Gépi látás

GKNB INTM038

Karakterfelismerés dokumentáció

Schöffer Fruzsina Zsuzsanna

TSH86V

Győr, 2019. május 31.

Tartalomjegyzék

1. Bevezetés **2**

2. Neurális háló keras **3**

3. Felismerés tesseracttal **8**

4. Összefoglalás **13**

1. Bevezetés

Napjainkban egyre többször hallani a mesterséges intelligenciáról és térhódításáról, hiszen egyre több, régen igen nehezen megoldható probléma megoldása miatt fordulnak hozzá. Ilyen összetettebb problémakörnek tekinthető maga a gépi látás és a képfeldolgozás is, melyek egyre inkább fontosabbak lesznek bizonyos intelligens rendszerek (például önvezető autók) kiépítésében is. A képfeldolgozás és a mesterséges intelligencia egyik összefonódása például a mai okostelefonok. Az ezekkel készített képekről a telefonunk sok esetben már automatikusan meghatározza hogy mi, vagy éppen ki látható a képeken és csoportosítja is ezeket. Az eszközök másik kényelmi funkciója, hogy karaktereket rajzolunk a képernyőre és az egyes alkalmazások azt egy bizonyos karakterré alakítják. Utóbbi esetén már kézírásfelismerésről beszélhetünk.

A képfeldolgozás egyik problémája maga a kézírásfelismerés, mely már a 19. század végén is foglalkoztatta a kutatókat, és a technológia fejlődésével mára már gyakorlatilag az összes manapság használatos okoseszközünk is képes rá. Míg régebben eleinte a vakoknak készített olvasógépek elkészítésében, nem sokkal utána már a Morse kód olvasásában volt nagy szerepe. Az ötvenes évektől kezdve ugyanakkor már árucímkék és útlevelek szkennelésére is használták. Manapság pedig a technológia fejlődésével már sok okoseszközben is hasznát vehetjük.

Felmerülhet a kérdés, hogy miért van szükség a mesterséges intelligenciára a kézírásfelismerés megoldására, hiszen karakterek felismeréséről van szó, ami már a 20. században is működött képfeldolgozási algoritmusokkal. Ezek az algoritmusok ugyanakkor nem feltétlen képesek megkülönböztetni az egymástól eltérő írásmódokat, a különböző karaktertulajdonságokat, míg az emberi agy viszont igen. Egy gépi rendszer nem képes gondolkodásra, mint az ember, viszont egy ahhoz hasonló működésű már képes lehet erre.

A képfeldolgozási algoritmusok ugyanakkor nem vesznek el, hanem úgymond átalakulnak, és egy másik feladatkört töltenek be a kézírásfelismerésnél. Ezek a felismerési rendszer bemenetének az előfeldolgozására vannak használva. A felismerést, mely az emberi agy szerinti relációk alapján végzi azt, pedig olyan rendszerek végzik, mint a neurális hálók, vagy a rejtett Markov modell.

A kézírásfelismerés egy olyan tág fogalom, ami alatt érthetjük a kézzel írott, vagy akár a nyomtatott dokumentumok szövegének felismerését, illetőleg az okoseszközökön bevitt írásokat is. A kézzel írott, vagy akár nyomtatott dokumentumok alapvetően oldalakból, sorokból, szavakból, és legvégső soron karakterekből állnak. Ahhoz, hogy a folyamat sikeres legyen, atomi szinten kell annak nekiállni, vagyis a karaktereket kell tudni felismerni, amiből a szavak, azokból a mondatok, sorok, és végül az oldalak lesznek.

A karakterfelismerés egyik legismertebb megközelítése a neurális hálók, így a félév során elsősorban egy neurális háló elkészítése volt a célom a keras library segítségével, mely a befejezés végére 99.3 százalékos pontossággal ismeri fel az MNIST adatbázis tesztelésre szánt karaktereit.

Mivel ebben a kódban úgy éreztem, hogy nem hasznosítottam a félév során tanultakat, így célom volt egy olyan kód megírása, ahol ezzel dolgozok. Ebből kifolyólag a pytesseract library segítségével elkészítettem egy rövid kódot, ahol a képek előfeldolgozása volt a lényeges feladat.

A dokumentációban ebből kifolyólag tehát mindkét kód részletesebben ismertetésre kerül.

