: Нейронные сети в задачах технического зрения и управления

Отчёт по лабораторной работе №3

«Применение свёрточных нейронных сетей»

Работу выполнил\_и

студент\_ка 3 курса

очного отделения

<ФИО>

Научный руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<ФИО>

**Цель работы**

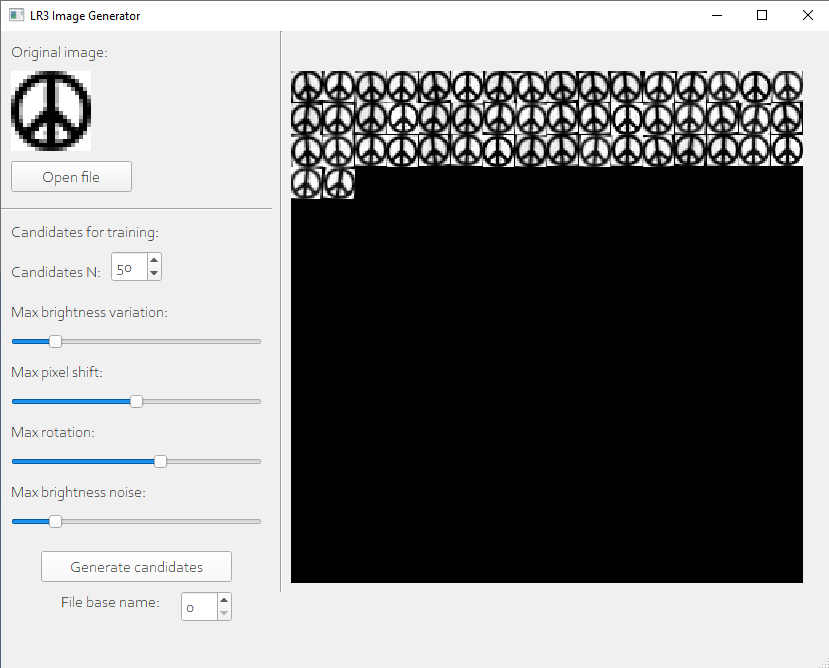
Разработать алгоритм классификации стандартизованных изображений с помощью свёрточной нейронной сети.

**Задачи**

* Подготовить приложение для генерации обучающей выборки из исходного набора изображений;
* Разработать функцию расчета свёрточной карты для выбранного ядра;
* Разработать класс, реализующий функциональность свёрточной сети для классификации изображений;
* Разработать функцию обучения с учителем для нейтронной сети по подготовленной ранее обучающей выборке.

**Ход работы**

В начале было разработано приложение, генерирующее набор изображений, размером 20x20 пикселей для обучения и проверки работы нейросети из любого цветного или ч/б изображения.



В изображения подмешивается яркостный шум, поворот, сдвиг пикселей, изменение яркости. Имеется возможность настройки каждого параметра. Полученные изображения сохраняются в отдельные файлы.

Основное приложение использует данные изображения для обучения и проверки работы нейросети.

Полученные изображения преобразуются в одномерный массив 0-1 и сопоставляясь с «метками»:

-------------------- INPUT DATA --------------------

Shape of loaded\_imagess: (4, 50, 20, 20)

Shape of loaded\_labels: (4, 50, 1)

Arrays:

[[[[0.37254902 0.22745098 0.1372549 ... 0.07058824 0.2627451

...

[1. 1. 1. ... 1. 1.

1. ]]

...

[[0.03921569 0.07058824 0.04313725 ... 0.55686275 0.61176471

1. ]

...

[0.61568627 0.55686275 0.49411765 ... 0.03921569 0.60784314

1. ]]]]

[[[0]

[0]

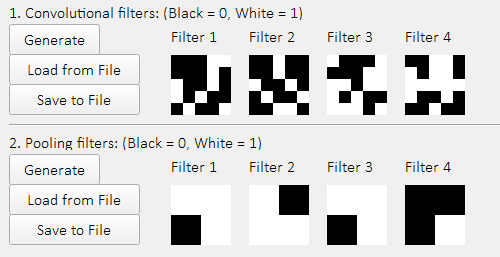
...

[3]

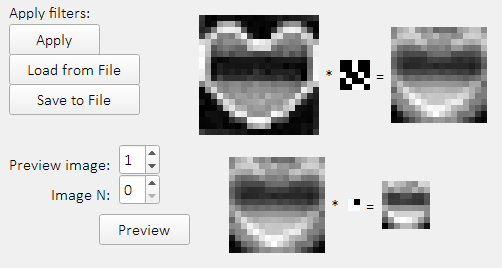
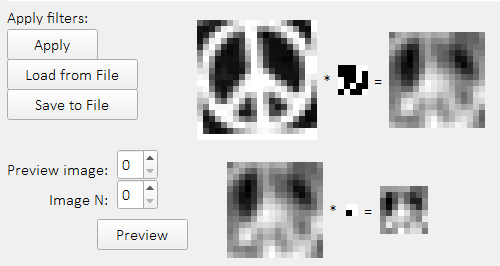
[3]]]

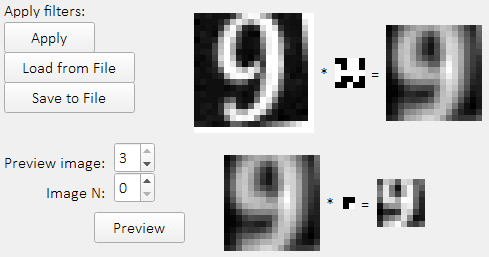
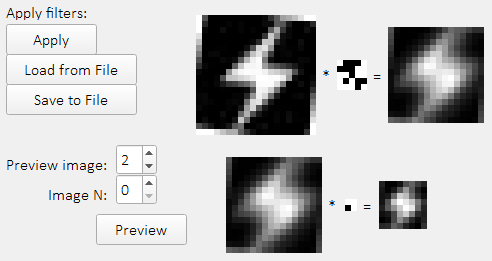
----------------------------------------------------

Далее выполняется генерация ядер свёртки (5x5) и (2x2) случайным образом. Диапазон значений {0; 1}. Также имеется возможность сохранить или загрузить ядра из файла.

