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 №2

«Применение перцептронов для распознавания изображений»

Работу выполнил\_и

студент\_ка 3 курса

очного отделения

<ФИО>

Научный руководитель

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

<ФИО>

**Цель работы**

Разработать алгоритм обработки зон интереса для классификации

изображения на основе нейронной сети.

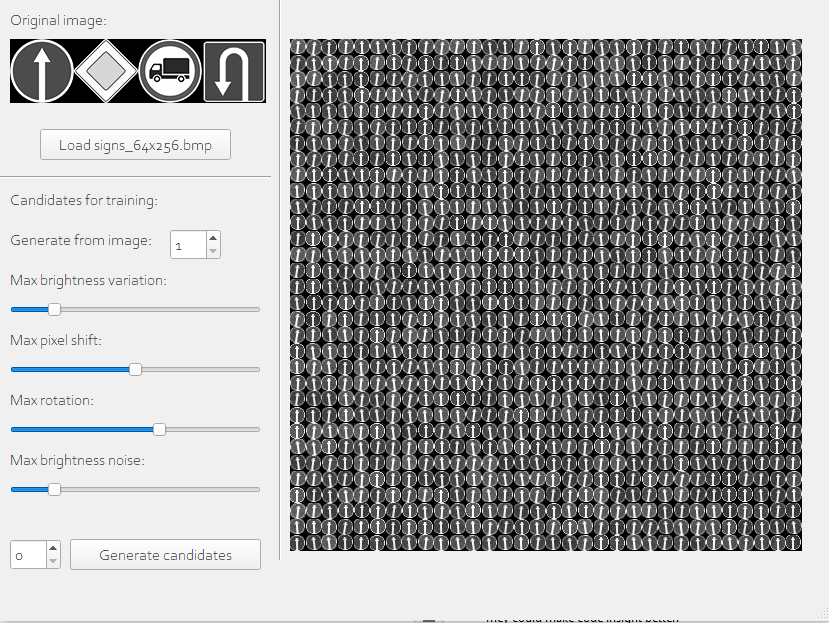
**Задачи**

* Подготовить приложение для генерации обучающей выборки из исходного набора изображений;
* Разработать класс, реализующий функциональность нейронной сети на основе перцептрона с одним скрытым слоем;
* Реализовать функцию классификации произвольного изображения;
* Разработать функцию обучения с учителем для нейтронной сети по подготовленной ранее обучающей выборке.

**Ход работы**

В начале было разработано приложение, генерирующее набор изображений для обучения и проверки работы нейросети из исходного цветного.





Каждое изображение является набором, содержащим 1024 чёрно-белых изображения, каждое размером 16x16 пикселей. В изображения подмешивается яркостный шум, поворот, сдвиг пикселей, изменение яркости. Имеется возможность настройки каждого параметра. Полученные изображения сохраняются в 4 файла (для каждого изображения индивидуально).

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

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

-------------------- TRAIN DATA --------------------

Shape of train\_images: (4096, 256)

Shape of train\_labels: (1, 4096)

Arrays:

[[0.75294118 0.72156863 0.91764706 ... 0.27843137 0.30588235 0.26666667]

[0. 0. 0. ... 0. 0.06666667 0.04705882]

[0. 0. 0. ... 0.71764706 0.70588235 0.84705882]

...

[0.17647059 0.09019608 0.09803922 ... 0.68627451 0.3372549 0.25490196]

[0.29411765 0.56078431 0.58039216 ... 0.29803922 0.52941176 0.49411765]

[0. 0. 0. ... 0.11372549 0.05490196 0.05490196]]

[[3 1 3 ... 2 2 1]]

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

Далее производится генерация весов случайным образом. Первые веса содержат значения {-1; 0; 1}, причем, не более трёх значимых весов. Второй слой имеет диапазон [0; 1].

-------------------- WEIGHTS --------------------

Shape of synaptic\_weights\_0: (512, 256)

Shape of synaptic\_weights\_1: (4, 512)

Arrays:

[[ 0 0 0 ... 0 -1 0]

...