Githubon Kod1 mappa alatt a neurális háló neuralnet.py néven és annak tartozékai találhatók. Ezek az MNIST adatbázis (manuális beimportálás esetén) mnist.pkl.gz néven, egy kimentett modell neuralnet.h5 néven, és egy tesztelési kód modeltest.py néven. A Kod2Tesseract mappában a felismer.py tartalmazza a forráskódot, a további test1/2/3.jpg/png név alatt a teszthez szükséges képek találhatók.

A megvalósításhoz Python 3-mat és ennek több libraryjét használtam, a saját IDLE ide-jével mind Windows 10, mind Linux Mint alatt.

2. Neurális háló keras

A tesztek megvalósításához egy, a Pythonhoz készült keras library segítségével írt neurális hálót használtam. A kód megírásában a keras saját dokumentációja nyújtott segítséget.

A kódhoz alapvetően több Python libraryre is szükség volt, melyek a következők voltak: Matplotlib, Matplotlib pyplot, Pandas, Sys, Pickle, Gzip, Keras, Numpy.

A matplotlib és a numpy a tesztadatok mentésében és ábrázolásában, míg a sys, gzip és a pickle az MNIST adatbázisának beolvasásában/importálásában játszott szerepet.

Itt megemlítendő, hogy, bár lehetséges, hogy az adatbázis importálása az utolsó 3 library segítsége nélkül működjön csakis a keras segítségével, sajnos Windows 10 alatt a kód írása során egyszer sem tudtam sikeresen letölteni ennek a segítségével az adatbázist, mivel az éppen adott távoli kiszolgáló nem volt elérhető.

Ennek megkerülésére az adatbázist manuálisan töltöttem le, majd az említett libraryk segítségével importáltam be. (Ezt meg lehet kerülni azzal is, ha nem ide-ben futtatjuk a kódot, hanem command line segítségével.)

Linux Mint alatt az adatbázis betöltése az mnist.load_data() függvénnyel ugyanakkor pár másodperc alatt tökéletesen működik.

Az arra való tekintettel, hogy könnyebb az utóbb említettet begépelni, mint az alábbit kódot, így ez maradt benn a kódban és Githubra feltöltésre került az ehhez szükséges adat is.

1. kódrészlet. Forráskód az MNIST adatbázis manuális beimportálására

```
1  f = gzip.open('mnist.pkl.gz', 'rb')
2  if sys.version_info < (3,):
3      data = pickle.load(f)
4  else:
5      data = pickle.load(f, encoding='bytes')
6  f.close()
7
8  from keras.models import Sequential, load_model
9  from keras.layers.core import Dense, Dropout, Activation
10 from keras.utils import np_utils
11 (X_train, y_train), (X_test, y_test) = data
```

A kerasból beimportálásra került a szekvenciális modell felépítéséért felelős egysége, illetve több elemi egység (pl. aktivációs függvények, rétegek) is. Miután a megfelelő eszközök importálása megtörtént, a betanítás megkezdése előtt először muszáj a képeket előfeldolgozni.

A tesztelésre és betanításra szánt képeket először is vektorrá alakítjuk, majd normalizáljuk. Mivel a képek 28*28 pixel méretűek, így a vektor 784 pixel méretű kell, hogy legyen. Utóbbi segít felgyorsítani a betanítást, és elkerülni azt, hogy egy lokális minimumon ragadjon az eljárás.

A betanításhoz az MNIST adatbázis elemei lettek felhasználva, amiben pontosan 60.000 karakter van betanításra, és további 10.000 pedig validálásra. Ahhoz, hogy pontos eredményeket adjon a háló, a betanítási és validálási adatokat muszáj még átalakítani float-tá.

2. kódrészlet. Forráskód a modell betanítására

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

Ezek után történik az úgynevezett One-hot encoding, ami lényegében azért felel, hogy egy olyan vektort hozzon létre, aminek hossza az osztályozási lehetőségek számával megegyezik. A kódban a számjegyek felismerése a cél, így a hossz alapvetően 10 lesz. Jelöljük a vektort v-vel, és amennyiben 2-est kapunk eredményül az a következőképp fog kinézni:

v=[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

3. kódrészlet. Forráskód a one-hot encodingról

```
n_classes = 10

Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
```

Az előfeldolgozás megtörténte után kerül a modell összeállításra:

4. kódrészlet. Forráskód a modell felépítésére

```
model = Sequential()

model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))
```

A modell alapvetően 2 rétegű (bemeneti, egy rejtett, kimeneti). A bemeneti réteg 784 csomóponttal fog rendelkezni, a képek 28*28 pixeles mérete miatt. Ezután következik a rejtett réteg egy adott aktivációs függvénnyel (ami ez esetben a ReLU), majd pedig a kimeneti réteg, ami 10 csomóponttal rendelkezik a lehetőségek száma szerint softmax aktivációs függvénnyel.