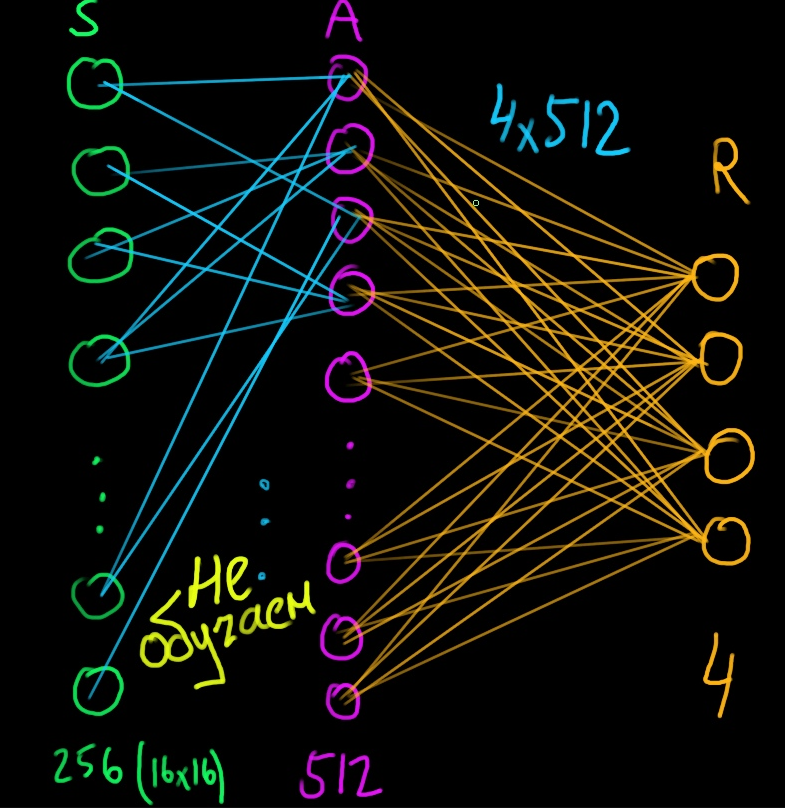


Первые фильтры применяются со страйдом 1, вторые – со страйдом 2. В итоге, на выходе получается изображение 8x8 пикселей. После операции свёртки производится нормализация полученных изображений. Имеется возможность предпросмотра операций свёртки и пуллинга. А также, загрузка и сохранение результатов в файл.





После свёртки, изображение преобразуется в одномерный массив с диапазоном [0; 1] и подаётся на вход MLP-сети, структура которой выглядит следующим образом:



Обучение производиться только оранжевых весов и выглядит следующим образом:

* За первый слой принимается массив входных данных (изображения)
* Далее, вычисляется скалярное произведение первого слоя на первые веса (значения которых {-1; 0; 1}). Результат проходит через функцию активации:

def layer\_0\_activator(weights\_sum):result = [[0] \* weights\_sum[0]] \* weights\_sum  
 for i in range(len(weights\_sum)):  
 sample\_result = [0] \* weights\_sum[0]  
 for k in range(len(weights\_sum[i])):  
  
 threshold = 1.8 if weights\_sum[i][k] >= threshold:  
 sample\_result[k] = weights\_sum[i][k] - 2.2else:  
 sample\_result[k] = 0  
  
 result[i] = sample\_result  
 result = np.array(result)return np.array(result)

* Результат считается вторым слоем. Далее производится скалярное произведение этого [второго] слоя на вторые веса (значения которых [0; 1]). Результат пропускается через сигмоидную функцию активации:

def sigmoid(x):  
 return 1 / (1 + np.exp(-x))

* В итоге, последний (третий) слой содержит 4 элемента.
* Вычисляется ошибка (тот выход, который должен соответствовать метке принимается за 1, остальные три – за 0.).
* На основе ошибки вычисляются корректирующие значения для весов

adjustments\_l2 = output\_l1.T.dot(error\_l2 \* (output\_l2 \* (1 - output\_l2)))

* Эти корректировки прибавляются к весам второго слоя.

В среднем, необходимо 2 итерации чтобы достичь точности в 80-90% и 30-40 итераций, чтобы достичь точности в 95-100%

-------------------- I: 0 --------------------

accuracy: 0.145

-------------------- I: 1 --------------------

accuracy: 0.85

-------------------- I: 2 --------------------

accuracy: 0.925

-------------------- I: 3 --------------------

accuracy: 0.935

...