[ 1 0 0 ... 0 0 0]]

[[-0.16595599 0.44064899 -0.99977125 ... -0.51928875 0.14843818

-0.30202481]

[-0.8860712 -0.54237266 0.32820511 ... 0.23492202 -0.13528881

0.69540094]

[-0.09188189 -0.96919296 0.7461363 ... 0.56040467 -0.70842914

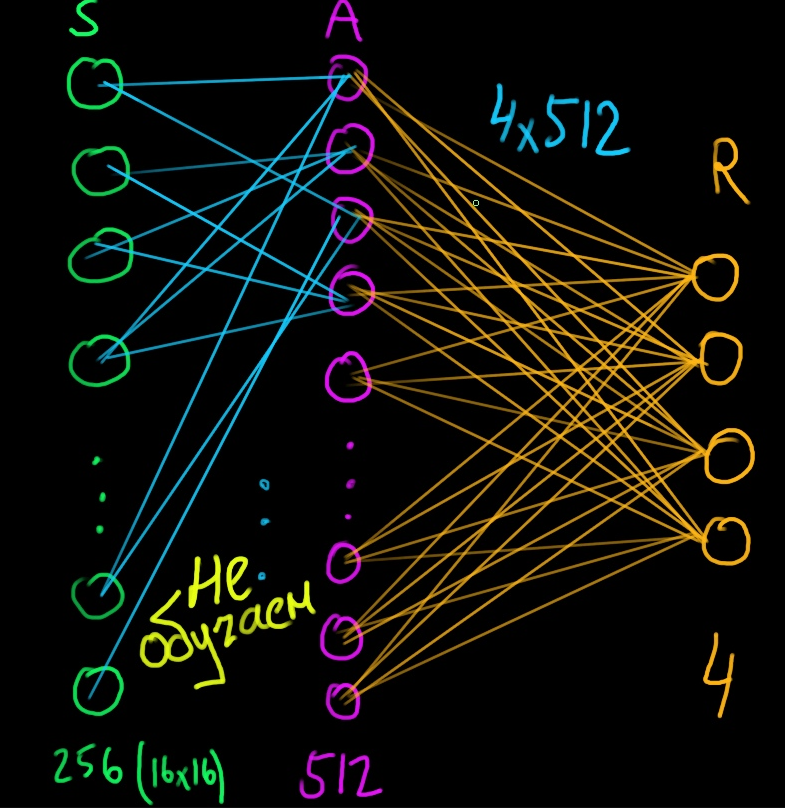
-0.51023872]

[-0.41537248 -0.60813611 -0.82412263 ... -0.65459782 -0.35528502

-0.50315151]]

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

Структура MLP-сети 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 20-30 итераций обучения, чтобы достичь точности в 80-90%  
  
-------------------- I: 0 --------------------

predicted: [3 1 1 ... 2 2 1]

accuracy: 0.25146484375

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

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.61328125

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

predicted: [1 2 2 ... 3 2 3]

accuracy: 0.614501953125

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

predicted: [1 2 2 ... 3 2 3]

accuracy: 0.611083984375

-------------------- I: 4 --------------------

predicted: [1 2 2 ... 3 2 3]

accuracy: 0.615478515625

-------------------- I: 5 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.658935546875

-------------------- I: 6 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.67236328125

-------------------- I: 7 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.6884765625

-------------------- I: 8 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.6875

-------------------- I: 9 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.7099609375

-------------------- I: 10 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.6953125

-------------------- I: 11 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.726806640625

-------------------- I: 12 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.72802734375

-------------------- I: 13 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.75048828125

-------------------- I: 14 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.75390625

-------------------- I: 15 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.7763671875

-------------------- I: 16 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.782470703125

-------------------- I: 17 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.807861328125