A kódban fellelhető egy „Dropout” attribútum is, ami a túlilleszkedést hivatott elkerülni. Ilyenkor néhány súly értékét megtartjuk, és nem frissítjük. Az értéke 0,2, ami azért nem lett nagyobb, mivel a háló mérete sem túl nagy, így nem lenne ideális.

5. kódrészlet. Forráskód a modell összeállítására

```
model.compile(loss='categorical_crossentropy', metrics=['accuracy'
    ], optimizer='adam')
```

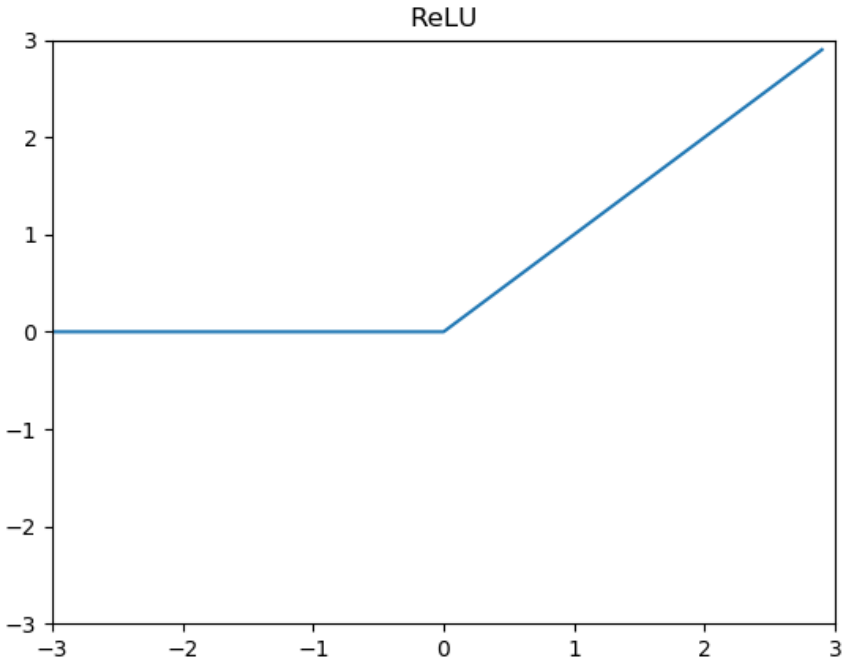
A kód következő részében a keras segítségével a modell össze lesz állítva, és itt adható meg a veszteségfüggvény, illetve az optimalizáló is. Az optimalizálónak az elterjedt „adam”-ot használtam, ami az adaptív lendület módszere. Ez a tanulási rátát befolyásolja, és meghatározza, hogy az optimális súlyok milyen gyorsasággal lesznek kiszámítva. Egy alacsonyabb ráta bizonyos szintig pontosabb súlyokat eredményezhet, viszont figyelembe kell venni, hogy a futási idő így nő.

Az összeállítás után kezdődik a modell betanítása:

6. kódrészlet. Forráskód a modell paramétereire

```
history = model.fit(X_train, Y_train,
          batch_size=64, epochs=10,
          verbose=2,
          validation_data=(X_test, Y_test))
```

A kódban szabadon módosítható, hogy éppen milyen aktivációs függvényt használjunk, esetemben én ReLU-t használtam, mivel az bizonyult a legnagyobb pontosságúnak. Az alábbi ábrán maga a függvény látható.