-------------------- I: 29 --------------------

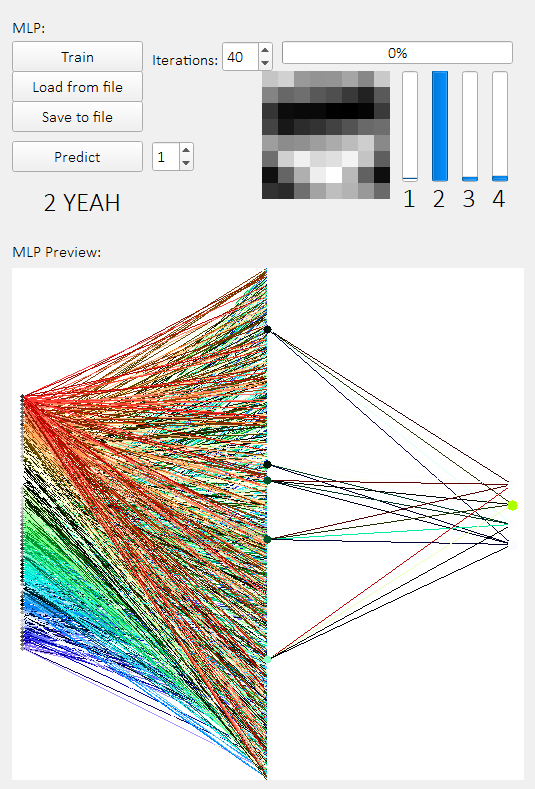
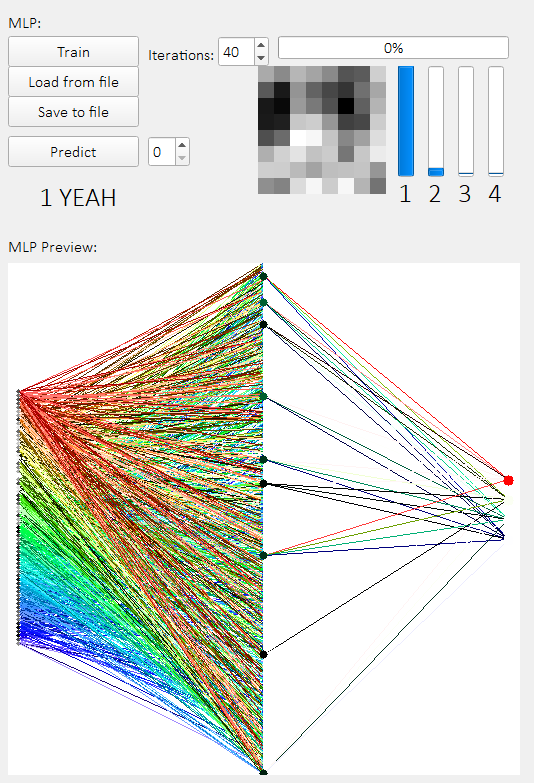
accuracy: 0.98

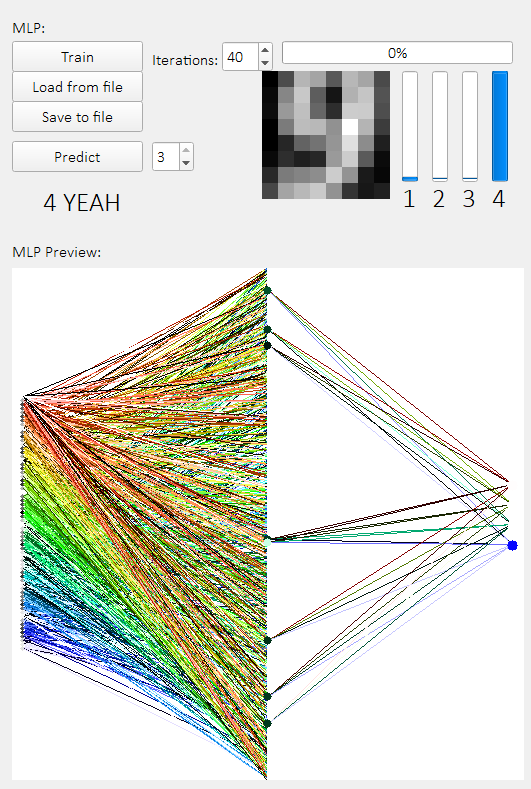
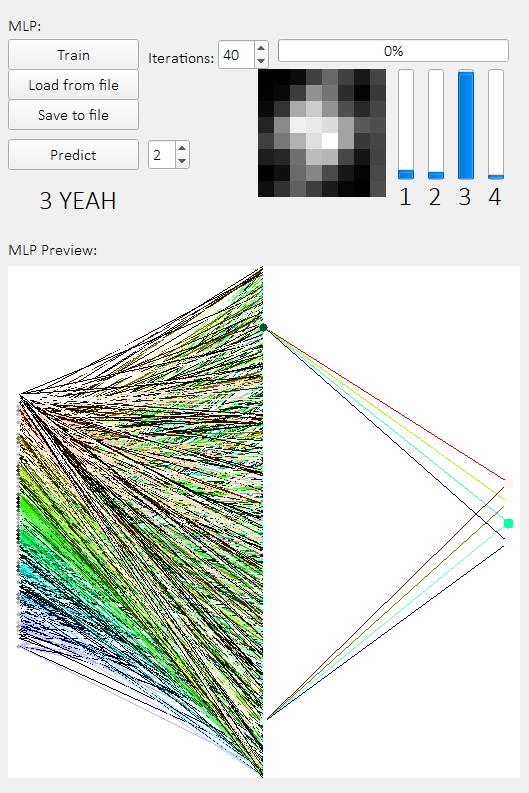
-------------------- I: 30 --------------------

accuracy: 0.98

Обученные веса можно сохранить в файл для дальнейшего использования.

Результат предсказания отображается в виде выходов последнего слоя. А также имеется графический предпросмотр работы MLP в виде слоёв и весов.





**Вывод**

В ходе данной работы была создана свёрточная нейросеть, в начале выполняющая операции свёртка и пуллинга, а также, дальнейшую классификацию с высокой точностью.

