Универзитет у Крагујевцу
Факултет инжењерских наука

Дипломски рад
Програмирање система који раде у реалном времену

Детекција карата за игру коришћењем YOLO алгоритма за детекцију објеката

Студент:
Алекса Величковић 576/2015

Професор:
др Владимир Миловановић

Крагујевац, септембар 2020.

Садржај

1 Увод **1**

2 Припрема тренинг скупа **2**
2.1 Добијање слика из видеа 2
2.2 Означавање знака и броја карте на слици 7
2.3 Прављење скупа за тренирање 11
2.4 Превођење XML датотека у формат YOLO алгоритма 19

3 Вештачке неуронске мреже - теорија **20**
3.1 Увод у вештачке неуронске мреже 20
3.2 Конволуцијске неуронске мреже 27
3.3 YOLO алгоритам 33

4 Тренирање - Darknet програмски оквир **42**
4.1 Промене конфигурација током тренирања 44
4.2 Резултати тренирања 44

5 Закључак **45**

Литература **46**

1 Увод

У овом раду ће бити описана реализација система за детекцију карата за игру. Биће објашњена припрема слика над којима се тренира вештачка неуронска мрежа, начин рада YOLO алгоритма, поступак тренирања вештачке неуронске мреже и начин покретања система.

Свака карта је снимљена под различитим осветљењем у кратком видеу из кога се добијају слике; дефинисана је позиција знака и броја карте, па се насумичним позиционирањем карте преко насумично изабране позадине генерише скуп слика над којима се врши тренирање.

Тренирање над сликама је извршено коришћењем Darknet програмског оквира за вештачке неуронске мреже и омогућава детекцију над сликом, видеом и у реалном времену са камере.

Пројекат се може посматрати као први корак у прављењу карташке игре која је под контролом рачунара. Рачунар би коришћењем овог система детектовао карте, а у другом кораку би, при свакој детекцији, одредио који играч је ту карту одиграо. Игра би укључивала четири играча и сваки би био придружен страници слике са камере. По детектовању карте, одредило би се са које стране је ушла рука играча у кадар и њему би била додељена детектована карта.

YOLO алгоритам је могуће применити за детекцију било којих објеката и овим пројектом се показује „пут” од снимка карте до програма који њу детектује у потпуно другачијим условима и у стању је да то уради невероватно брзо, чак у реалном времену.

2 Припрема тренинг скупа

Генерисање слика за тренирање је реализовано коришћењем Jupyter свеске `creating_playing_cards_dataset.ipynb`, која се налази у GitHub репозиторијуму, наведеном у литератури [1].

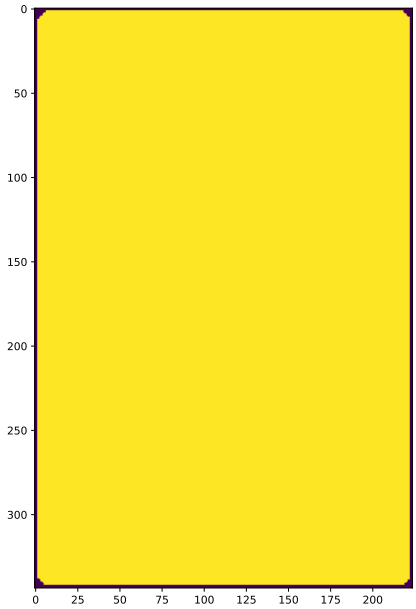
2.1 Добијање слика из видеа

Свака карта је снимљена приближно 20 секунди под различитим осветљењем. Из сваког снимка је потребно издвојити само карту и сачувати све слике као *.png* датотеке. Програм генрише око 100 слика карата по видеу.