1. ábra. A ReLU függvény grafikus ábrázolása
Forrás: Saját ábra

A függvényhez szükséges kód az alábbiakban látható:

7. kódrészlet. Forráskód a függvényhez

```
import numpy as np
import matplotlib.pyplot as plt

z = np.arange(-3, 3, .1)
zero = np.zeros(len(z))
y = np.max([zero, z], axis=0)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, y)
ax.set_ylim([-3.0, 3.0])
ax.set_xlim([-3.0, 3.0])
ax.set_title('ReLU')

plt.show()
```

A kód tartalmaz pár olyan további részletet, ami az eredményeket menti ki txt-be, a futási időt mutatja, vagy éppen függvényen ábrázolja őket. Az alábbi kódrészlet is ezt mutatja be:

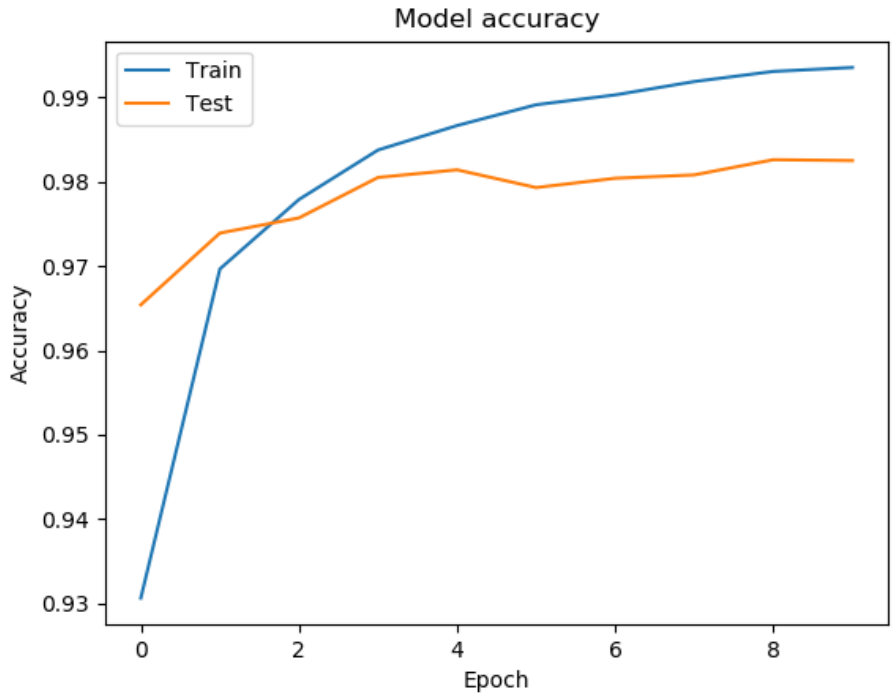
8. kódrészlet. Forráskód az eredmények grafikus ábrázolásához

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

Ez a 2. ábrát eredményezi.

A kód mellé ugyanakkor egy kisebb, tesztelési kódot is írtam, ami a már elkészített modellt importálja be, és az adatbázis kijelölt elemeit ellenőrzi. Itt megjegyzendő, hogy az adatbázis itt nem manuálisan kerül importálásra, viszont az előzőekben említett részletet (1. kódrészlet) bemásolva ez is megvalósítható, amennyiben esetleg ez nem működne.



2. ábra. A modell pontosságának fejlődése:
Forrás: Saját ábra

9. kódrészlet. Forráskód a teszteléshez

```
1  from keras.models import load_model
2  from keras.preprocessing import image
3  import numpy as np
4  import cv2
5  import matplotlib
6  from keras.datasets import mnist
7  import matplotlib.pyplot as plt
8
9  model=load_model('./neuralnet.h5')
10 model.compile(loss='categorical_crossentropy', metrics=['accuracy'
       ], optimizer='adam')
11 model.load_weights('./neuralnet.h5')
12 (X_train, y_train), (X_test, y_test) = mnist.load_data()
13
14 image_index = 334
15 plt.imshow(X_test[image_index].reshape(28, 28))
16 pred = model.predict(X_test[image_index].reshape(1, 784))
17 print(pred.argmax())
18 plt.show()
```

Kikommentezve ez a neuralnet.py fájlban is megtalálható, amennyiben valakinek ez kényelmesebb tesztelést jelent.

3. Felismerés tesseracttal

Mivel a neurális háló elkészítéséhez nem használtam az órán tanultakat, úgy éreztem, hogy a megszerzett ismereteket mégiscsak be kéne mutatni, így elkészítettem egy második kódot. A kód elkészítéséhez szükség volt az opencv, pytesseract és numpy librarykre. A pytesseract használatához Linux mint operációs rendszeren dolgoztam. Fontos megjegyezni, hogy a tesseract kézzel írott szövegekkel nehezen dolgozik, így az MNIST adatbázisnak a tesztelése itt nem célszerű, ezért nyomtatott szövegekkel lett tesztelve.