**Исходный код генератора изображений**

import os  
import numpy as np  
import cv2  
import sys  
from PyQt5 import QtWidgets  
from PyQt5.QtGui import QPixmap  
import qimage2ndarray  
import random  
from tkinter import filedialog  
import tkinter  
  
root = tkinter.Tk()  
root.withdraw()  
  
import gui\_3\_Collector  
  
  
def valmap(value, istart, istop, ostart, ostop):  
 return ostart + (ostop - ostart) \* ((value - istart) / (istop - istart))  
  
  
class CollectorApp(QtWidgets.QMainWindow, gui\_3\_Collector.Ui\_MainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setupUi(self)  
 self.btn\_open\_file.clicked.connect(self.open\_file)  
 self.btn\_candidates\_generate.clicked.connect(self.generate\_candidates)  
  
 *# self.cvl\_original.setStyleSheet("background-color: lightgreen")  
 # self.cvl\_candidates.setStyleSheet("background-color: lightgreen")* self.original\_image = None  
 self.candidates\_image = None  
  
 def open\_file(self):  
 file\_path = filedialog.askopenfilename()  
 temp\_image = cv2.cvtColor(cv2.imread(file\_path), cv2.COLOR\_BGR2GRAY)  
 self.original\_image = cv2.resize(temp\_image, (20, 20), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_original.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(cv2.resize(self.original\_image, (80, 80), interpolation=cv2.INTER\_NEAREST),  
 cv2.COLOR\_GRAY2RGB))))  
  
 print(file\_path)  
  
 def generate\_candidates(self):  
 self.candidates\_image = np.zeros(shape=[320, 320], dtype=np.uint8) *# [16 X 16] images or [16\*20 X 16\*20] pixels* generated\_images = []  
  
 *# Generation of modified images* for n in range(self.spin\_cindidates\_n.value()):  
 temp\_image = self.original\_image.copy().astype(**'float32'**)  
  
 *# Brightness noise* if self.slider\_brightness.value():  
 temp\_image += random.randrange(-self.slider\_brightness.value(), self.slider\_brightness.value())  
  
 temp\_image = np.clip(temp\_image, 0, 255)  
 temp\_image = temp\_image.astype(**'uint8'**)  
  
 *# Pixel noise* if self.slider\_noise.value():  
 for i in range(20):  
 for j in range(20):  
 if random.randrange(0, 2) and temp\_image[i][j] > 0:  
 brightness\_pixel = temp\_image[i][j] + random.randrange(-self.slider\_noise.value(),  
 self.slider\_noise.value())  
 if brightness\_pixel > 255: brightness\_pixel = 255  
 if brightness\_pixel < 0: brightness\_pixel = 0  
 temp\_image[i][j] = brightness\_pixel  
  
 *# Rotation* if self.slider\_rotation.value() > 0:  
 num\_rows, num\_cols = temp\_image.shape[:2]  
 rotation\_matrix = cv2.getRotationMatrix2D(  
 (num\_cols / 2, num\_rows / 2),  
 random.randrange(-self.slider\_rotation.value(), self.slider\_rotation.value()), 1)  
 temp\_image = cv2.warpAffine(temp\_image, rotation\_matrix, (num\_cols, num\_rows))  
  
 *# Pixel shift* if self.slider\_shift.value() > 0:  
 shift\_size\_x = random.randrange(-self.slider\_shift.value(), self.slider\_shift.value())  
 shift\_size\_y = random.randrange(-self.slider\_shift.value(), self.slider\_shift.value())  
 rows, cols = temp\_image.shape  
 M = np.float32([[1, 0, shift\_size\_x], [0, 1, shift\_size\_y]])  
 temp\_image = cv2.warpAffine(temp\_image, M, (cols, rows))  
  
 generated\_images.append(temp\_image)  
  
 if not os.path.exists(**'LR3\_data'**):  
 os.makedirs(**'LR3\_data'**)  
  
 image\_counter = 0  
 for y in range(16):  
 for x in range(16):  
 temp\_image = generated\_images[image\_counter]  
  
 *# Save to file* cv2.imwrite(**'LR3\_data/'** + str(self.spin\_cindidates\_file.value())  
 + **'\_'** + str(image\_counter) + **'.bmp'**, temp\_image)  
  
 self.candidates\_image[y \* 20: y \* 20 + 20, x \* 20: x \* 20 + 20] = temp\_image *# y:y + h, x:x + w* image\_counter += 1  
 if image\_counter == len(generated\_images):  
 break  
 if image\_counter == len(generated\_images):  
 break  
  
 self.cvl\_candidates.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(cv2.resize(self.candidates\_image, (512, 512),  
 interpolation=cv2.INTER\_NEAREST), cv2.COLOR\_GRAY2RGB))))  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 app = QtWidgets.QApplication(sys.argv)  
 app.setStyle(**"fusion"**)  
 window = CollectorApp()  
 window.show()  
 app.exec\_()

**Исходный код основной программы**

import threading  
  
import numpy as np  
import cv2  
import sys  
from PyQt5 import QtWidgets  
import random  
from PyQt5.QtGui import QPixmap  
import qimage2ndarray  
import pickle  
  
import gui\_3\_CNN  
  
  
def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
  
def dot\_0\_layer(input\_layer, synaptic\_weights):  
 return layer\_0\_activator(np.dot(input\_layer, synaptic\_weights.T))  
  
  
def valmap(value, istart, istop, ostart, ostop):  
 return ostart + (ostop - ostart) \* ((value - istart) / (istop - istart))  
  
  
def layer\_0\_activator(weights\_sum):  
 *# return max(0, weights\_sum)* result = [[0] \* weights\_sum[0]] \* weights\_sum  
 for i in range(len(weights\_sum)):  
 sample\_result = [0] \* weights\_sum[0]  
 for k in range(len(weights\_sum[i])):  
  
 threshold = 1.8 *# 1.79* if weights\_sum[i][k] >= threshold:  
 sample\_result[k] = weights\_sum[i][k] - 2.2 *# 0.79* else:  
 sample\_result[k] = 0  
  
 result[i] = sample\_result  
 result = np.array(result)  
 *# exit(0)* return np.array(result)  
  