-------------------- I: 18 --------------------

predicted: [1 2 1 ... 3 2 3]

accuracy: 0.799072265625

-------------------- I: 19 --------------------

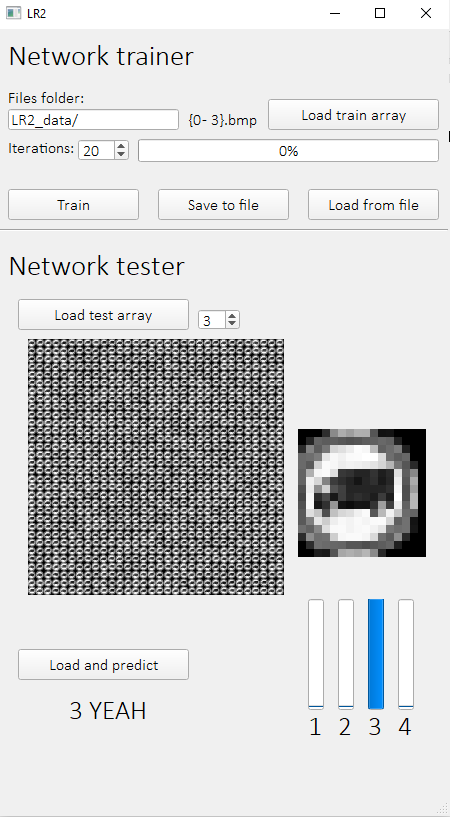
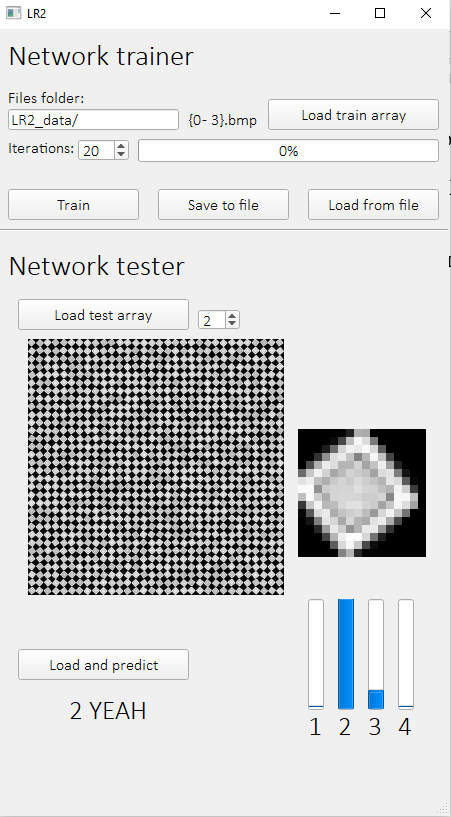
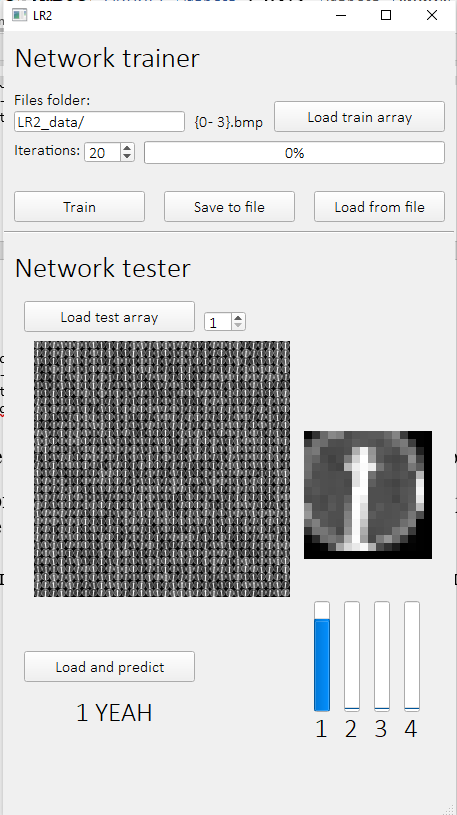
predicted: [1 2 1 ... 3 2 3]

accuracy: 0.83544921875

Обученные веса можно сохранить в файл для дальнейшего использования.