A kód alapvetően a tesseract image_to_string() függvényén alapult. A működéshez gyakorlatilag elég lenne annyi is, hogy egy kép beolvasásra kerül, majd át van adva ennek a függvénynek paraméterként, viszont ez sok esetben nem ad megfelelő eredményt.

Ebből kifolyólag szükséges volt a képek előfeldolgozása. Az előfeldolgozáshoz a preproc(imgpath) függvény került implementálásra, ami a javított képet adja vissza. A függvényben zajtalanítás és szürkeárnyalattá konvertálás történik.

10. kódrészlet. Forráskód a preproc függvényről

```
1 def preproc(imgpath):
2     img=cv2.imread(imgpath)
3     img=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
4     kernel=np.ones((1,1), np.uint8)
5     img=cv2.dilate(img, kernel, iterations=1)
6     img=cv2.erode(img, kernel, iterations=1)
7     #img = cv2.adaptiveThreshold(img, 255, cv2.
          ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 31, 2)
8     cv2.imwrite(srcpath+ "thresh.png", img)
9     return img
```

Az opencv dokumentációja alapján zajtalanítás esetén általános mód, hogy először az erode, majd a dilate funkció kerül használatra ebben a sorrendben. Ennek oka, hogy az erosion bár eltünteti a zajt, mégis vékonyítja kicsit a vizsgált objektumot, így a dilate-tal ezt kövérítjük. A kódban is ezt követtem, viszont kíváncsiságból felcseréltem a két funkciót. Meglepetésemre a kettő felcserélése jobb eredményt hozott, így a későbbiekben is ez a sorrend került használatra.

11. kódrészlet. Erosion és dilate sorrendje

```
1 kernel=np.ones((1,1), np.uint8)
2 img=cv2.dilate(img, kernel, iterations=1)
3 img=cv2.erode(img, kernel, iterations=1)
```

A függvényben a javított kép ugyanakkor elmentésre is került, amit utána a getstring() függvény használ majd. A javítás után a getstring() függvény került implementálásra, aminek a dolga a tessereact feldolgozó függvényének meghívása, amivel maga a felismerés történik. Az eredmény ezek után egy result nevű txt fájlba lesz elmentve, illetve a függvény meghívásakor kiiratva.

12. kódrészlet. Forráskód a getstring függvényről

```
1  def getstring():
2      config= ("-psm 7")
3      result=pytesseract.image_to_string(Image.open(srcpath + "thresh
           .png"),config=config)
4      file=open("result", "w")
5      file.write(result)
6      return result
```

Itt megjegyzendő, hogy a függvény egyik paramétere a config, amivel beállíthó, hogy az esetleg neurális hálóval, egy adott nyelven, illetve, hogy milyen szegmentáció alapján ellenőrizzen. Az nyelvi beállítás eleinte a kód részét képezte, viszont annak ellenére, hogy a language packből a magyar telepítésre került, ez semmiben sem javította a felismerést magyar nyelvű szövegeken, csupán rontotta azt. A kód esetében psm (lapszegmentáció) a default 3-mas értéket, az oem (OCR engine módok) pedig az 1-es értéket kapta, ami neurális hálókra és LSTM enginekre vonatkozik.

A kódban a harmadik függvény a boxes(imgpath), aminek érdemi nem, de szemléletes funkciója annál inkább van. Itt ugyancsak a tesseract saját függvénye kerül meghívásra, amit egy for ciklussal kiegészítve a felismert szavak kerülnek bekeretezésre. Az így kapott kép a kód mappájába ugyanúgy mint az előzőekben elmentésre kerül. Ez nem létfontosságú funkció, viszont a szemléltetés miatt bele került implementálásra.