  
class CNNApp(QtWidgets.QMainWindow, gui\_3\_CNN.Ui\_MainWindow):  
 def \_\_init\_\_(self):  
 self.DEBUG = True  
  
 super().\_\_init\_\_()  
 self.setupUi(self)  
 self.btn\_load\_images.clicked.connect(self.load\_images)  
 self.btn\_filter\_1\_generate.clicked.connect(self.filter\_1\_generate)  
 self.btn\_filter\_1\_load.clicked.connect(self.filter\_1\_load)  
 self.btn\_filter\_1\_save.clicked.connect(self.filter\_1\_save)  
 self.btn\_filter\_2\_generate.clicked.connect(self.filter\_2\_generate)  
 self.btn\_filter\_2\_load.clicked.connect(self.filter\_2\_load)  
 self.btn\_filter\_2\_save.clicked.connect(self.filter\_2\_save)  
 self.btn\_filters\_load.clicked.connect(self.filters\_load)  
 self.btn\_filters\_save.clicked.connect(self.filters\_save)  
  
 self.btn\_apply\_filters.clicked.connect(self.apply\_filters)  
 self.btn\_preview\_filters.clicked.connect(self.preview\_filters)  
  
 self.btn\_start\_training.clicked.connect(self.start\_training)  
 self.btn\_predict.clicked.connect(self.predict\_test\_image)  
 self.btn\_save\_to\_file.clicked.connect(self.save\_model\_to\_file)  
 self.btn\_load\_from\_file.clicked.connect(self.load\_model\_from\_file)  
 self.test\_values = False  
  
 self.synaptic\_weights\_0 = np.array([])  
 self.synaptic\_weights\_1 = np.array([])  
  
 self.loaded\_images = []  
 self.loaded\_labels = []  
 self.cnn\_filters\_1 = []  
 self.cnn\_convoluted\_1 = []  
 self.cnn\_filters\_2 = []  
 self.cnn\_convoluted\_2 = []  
  
 def load\_images(self):  
 self.loaded\_images = []  
 self.loaded\_labels = []  
  
 for i in range(4):  
 loaded\_images\_temp = []  
 loaded\_labels\_temp = []  
 for k in range(self.spin\_images\_n.value()):  
 loaded\_images\_temp.append(1.0 - cv2.cvtColor(cv2.imread(  
 str(self.line\_folder.text()) + str(i) + **'\_'** + str(k) + **'.bmp'**), cv2.COLOR\_BGR2GRAY) / 255.0)  
 loaded\_labels\_temp.append([i])  
 self.loaded\_labels.append(loaded\_labels\_temp)  
 self.loaded\_images.append(loaded\_images\_temp)  
  
 self.loaded\_images = np.array(self.loaded\_images)  
 self.loaded\_labels = np.array(self.loaded\_labels)  
  
 if self.DEBUG:  
 print(**'-------------------- INPUT DATA --------------------'**)  
 print(**'Shape of loaded\_imagess: '** + str(self.loaded\_images.shape))  
 print(**'Shape of loaded\_labels: '** + str(self.loaded\_labels.shape))  
 print(**'Arrays:'**)  
 print(self.loaded\_images)  
 print()  
 print(self.loaded\_labels)  
 print(**'----------------------------------------------------'**)  
  
 def filter\_1\_generate(self):  
 self.cnn\_filters\_1 = []  
 for i in range(4):  
 x\_temp = []  
 for x in range(5):  
 y\_temp = []  
 for y in range(5):  
 y\_temp.append(random.randrange(0, 2)) *# {0; 1}* x\_temp.append(y\_temp)  
 self.cnn\_filters\_1.append(x\_temp)  
 self.cnn\_filters\_1 = np.array(self.cnn\_filters\_1)  
 self.filter\_1\_show()  
  
 def filter\_1\_load(self):  
 with open(self.line\_folder.text() + **'cnn\_filters\_1.dat'**, **'rb'**) as filehandle:  
 self.cnn\_filters\_1 = np.array(pickle.load(filehandle))  
 self.filter\_1\_show()  
  
 def filter\_1\_save(self):  
 with open(self.line\_folder.text() + **'cnn\_filters\_1.dat'**, **'wb'**) as filehandle:  
 pickle.dump(self.cnn\_filters\_1, filehandle)  
  
 def filter\_1\_show(self):  
 *# filter 1* image\_temp = (self.cnn\_filters\_1[0]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_1\_1.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 2* image\_temp = (self.cnn\_filters\_1[1]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_1\_2.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 3* image\_temp = (self.cnn\_filters\_1[2]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_1\_3.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 4* image\_temp = (self.cnn\_filters\_1[3]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_1\_4.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 def filter\_2\_generate(self):  
 self.cnn\_filters\_2 = []  
 for i in range(4):  
 x\_temp = []  
 for x in range(2):  
 y\_temp = []  
 for y in range(2):  
 y\_temp.append(random.randrange(0, 2)) *# {0; 1}* x\_temp.append(y\_temp)  
 self.cnn\_filters\_2.append(x\_temp)  
 self.cnn\_filters\_2 = np.array(self.cnn\_filters\_2)  
 self.filter\_2\_show()  
  
 def filter\_2\_load(self):  
 with open(self.line\_folder.text() + **'cnn\_filters\_2.dat'**, **'rb'**) as filehandle:  
 self.cnn\_filters\_2 = np.array(pickle.load(filehandle))  
 self.filter\_2\_show()  
  
 def filter\_2\_save(self):  
 with open(self.line\_folder.text() + **'cnn\_filters\_2.dat'**, **'wb'**) as filehandle:  
 pickle.dump(self.cnn\_filters\_2, filehandle)  
  