Прво, потребно је направити шаблон тј. правоугаоник, који ће имати однос дужина страница исти као и карта коришћеног шпила. Пошто су у првој „ћелији" (сегменту кода) Jupyter свеске дефинисане дужина и ширина (mm) карте одговарајућег шпила, линијама кода `cardW=56` и `cardH=86` (слика 2), тако се следећим кодом прави шаблон (маска):

```
bord_size=2 # bord_size alpha=0
alphamask=np.ones((cardH,cardW),dtype=np.uint8)*255
cv2.rectangle(alphamask,(0,0),(cardW-1,cardH-1),0,bord_size)
cv2.line(alphamask,(bord_size*3,0),(0,bord_size*3),0,bord_size)
cv2.line(alphamask,(cardW-bord_size*3,0),(cardW,bord_size*3),0,bord_size)
cv2.line(alphamask,(0,cardH-bord_size*3),(bord_size*3,cardH),0,bord_size)
cv2.line(alphamask,(cardW-bord_size*3,cardH),
                                  (cardW,cardH-bord_size*3),0,bord_size)
plt.figure(figsize=(10,10))
plt.imshow(alphamask)
```

Неопходно је да карта буде јасно издвојена од позадине, узимајући у обзир и ћошкове који су закривљени; па се после стварања правоугаоника, позивом функције `cv2.rectangle()`, црта и сваки од ћошкова функцијом `cv2.line()`. Извршавањем дела кода, наведеног изнад, се приказује описани правоугаоник, као на слици 1.



Слика 1: Шаблон за издвајање карте са снимка

Сада се дефинише функција `extract_card()`, која издваја карту из прослеђене слике. На њеном почетку се врши провера замућења слике позивом функције `varianceOfLaplacian()`, која враћа варијансу Лапласове функције позивом `cv2.Laplacian(img,cv2.CV_64F).var()`. Јасна слика би требало да има високу варијансу [2]. У наредном кораку се слика претвара у црно-белу са и над њом позива неколико филтера. Позивом `cv2.bilateralFilter()` се смањује шум на слици и овај филтер не утиче на ивице (прелазе) на слици; након тога се издвајају ивице са слике позивом `cv2.Canny()`. После извршених филтрирања, потребно је наћи контуре на слици коју смо добили (са наглашеним ивицама) и то се извршава са `cv2.findContours()`, где контура представља криву која спаја све тачке дуж границе, исте боје и интензитета [3]. Претпостављено је да највећа препозната контура представља карту, па се она добија сортирањем свих издвојених контура у опадајућем редоследу и издвајањем прве из листе. Проверава се да ли је добијена контура правоугаоног облика, тако што се изврши однос површина контуре и најмањег правоугаоника који је окружује и уколико су вредности приближне променљива `valid` има вредност `True`.