Для проверки работы нейросети используются новые наборы изображений (на которых не проходило обучение)

Результат предсказания отображается в виде выходов последнего слоя



**Вывод**

В ходе данной работы была создана нейросеть с возможностью обучения, способная с приемлемой точностью классифицировать изображения.

**Исходный код генератора изображений**

import numpy as np  
import cv2  
import sys  
from PyQt5 import QtWidgets  
from PyQt5.QtGui import QPixmap  
import qimage2ndarray  
import random  
  
import gui\_2  
  
  
def valmap(value, istart, istop, ostart, ostop):  
 return ostart + (ostop - ostart) \* ((value - istart) / (istop - istart))  
  
  
class CollectorApp(QtWidgets.QMainWindow, gui\_2.Ui\_MainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setupUi(self)  
 self.btn\_load\_original.clicked.connect(self.load\_original)  
 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 load\_original(self):  
 self.original\_image = cv2.cvtColor(cv2.imread(**'LR2\_data/signs\_64x256.bmp'**), cv2.COLOR\_BGR2GRAY)  
 self.cvl\_original.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(self.original\_image, cv2.COLOR\_GRAY2RGB))))  
  
 def generate\_candidates(self):  
 self.candidates\_image = np.zeros(shape=[512, 512], dtype=np.uint8)  
 from\_id\_x = (self.from\_id.value() - 1) \* 64  
 original\_resized\_image = cv2.resize(self.original\_image[0: 64, from\_id\_x: from\_id\_x + 64], (16, 16))  
  
 for x in range(32):  
 for y in range(32):  
 *# Brightness change + brightness noise* temp\_image = original\_resized\_image.copy().astype(**'float32'**)  
 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'**)  
  
 if self.slider\_noise.value():  
 for i in range(16):  
 for j in range(16):  
 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))  
  
 self.candidates\_image[y \* 16: y \* 16 + 16, x \* 16: x \* 16 + 16] = temp\_image *# y:y + h, x:x + w* self.cvl\_candidates.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(self.candidates\_image, cv2.COLOR\_GRAY2RGB))))  
 cv2.imwrite(**'LR2\_data/'** + str(self.spin\_cindidates\_n.value()) + **'.bmp'**, self.candidates\_image)  
  
  
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  
*# from PyQt5.QtGui import QPalette, QColor*import qimage2ndarray  
import pickle  
  
import gui\_2\_2  
  
  
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)  
 *# print(result)  
 # exit(0)* return np.array(result)  
  
  
class CollectorApp(QtWidgets.QMainWindow, gui\_2\_2.Ui\_MainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setupUi(self)  
 self.btn\_load\_train\_array.clicked.connect(self.load\_train\_array)  
 self.btn\_start\_training.clicked.connect(self.start\_training)  
 self.btn\_load\_test\_array.clicked.connect(self.load\_test\_array)  
 self.btn\_predict.clicked.connect(self.predict\_test\_image)  
 self.btn\_save\_to\_file.clicked.connect(self.save\_to\_file)  
 self.btn\_load\_from\_file.clicked.connect(self.load\_from\_file)  
 self.test\_values = False  
  
 self.synaptic\_weights\_0 = np.array([])  
 self.synaptic\_weights\_1 = np.array([])  
 self.train\_images = np.array([])  
 self.train\_labels = np.array([])  
 self.test\_array = np.array([])  
 self.test\_labels = np.array([])  
 self.test\_image = None  
  
 def load\_train\_data(self, debug, folder):  
 train\_images = []  
 train\_labels = []  
 for i in range(4):  
 train\_data\_temp = cv2.cvtColor(cv2.imread(folder + str(i) + **'.bmp'**), cv2.COLOR\_BGR2GRAY) / 255.0  
 for x in range(32): *# 32* for y in range(32): *# 32* temp\_data = train\_data\_temp[y: y + 16, x: x + 16].flatten()  
 train\_images.append(temp\_data)  
 train\_labels.append(i)  
  
 combined\_lists = list(zip(train\_images, train\_labels))  
 random.shuffle(combined\_lists)  
 train\_images, train\_labels = zip(\*combined\_lists)  
 self.train\_images = np.array(train\_images)  
 self.train\_labels = np.array([train\_labels])  
  