13. kódrészlet. Forráskód a boxes függvényről

```
1  def boxes(imgpath):
2      d = pytesseract.image_to_data(imgpath, output_type=Output.DICT)
3      n_boxes = len(d['level'])
4      for i in range(n_boxes):
5          (x, y, w, h) = (d['left'][i], d['top'][i], d['width'][i], d
               ['height'][i])
6          cv2.rectangle(imgpath, (x, y), (x + w, y + h), (0, 255, 0),
               2)
7      cv2.imshow("im", imgpath)
8      cv2.imwrite(srcpath+"boxedimg.png",imgpath)
9      cv2.waitKey(0)
10     cv2.destroyAllWindows()
```

A teljes működéshez végül a fenti függvények meghívása szükséges már csak:

14. kódrészlet. Függvények meghívása

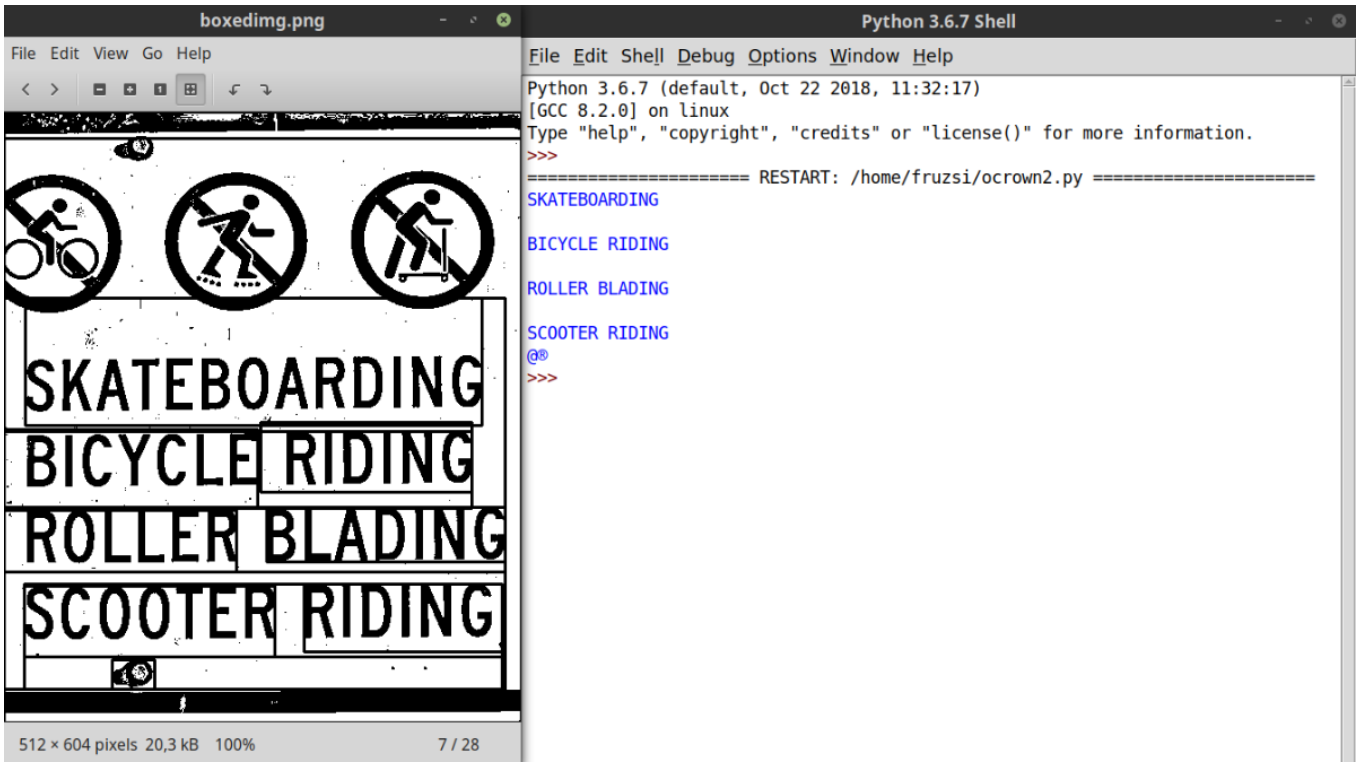
```
1  preprocimg=preproc(srcpath+imgname)
2  boxes(preprocimg)
3  print(getstring())
```

A felismerés ezzel a kóddal természetesen nem olyan pontos mint mondjuk a neurális hálóval. A kód pontosságát nagyban befolyásolja a megfelelő zajtalanítás és előfeldolgozás, viszont ezzel a ló másik oldalára is át lehet esni. Erre ékes példaként szolgál az alábbi sor, ami alapvetően kikommentezve van a kódban, mivel inkább ront, mint javít, viszont előfordult olyan eset is, hogy javított.