 def filter\_2\_show(self):  
 *# filter 1* image\_temp = (self.cnn\_filters\_2[0]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_2\_1.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 2* image\_temp = (self.cnn\_filters\_2[1]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_2\_2.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 3* image\_temp = (self.cnn\_filters\_2[2]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_2\_3.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 *# filter 4* image\_temp = (self.cnn\_filters\_2[3]) \* 255.0  
 image\_temp = cv2.resize(image\_temp.astype(int), (60, 60), interpolation=cv2.INTER\_NEAREST)  
 self.cvl\_filter\_2\_4.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(image\_temp)))  
  
 def apply\_filters(self):  
 *# CONVOLUTION 1* self.cnn\_convoluted\_1 = []  
 for i in range(4):  
 image\_array = []  
 for k in range(self.spin\_images\_n.value()):  
 single\_image = []  
 for x in range(16):  
 convoluted\_row = []  
 for y in range(16):  
 convoluted\_pixel = 0  
 for x\_k in range(5):  
 for y\_k in range(5):  
 convoluted\_pixel += self.loaded\_images[i][k][x + x\_k][y + y\_k] \  
 \* self.cnn\_filters\_1[i][x\_k][y\_k]  
 convoluted\_row.append(convoluted\_pixel)  
 single\_image.append(convoluted\_row)  
 image\_array.append(single\_image)  
 self.cnn\_convoluted\_1.append(image\_array)  
  
 self.cnn\_convoluted\_1 = np.array(self.cnn\_convoluted\_1)  
 self.cnn\_convoluted\_1 /= 25.0  
  
 *# Normalization* for i in range(4):  
 for k in range(self.spin\_images\_n.value()):  
 self.cnn\_convoluted\_1[i][k] = self.cnn\_convoluted\_1[i][k] - self.cnn\_convoluted\_1[i][k].min()  
 if self.cnn\_convoluted\_1[i][k].max() > 0:  
 self.cnn\_convoluted\_1[i][k] = self.cnn\_convoluted\_1[i][k] \*\  
 (1.0 / self.cnn\_convoluted\_1[i][k].max())  
  
 *# CONVOLUTION 2* self.cnn\_convoluted\_2 = []  
 for i in range(4):  
 image\_array = []  
 for k in range(self.spin\_images\_n.value()):  
 single\_image = []  
 x = 0  
 while x <= 14:  
 convoluted\_row = []  
 y = 0  
 while y <= 14:  
 convoluted\_pixel = 0  
 for x\_k in range(2):  
 for y\_k in range(2):  
 convoluted\_pixel += self.cnn\_convoluted\_1[i][k][x + x\_k][y + y\_k] \  
 \* self.cnn\_filters\_2[i][x\_k][y\_k]  
 convoluted\_row.append(convoluted\_pixel)  
 y += 2  
 single\_image.append(convoluted\_row)  
 x += 2  
 image\_array.append(single\_image)  
 self.cnn\_convoluted\_2.append(image\_array)  
  
 self.cnn\_convoluted\_2 = np.array(self.cnn\_convoluted\_2)  
 self.cnn\_convoluted\_2 /= 4.0  
  
 *# Normalization* for i in range(4):  
 for k in range(self.spin\_images\_n.value()):  
 self.cnn\_convoluted\_2[i][k] = self.cnn\_convoluted\_2[i][k] - self.cnn\_convoluted\_2[i][k].min()  
 if self.cnn\_convoluted\_2[i][k].max() > 0:  
 self.cnn\_convoluted\_2[i][k] = self.cnn\_convoluted\_2[i][k] \* \  
 (1.0 / self.cnn\_convoluted\_2[i][k].max())  
 self.preview\_filters()  
  
 def filters\_load(self):  
 *# noinspection PyBroadException* try:  
 print(**'Loading filters and convolutions from file...'**)  
 self.filter\_1\_load()  
 self.filter\_2\_load()  
 with open(self.line\_folder.text() + **'cnn\_convolution\_1.dat'**, **'rb'**) as filehandle:  
 self.cnn\_convoluted\_1 = np.array(pickle.load(filehandle))  
 with open(self.line\_folder.text() + **'cnn\_convolution\_2.dat'**, **'rb'**) as filehandle:  
 self.cnn\_convoluted\_2 = np.array(pickle.load(filehandle))  
 self.preview\_filters()  
 print(**'Done.'**)  
 except:  
 print(sys.exc\_info())  
  
 def filters\_save(self):  
 print(**'Saving filters and convolutions to file...'**)  
 self.filter\_1\_save()  
 self.filter\_2\_save()  
 with open(self.line\_folder.text() + **'cnn\_convolution\_1.dat'**, **'wb'**) as filehandle:  
 pickle.dump(self.cnn\_convoluted\_1, filehandle)  
 with open(self.line\_folder.text() + **'cnn\_convolution\_2.dat'**, **'wb'**) as filehandle:  
 pickle.dump(self.cnn\_convoluted\_2, filehandle)  
 print(**'Done.'**)  
  
 def preview\_filters(self):  
 src\_data = self.loaded\_images[self.spin\_preview\_from.value()][self.spin\_preview\_n.value()]  
 filter\_data\_1 = self.cnn\_filters\_1[self.spin\_preview\_from.value()]  
 result\_data\_1 = self.cnn\_convoluted\_1[self.spin\_preview\_from.value()][self.spin\_preview\_n.value()]  
 filter\_data\_2 = self.cnn\_filters\_2[self.spin\_preview\_from.value()]  
 result\_data\_2 = self.cnn\_convoluted\_2[self.spin\_preview\_from.value()][self.spin\_preview\_n.value()]  
  
 src\_image = (src\_data \* 255.0).astype(int)  
 filter\_image\_1 = (filter\_data\_1 \* 255.0).astype(int)  
 result\_image\_1 = (result\_data\_1 \* 255.0).astype(int)  
 filter\_image\_2 = (filter\_data\_2 \* 255.0).astype(int)  
 result\_image\_2 = (result\_data\_2 \* 255.0).astype(int)  
  