 *# TEST VALUES* if self.test\_values:  
 self.train\_images = []  
 self.train\_labels = []  
 for \_ in range(50):  
 a = []  
 b = []  
 for \_ in range(8):  
 a.append(random.randrange(0, 10) / 10)  
 b.append(a[0] + a[1])  
 b.append(a[2] + a[3])  
 b.append(a[4] + a[5])  
 b.append(a[6] + a[7])  
 self.train\_labels.append(np.argmax(b))  
 self.train\_images.append(a)  
 self.train\_images = np.array(self.train\_images)  
 self.train\_labels = np.array([self.train\_labels])  
  
 if debug:  
 print(**'-------------------- TRAIN DATA --------------------'**)  
 print(**'Shape of train\_images: '** + str(self.train\_images.shape))  
 print(**'Shape of train\_labels: '** + str(self.train\_labels.shape))  
 print(**'Arrays:'**)  
 print(self.train\_images)  
 print()  
 print(self.train\_labels)  
 print(**'----------------------------------------------------'**)  
  
 def load\_test\_array(self):  
 self.test\_image = cv2.cvtColor(cv2.imread(self.line\_folder.text() + **'test/'** +  
 str(self.spin\_test\_array\_id.value() - 1) + **'.bmp'**),  
 cv2.COLOR\_BGR2GRAY)  
 self.ocl\_test\_array.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(cv2.resize(self.test\_image, (256, 256)), cv2.COLOR\_GRAY2RGB))))  
  
 def predict\_test\_image(self):  
 *# noinspection PyBroadException* try:  
 x = random.randrange(0, 32)  
 y = random.randrange(0, 32)  
 temp\_image = self.test\_image[y \* 16: y \* 16 + 16, x \* 16: x \* 16 + 16]  
 self.ocl\_test\_image.setPixmap(QPixmap.fromImage(qimage2ndarray.array2qimage(  
 cv2.cvtColor(cv2.resize(temp\_image, (128, 128), interpolation=cv2.INTER\_NEAREST), cv2.COLOR\_GRAY2RGB))))  
 self.test\_array = np.array([(temp\_image / 255.0).flatten()])  
  
 self.test\_labels = np.array([[self.spin\_test\_array\_id.value() - 1]])  
 output\_l0 = self.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]) == self.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)  
 except:  
 print(sys.exc\_info())  
  
 def save\_to\_file(self):  
 compressed\_data = [self.synaptic\_weights\_0, self.synaptic\_weights\_1]  
 with open(self.line\_folder.text() + **'model.txt'**, **'wb'**) as filehandle:  
 pickle.dump(compressed\_data, filehandle)  
  
 def load\_from\_file(self):  
 with open(self.line\_folder.text() + **'model.txt'**, **'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 load\_train\_array(self):  
 random.seed = 1  
 np.random.seed(1)  
 self.load\_train\_data(True, self.line\_folder.text())  
  
 *# Synaptic weights arrays* self.synaptic\_weights\_0 = []  
 for i in range(4096): *# 4096* string\_array = [int(random.randrange(-1, 2)) for \_ in range(3)] + [0 for \_ in range(253)]  
 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, 4096)) - 1) *# 4096  
  
 # TEST VALUES* if self.test\_values:  
 self.synaptic\_weights\_0 = []  
 for i in range(16):  
 string\_array = [int(random.randrange(-1, 2)) for \_ in range(3)] + [0 for \_ in range(5)]  
 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, 16)) - 1)  
 if True:  
 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):  
 thread = threading.Thread(target=self.training)  
 thread.start()  
  
 def training(self):  
 *# noinspection PyBroadException* try:  
 i = 0  
 while i < int(self.spin\_iterations.value()):  
 output\_l0 = self.train\_images  
  
 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 == self.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]) == self.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 = CollectorApp()  
 window.show()  
 app.exec\_()