Уколико је утврђено да је пронађена контура правоугаоник, желимо да претворимо простор унутар контуре у правоугаоник димензија карте из коришћеног шпила.

```
def varianceOfLaplacian(img):
    return cv2.Laplacian(img, cv2.CV_64F).var()
def extract_card (img, output_fn=None, min_focus=120, debug=False):
```

```
imgwarp=None

# Check the image is not too blurry
focus=varianceOfLaplacian(img)
if focus < min_focus:
    if debug: print("Focus too low :", focus)
    return False,None

# Convert in gray color
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Noise-reducing and edge-preserving filter
gray=cv2.bilateralFilter(gray,11,17,17)

# Edge extraction
edge=cv2.Canny(gray,30,200)

# Find the contours in the edged image
cnts, _ = cv2.findContours(edge.copy(), cv2.RETR_EXTERNAL,
                           cv2.CHAIN_APPROX_SIMPLE) #_,cnts, _ = cv2.

# We suppose that the contour with largest area corresponds
                                  #to the contour delimiting the card
cnt = sorted(cnts, key = cv2.contourArea, reverse = True)[0]

# We want to check that 'cnt' is the contour of a rectangular shape
# First, determine 'box', the minimum area bounding rectangle of 'cnt'
# Then compare area of 'cnt' and area of 'box'
# Both areas sould be very close
rect=cv2.minAreaRect(cnt)
box=cv2.boxPoints(rect)
box=np.int0(box)
areaCnt=cv2.contourArea(cnt)
areaBox=cv2.contourArea(box)
valid=areaCnt/areaBox>0.95

if valid:
    # We want transform the zone inside the contour into
                #the reference rectangle of dimensions (cardW,cardH)
    ((xr,yr),(wr,hr),thetar)=rect
    # Determine 'Mp' the transformation that transforms
                               #'box' into the reference rectangle
    if wr>hr:
        Mp=cv2.getPerspectiveTransform(np.float32(box),refCard)
    else:
```

```
        Mp=cv2.getPerspectiveTransform(np.float32(box),refCardRot)
    # Determine the warped image by applying
                                #the transformation to the image
    imgwarp=cv2.warpPerspective(img,Mp,(cardW,cardH))
    # Add alpha layer
    imgwarp=cv2.cvtColor(imgwarp,cv2.COLOR_BGR2BGRA)

    # Shape of 'cnt' is (n,1,2), type=int with
                                            #n = number of points
    # We reshape into (1,n,2), type=float32,
                            #before feeding to perspectiveTransform
    cnta=cnt.reshape(1,-1,2).astype(np.float32)
    # Apply the transformation 'Mp' to the contour
    cntwarp=cv2.perspectiveTransform(cnta,Mp)
    cntwarp=cntwarp.astype(np.int)

    # We build the alpha channel so that we have transparency on the
    # external border of the card
    # First, initialize alpha channel fully transparent
    alphachannel=np.zeros(imgwarp.shape[:2],dtype=np.uint8)
    # Then fill in the contour to make opaque this zone of the card
    cv2.drawContours(alphachannel,cntwarp,0,255,-1)

    # Apply the alphamask onto the alpha channel to clean it
    alphachannel=cv2.bitwise_and(alphachannel,alphamask)

    # Add the alphachannel to the warped image
    imgwarp[:,:,3]=alphachannel

    # Save the image to file
    if output_fn is not None:
        cv2.imwrite(output_fn,imgwarp)

if debug:
    cv2.imshow("Gray",gray)
    cv2.imshow("Canny",edge)
    edge_bgr=cv2.cvtColor(edge,cv2.COLOR_GRAY2BGR)
    cv2.drawContours(edge_bgr,[box],0,(0,0,255),3)
    cv2.drawContours(edge_bgr,[cnt],0,(0,255,0),-1)
    cv2.imshow("Contour with biggest area",edge_bgr)
    if valid:
        cv2.imshow("Alphachannel",alphachannel)
        cv2.imshow("Extracted card",imgwarp)

return valid,imgwarp
```

Следећи корак је да се издвоје фрејмови из снимка и над њима позива изнад дефинисана функција; за то је дефинисана функција `extract_cards_from_video()`.