 self.ocl\_preview\_src.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(src\_image, (120, 120), interpolation=cv2.INTER\_NEAREST))))  
 self.ocl\_preview\_filter.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(filter\_image\_1, (30, 30), interpolation=cv2.INTER\_NEAREST))))  
 self.ocl\_preview\_result.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(result\_image\_1, (96, 96), interpolation=cv2.INTER\_NEAREST))))  
 self.ocl\_preview\_src\_2.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(result\_image\_1, (96, 96), interpolation=cv2.INTER\_NEAREST))))  
 self.ocl\_preview\_filter\_2.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(filter\_image\_2, (12, 12), interpolation=cv2.INTER\_NEAREST))))  
 self.ocl\_preview\_result\_2.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(result\_image\_2, (48, 48), interpolation=cv2.INTER\_NEAREST))))  
  
 def predict\_test\_image(self):  
 *# noinspection PyBroadException* try:  
 *# self.load\_images()  
 # self.filters\_load()  
 # self.load\_model\_from\_file()* random\_index = random.randrange(0, self.spin\_images\_n.value())  
 temp\_data = self.cnn\_convoluted\_2[self.spin\_test\_array\_id.value()][random\_index]  
  
 temp\_image = (temp\_data \* 255.0).astype(int)  
 self.ocl\_test\_image.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(temp\_image, (128, 128), interpolation=cv2.INTER\_NEAREST))))  
  
 test\_array = np.array([temp\_data.flatten()])  
  
 test\_labels = np.array([[self.spin\_test\_array\_id.value()]])  
 output\_l0 = test\_array  
 output\_l1 = dot\_0\_layer(output\_l0, self.synaptic\_weights\_0)  
 output\_l2 = sigmoid(np.dot(output\_l1, self.synaptic\_weights\_1.T))  
 if np.argmax(output\_l2[0]) == test\_labels[0][0]:  
 self.label\_predicted.setText(str(int(np.argmax(output\_l2[0])) + 1) + **' YEAH'**)  
 else:  
 self.label\_predicted.setText(str(int(np.argmax(output\_l2[0])) + 1) + **' NOPE'**)  
 self.progressBar\_2.setValue(output\_l2[0][0] \* 100)  
 self.progressBar\_3.setValue(output\_l2[0][1] \* 100)  
 self.progressBar\_4.setValue(output\_l2[0][2] \* 100)  
 self.progressBar\_5.setValue(output\_l2[0][3] \* 100)  
  
 *# Preview CNN* cnn\_preview\_image = 255 \* np.ones((512, 512, 3), dtype=np.uint8)  
 y = int((512 - 256) / 2)  
 for i in range(64):  
 color = int((1 - test\_array[0][i]) \* 255)  
 cv2.circle(cnn\_preview\_image, (10, y + 4 \* i), 2, (color, color, color), -1)  
  
 y = int((512 - 256) / 2)  
 for i in range(512):  
 for k in range(64):  
 calculated = self.synaptic\_weights\_0[i][k] \* output\_l0[0][k]  
 below\_zero = False if calculated > 0 else True  
 saturation = calculated \* 255 if not below\_zero else 255  
 value = calculated \* -255 if below\_zero else 255  
  
 color\_hsv = np.uint8([[[k \* 2, saturation, value]]])  
 color\_rgb = cv2.cvtColor(color\_hsv, cv2.COLOR\_HSV2RGB)[0][0]  
 if calculated != 0:  
 cv2.line(cnn\_preview\_image, (254, i), (11, y + 4 \* k),  
 (int(color\_rgb[0]), int(color\_rgb[1]), int(color\_rgb[2])))  
  
 y = int((512 - 80) / 2)  
 for i in range(512):  
 for k in range(4):  
 calculated = self.synaptic\_weights\_1[k][i] \* output\_l1[0][i]  
 below\_zero = False if calculated > 0 else True  
 saturation = calculated \* 127 if not below\_zero else 255  
 value = calculated \* -127 if below\_zero else 255  
  
 color\_hsv = np.uint8([[[k \* 40, saturation, value]]])  
 color\_rgb = cv2.cvtColor(color\_hsv, cv2.COLOR\_HSV2RGB)[0][0]  
 if calculated != 0:  
 cv2.line(cnn\_preview\_image, (256, i), (500, y + 20 \* k),  
 (int(color\_rgb[0]), int(color\_rgb[1]), int(color\_rgb[2])))  
  
 y = int((512 - 80) / 2)  
 for i in range(4):  
 saturation = output\_l2[0][i] \* 255  
 color\_hsv = np.uint8([[[i \* 40, saturation, 255]]])  
 color\_rgb = cv2.cvtColor(color\_hsv, cv2.COLOR\_HSV2RGB)[0][0]  
  
 cv2.circle(cnn\_preview\_image, (500, y + 20 \* i + 1), 5,  
 (int(color\_rgb[0]), int(color\_rgb[1]), int(color\_rgb[2])), -1)  
  
 for i in range(512):  
 calculated = output\_l1[0][i]  
 below\_zero = False if calculated > 0 else True  
 saturation = calculated \* 255 if not below\_zero else 255  
 value = calculated \* -255 if below\_zero else 255  
 color\_hsv = np.uint8([[[255, saturation, value]]])  
 color\_rgb = cv2.cvtColor(color\_hsv, cv2.COLOR\_HSV2RGB)[0][0]  
 if calculated != 0:  
 cv2.circle(cnn\_preview\_image, (255, i), 4,  
 (int(color\_rgb[0]), int(color\_rgb[1]), int(color\_rgb[2])), -1)  
  