15. kódrészlet. A kérdéses sor

```
1 img = cv2.adaptiveThreshold(img, 255, cv2.
    ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 31, 2)
```

Erre a javításra példa az alábbi:



3. ábra. Megfelelő felismerés
Forrás: Saját ábra

A 3. ábrán az látható, ahogyan a zaj elüntetésre kerül, és milyen eredményt ismer fel a tesseract.

A sor ezután ki lett kommentelve, ahol viszont a 4. ábrán látható eredményt kaptuk.

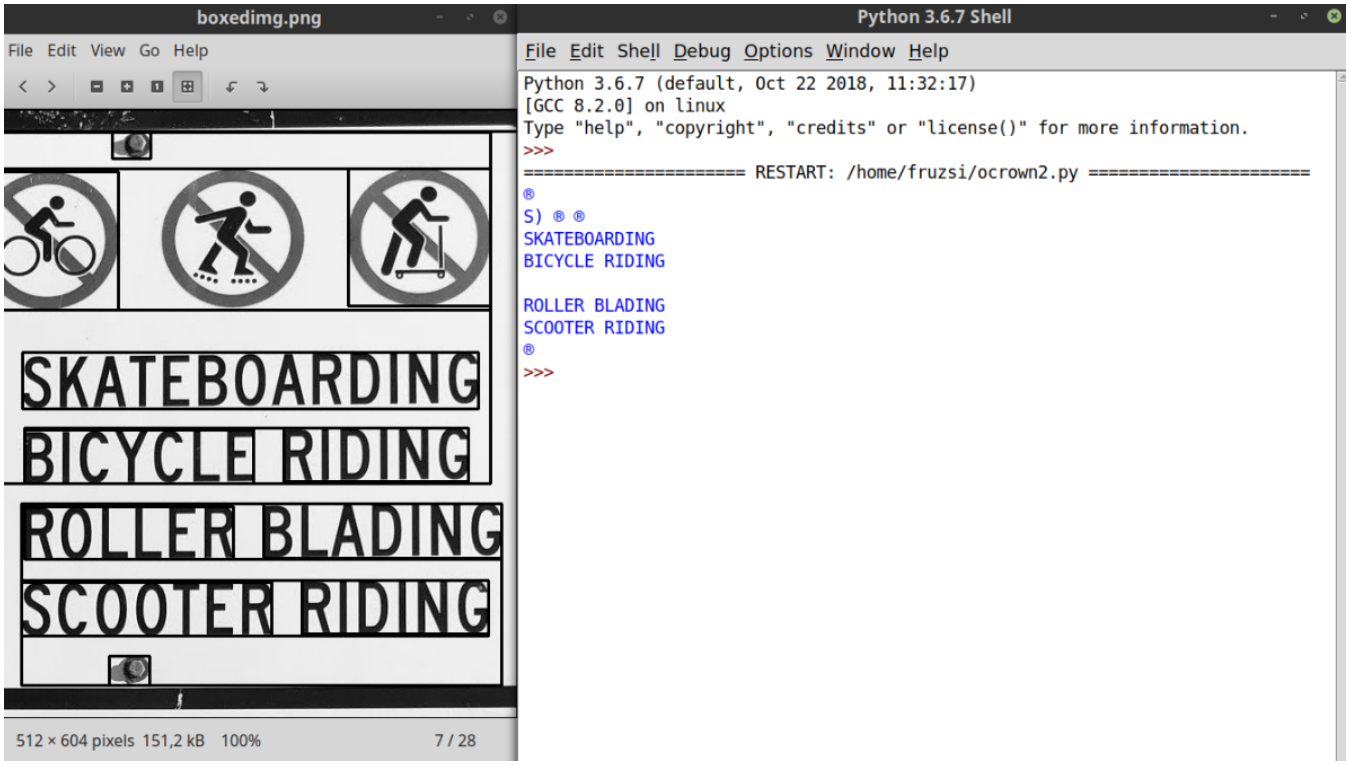
Látható tehát, hogy a sor nélkül több képrészletet (pl.csavar) is szövegként értelmez.

Ugyanakkor egy másik képet megfigyelve, ahol a sor nincs kikommentelve, igen rossz eredményt kaphatunk, ez az 5. ábrán látszik.