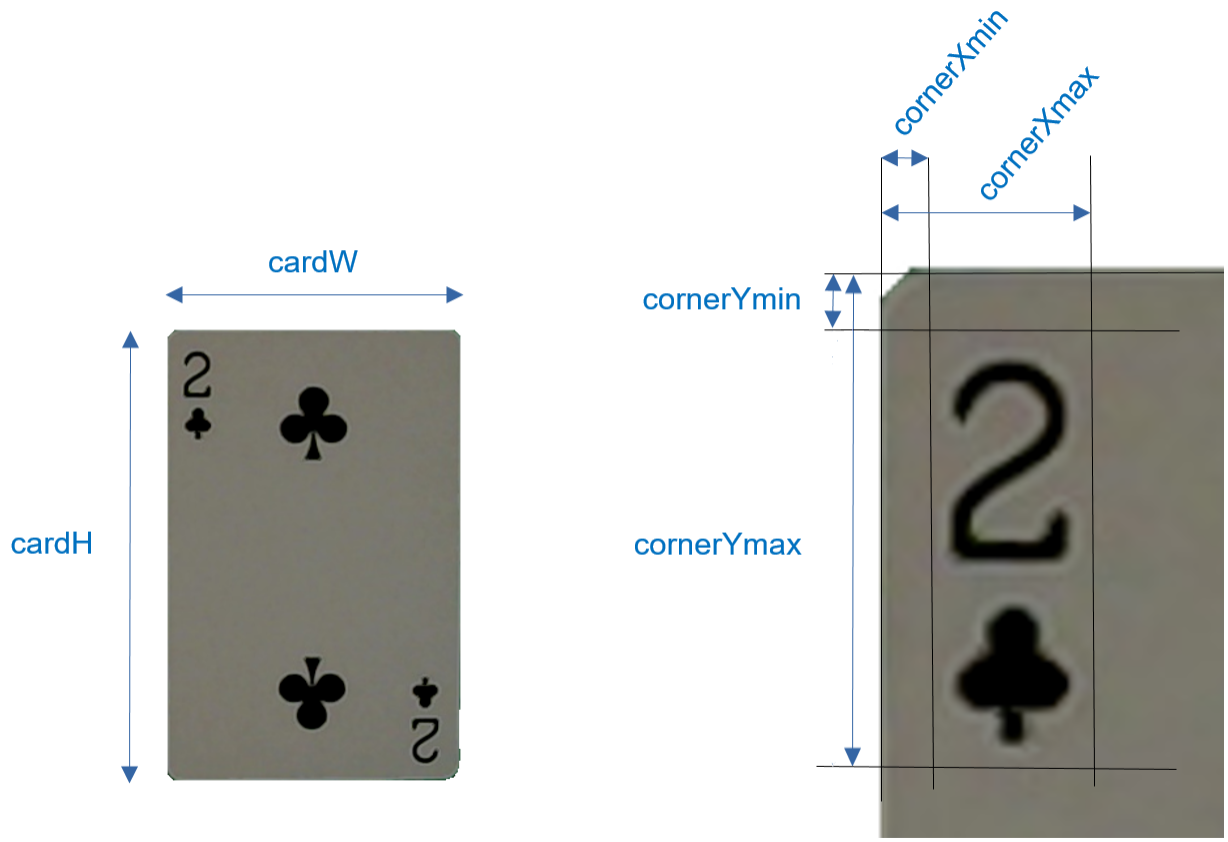
Поред провера да ли постоје улазна видео датотека и излазна фасцикла, где ће бити спаковане слике, отвара се видео датотека коришћењем OpenCV библиотеке са `cap=cv2.VideoCapture(video_fn)`, а онда чита фрејм позивом `cap.read()`. На сваких 5 фрејмова се позива функција `extract_card()`.

```
def extract_cards_from_video(video_fn, output_dir=None, keep_ratio=5,
                                                min_focus=120, debug=False):
    if not os.path.isfile(video_fn):
        print(f"Video file {video_fn} does not exist !!!")
        return -1,[]
    if output_dir is not None and not os.path.exists(output_dir):
        os.makedirs(output_dir)

    cap=cv2.VideoCapture(video_fn)

    frame_nb=0
    imgs_list=[]
    while True:
        ret,img=cap.read()
        if not ret: break
        # Work on every 'keep_ratio' frames
        if frame_nb%keep_ratio==0:
            if output_dir is not None:
                output_fn=give_me_filename(output_dir,"png")
            else:
                output_fn=None
            valid,card_img = extract_card(img,output_fn,
                                  min_focus=min_focus,debug=debug)
            if debug:
                k=cv2.waitKey(1)
                if k==27: break
            if valid:
                imgs_list.append(card_img)
        frame_nb+=1

    if debug:
        cap.release()
        cv2.destroyAllWindows()

    return imgs_list
```

Овако дефинисану функцију је потребно позвати над снимцима свих карата и онда су слике сачуване у фасциклама, посебно за сваку карту.

2.2 Означавање знака и броја карте на слици

Поред дефинисања димензија карте коришћеног шпила, на почетку Jupyter свеске се дефинишу и позиције ћошкова правоугаоника који окружује знак и број карте, као на слици 2, и сачуване су у променљивама `cornerXmin`, `cornerXmax`, `cornerYmin` и `cornerYmax`.



Слика 2: Димензије карте и правоугаоника (око броја и знака)

Потребно је дефинисати функцију која ће унутар овог правоугаоника одредити контуру унутар које ће се налазити број и знак.

Ова функција је названа `find_hull()` и поред слике карте, као аргумент `corner` прима позицију правоугаоника и могуће је проследити позицију горњег-левог и доњег-десног, а уписани су у променљиве `refCornerHL` и `refCornerLR`, респективно.