 self.ocl\_preview\_cnn.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.resize(cnn\_preview\_image, (512, 512), interpolation=cv2.INTER\_NEAREST))))  
  
 except:  
 print(sys.exc\_info())  
  
 def save\_model\_to\_file(self):  
 compressed\_data = [self.synaptic\_weights\_0, self.synaptic\_weights\_1]  
 with open(self.line\_folder.text() + **'model.dat'**, **'wb'**) as filehandle:  
 pickle.dump(compressed\_data, filehandle)  
  
 def load\_model\_from\_file(self):  
 with open(self.line\_folder.text() + **'model.dat'**, **'rb'**) as filehandle:  
 compressed\_data = pickle.load(filehandle)  
 self.synaptic\_weights\_0 = np.array(compressed\_data[0])  
 self.synaptic\_weights\_1 = np.array(compressed\_data[1])  
  
 print(**'-------------------- WEIGHTS --------------------'**)  
 print(**'Shape of synaptic\_weights\_0: '** + str(self.synaptic\_weights\_0.shape))  
 print(**'Shape of synaptic\_weights\_1: '** + str(self.synaptic\_weights\_1.shape))  
 print(**'Arrays:'**)  
 print(self.synaptic\_weights\_0)  
 print()  
 print(self.synaptic\_weights\_1)  
 print(**'-------------------------------------------------'**)  
  
 def start\_training(self):  
 if len(self.synaptic\_weights\_0) == 0 or len(self.synaptic\_weights\_1) == 0:  
 *# Synaptic weights arrays* self.synaptic\_weights\_0 = []  
 for i in range(512): *# 4096* string\_array = [int(random.randrange(-1, 2)) for \_ in range(3)] + [0 for \_ in range(61)]  
 random.shuffle(string\_array)  
 self.synaptic\_weights\_0.append(string\_array)  
 self.synaptic\_weights\_0 = np.array(self.synaptic\_weights\_0)  
 self.synaptic\_weights\_1 = np.array(2 \* np.random.random((4, 512)) - 1) *# 4096* if self.DEBUG:  
 print(**'-------------------- WEIGHTS --------------------'**)  
 print(**'Shape of synaptic\_weights\_0: '** + str(self.synaptic\_weights\_0.shape))  
 print(**'Shape of synaptic\_weights\_1: '** + str(self.synaptic\_weights\_1.shape))  
 print(**'Arrays:'**)  
 print(self.synaptic\_weights\_0)  
 print()  
 print(self.synaptic\_weights\_1)  
 print(**'-------------------------------------------------'**)  
  
 thread = threading.Thread(target=self.training)  
 thread.start()  
  
 def training(self):  
 self.cnn\_convoluted\_2 = np.array(self.cnn\_convoluted\_2)  
 self.loaded\_labels = np.array(self.loaded\_labels)  
 train\_data = []  
 train\_labels = []  
 for i in range(4):  
 for k in range(self.spin\_images\_n.value()):  
 train\_data.append(self.cnn\_convoluted\_2[i][k].flatten())  
 for i in range(4):  
 for k in range(self.spin\_images\_n.value()):  
 train\_labels.append(self.loaded\_labels[i][k][0])  
 train\_data = np.array(train\_data)  
 train\_labels = np.array([train\_labels])  
  
 if self.DEBUG:  
 print(**'-------------------- TRAIN DATA --------------------'**)  
 print(**'Shape of train\_data: '** + str(train\_data.shape))  
 print(**'Shape of train\_labels: '** + str(train\_labels.shape))  
 print(**'Arrays:'**)  
 print(train\_data)  
 print()  
 print(train\_labels)  
 print(**'----------------------------------------------------'**)  
  
 *# noinspection PyBroadException* try:  
 i = 0  
 while i < int(self.spin\_iterations.value()):  
 output\_l0 = train\_data  
 output\_l1 = dot\_0\_layer(output\_l0, self.synaptic\_weights\_0)  
 output\_l2 = sigmoid(np.dot(output\_l1, self.synaptic\_weights\_1.T))  
  
 *# Layer 2 error calculations* error\_l2 = []  
 for k in range(len(output\_l2)):  
 a = []  
 for m in range(4):  
 if m == train\_labels[0][k]:  
 a.append(1 - output\_l2[k][m])  
 else:  
 a.append(0 - output\_l2[k][m])  
 error\_l2.append(a)  
 error\_l2 = np.array(error\_l2)  
  
 adjustments\_l2 = output\_l1.T.dot(error\_l2 \* (output\_l2 \* (1 - output\_l2)))  
 self.synaptic\_weights\_1 += adjustments\_l2.T  
  
 *# Accuracy calculations* predicted = []  
 accuracy = 0  
 for k in range(len(output\_l2)):  
 predicted.append(np.argmax(output\_l2[k]))  
 if np.argmax(output\_l2[k]) == train\_labels[0][k]:  
 accuracy += 1  
 accuracy /= len(output\_l2)  
 predicted = np.array(predicted)  
  
 if i % 1 == 0:  
 print(**'-------------------- I: '** + str(i) + **' --------------------'**)  
 *# print('output\_l2: ' + str(output\_l2))  
 # print('error\_l2: ' + str(error\_l2))* print(**'predicted: '** + str(predicted))  
 print(**'accuracy: '** + str(accuracy))  
 *# print('adjustments\_l2: ' + str(adjustments\_l2))  
 # print('----------------------------------------------')* i += 1  
 self.progressBar.setValue(valmap(i, 0, self.spin\_iterations.value(), 0, 100))  
  
 self.progressBar.setValue(0)  
  
 except:  
 print(sys.exc\_info())  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 app = QtWidgets.QApplication(sys.argv)  
 app.setStyle(**"Fusion"**)  
  
 window = CNNApp()  
 window.show()  
 app.exec\_()