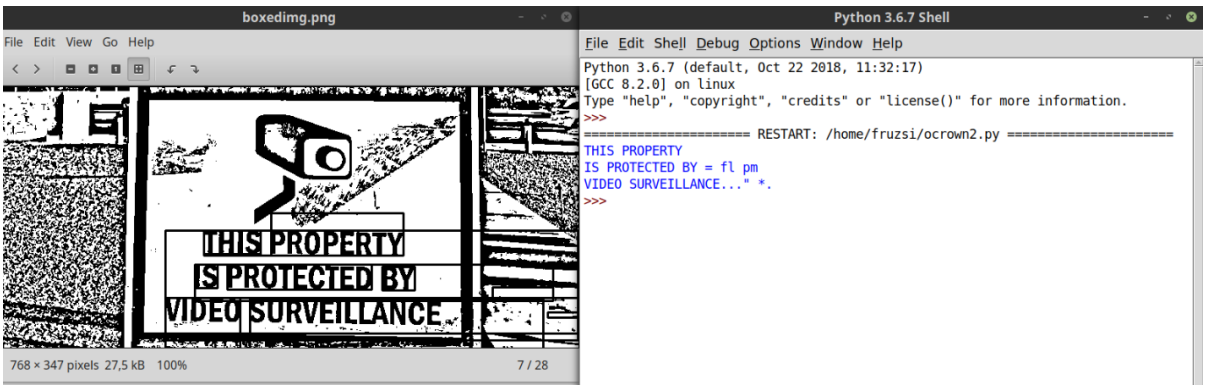
A sort újra kikommentelve az előőzekben felismert fantomszövegek már nem látszanak.

Több példán ezt megfigyelve arra jutottam, hogy a sor többet árt, mint használ, ezért véglegesen ki lett kommentelve.

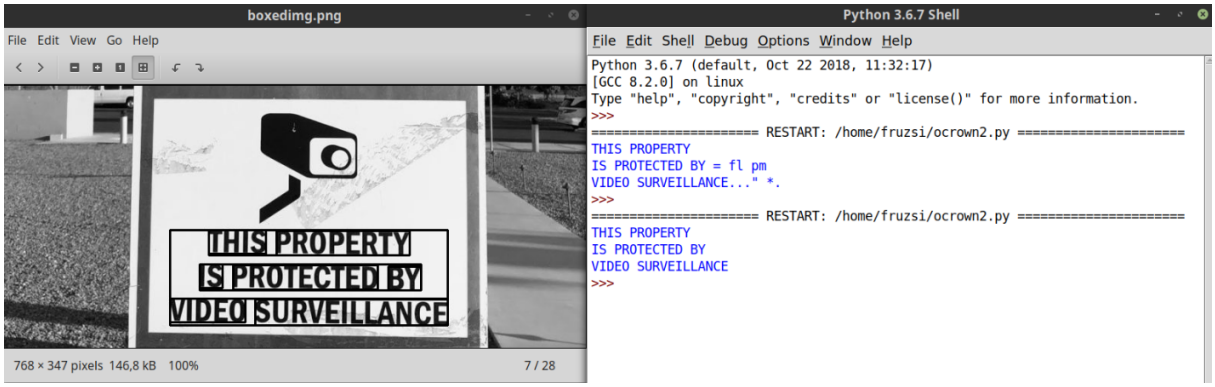
Githubra a teszteléshez feltöltésre került pár kép, amikkel a kód működése bemutatható. Megfigyeltem a tesztelés során ugyanakkor, hogy ha valami igen rossz eredményt ad a kód, akkor a következő futtatáskor a kapott eredmények kiírása előtt beragad. Ilyenkor egy újrafuttatás szükséges, és utána újból megfelelően működik.



4. ábra. Nem megfelelő felismerés
Forrás: Saját ábra



5. ábra. Nem megfelelő felismerés
Forrás: Saját ábra



6. ábra. Megfelelő felismerés
Forrás: Saját ábra

4. Összefoglalás

Az előzőek alapján, futtatva az egyes kódokat látható, hogy a kettő igen különbözik egymástól. Mint az a hosszakon is tükröződik, a neurális háló megírása, a keras működésének megértése jóval több időt vett igénybe, mint a tesseracttal való munka.