На почетку се издваја део слике означен позицијама правоугаоника и уписује у променљиву `zone`. Слика се претвара у црно-белу, издвајају се ивице и позивом `cv2.dilate()` се оне појачавају тј. функција ради тако што промени боју пиксела из црне у белу уколико тај црни пиксел има испод себе бели [4]. Издвајају се контуре позивом `cv2.findContours()` и дефинишу се минимална површина (`min_area`) и минимални однос површина посматране контуре и конвексног полигона око ње (`min_solidity`).

Сада се улази у `for` петљу која пролази кроз све нађене контуре; у њој се прво рачуна површина те контуре, а позивом `cv2.convexHull()` се налази конвексан полигон око контуре.

Функција `cv2.moments()` враћа тежиште контуре и потом се врше провере: да ли је површина контуре већа од минимално дефинисане (`min_area`), да ли је однос површина контуре и конвексног полигона који је окружује већи од минимално дефинисаног (`min_solidity`) и да ли се тежиште контуре налази близу центра правоугаоника унутар кога је тражена контура.

Све нађене контуре су спојене унутар променљиве `concat_contour` и сада се одређује коначни полигон око броја и знака, и његова површина. Уколико је површина полигона у границама минималне и максималне дефинисане вредности променљива `ok` и даље има вредност `True`, па је још само потребно позцију полигона дефинисати у односу ивицу карте тј. на израчунату позицију, која је релативна у односу на правоугаоник који је окруживао знак и број, додати и координате самог правоугаоника у односу на карту.

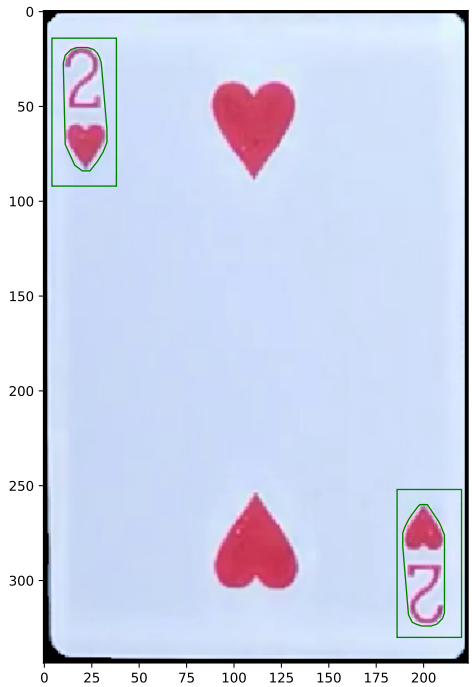
```
def findHull(img, corner=refCornerHL, debug="no"):
    kernel = np.ones((3,3),np.uint8)
    corner=corner.astype(np.int)

    # We will focus on the zone of 'img' delimited by 'corner'
    x1=int(corner[0][0])
    y1=int(corner[0][1])
    x2=int(corner[2][0])
    y2=int(corner[2][1])
    w=x2-x1
    h=y2-y1
    zone=img[y1:y2,x1:x2].copy()

    strange_cnt=np.zeros_like(zone)
    gray=cv2.cvtColor(zone,cv2.COLOR_BGR2GRAY)
    thld=cv2.Canny(gray,30,200)
    thld = cv2.dilate(thld,kernel,iterations=1)
    if debug!="no": cv2.imshow("thld",thld)

    # Find the contours
    contours,_=cv2.findContours(thld.copy(),cv2.RETR_EXTERNAL,
                                cv2.CHAIN_APPROX_SIMPLE) #_,contours,_

    min_area=30 # We will reject contours with small area.
                                            #TWEAK, 'zoom' dependant
    min_solidity=0.3 # Reject contours with a low solidity. TWEAK

    concat_contour=None # We will aggregate in 'concat_contour'
                                  #the contours that we want to keep
```

```
ok=True
for c in contours:
    area=cv2.contourArea(c)

    hull = cv2.convexHull(c)
    hull_area = cv2.contourArea(hull)
    solidity = float(area)/hull_area
    # Determine the center of gravity (cx,cy) of the contour
    M=cv2.moments(c)
    cx=int(M['m10']/M['m00'])
    cy=int(M['m01']/M['m00'])
    #  abs(w/2-cx)<w*0.3 and abs(h/2-cy)<h*0.4 :
    #        TWEAK, the idea here is to keep only the contours
    #                which are closed to the center of the zone
    if area >= min_area and abs(w/2-cx)<w*0.3 and
                    abs(h/2-cy)<h*0.4 and solidity>min_solidity:
        if debug != "no" :
            cv2.drawContours(zone,[c],0,(255,0,0),-1)
        if concat_contour is None:
            concat_contour=c
        else:
            concat_contour=np.concatenate((concat_contour,c))
    if debug != "no" and solidity <= min_solidity :
        print("Solidity",solidity)
        cv2.drawContours(strange_cnt,[c],0,255,2)
        cv2.imshow("Strange contours",strange_cnt)


if concat_contour is not None:
    # At this point, we suppose that 'concat_contour' contains
    #        only the contours corresponding the value and suit symbols
    # We can now determine the hull
    hull=cv2.convexHull(concat_contour)
    hull_area=cv2.contourArea(hull)
    # If the area of the hull is to small or too big,
    #        there may be a problem
    min_hull_area=940 # TWEAK, deck and 'zoom' dependant
    max_hull_area=2120 # TWEAK, deck and 'zoom' dependant
    if hull_area < min_hull_area or hull_area > max_hull_area:
        ok=False
        if debug!="no":
            print("Hull area=",hull_area,"too large or too small")
    # So far, the coordinates of the hull are relative to 'zone'
    # We need the coordinates relative to the image -> 'hull_in_img'
```

```
        hull_in_img=hull+corner[0]

    else:
        ok=False


    if debug != "no" :
        if concat_contour is not None:
            cv2.drawContours(zone,[hull],0,(0,255,0),1)
            cv2.drawContours(img,[hull_in_img],0,(0,255,0),1)
        cv2.imshow("Zone",zone)
        cv2.imshow("Image",img)
        if ok and debug!="pause_always":
            key=cv2.waitKey(1)
        else:
            key=cv2.waitKey(0)
        if key==27:
            return None
    if ok == False:

        return None

    return hull_in_img
```

Позивањем функције `find_hull()` два пута, једном за горњи-леви, а други пут за доњи-десни правоугаоник, добијају се два полигона, као на слици 3. Ово је потребно извршити над свим картама и сачувати полигоне у Python речнику, где је кључ назив карте, а његова вредност је листа која садржи слику карте и позиције два полигона.

Речник карата `cards`, са дефинисаним полигонима око знака и броја, се уписује у датотеку `cards_pck_fn.pck`, позивом `pickle.dump(cards,open(cards_pck_fn,'wb'))`, где је коришћен Pickle модул који служи за претварање Python објеката у ток бајтова и обрнуто [5].



Слика 3: Полигон који окружује знак и број

2.3 Прављење скупа за тренирање

Скуп за тренирање се генерише подвлачењем једне од много позадина испод карата, а оне се распоређују на два начина: насумичним транслирањем, ротацијом и увећавањем две карте; насумичним позиционирањем три карте, али тако да се налазе једна до друге као када се држе у руци.

Дефинисана је класа `Scene`, при чијем се инстанцирању у конструктору позива функција `create2CardsScene()` или `create3CardsScene()`, зависно од тога да ли су прослеђене слике две или три карте као аргументи функције. Као аргумент, конструктор прима и слику позадине, поред слика карата и полигона који окружују њихов знак и број.

Функцијом `create2CardsScene()` се извршава распоређивање две карте. На њеном почетку се позива функција `hull_to_kps()`, за сваки полигон који окружује знак и број тј. њих четири.

```
def create2CardsScene(self,bg,img1, class1, hulla1,hullb1,img2,
                                                class2,hulla2,hullb2):
        kpsa1=hull_to_kps(hulla1)
        kpsb1=hull_to_kps(hullb1)
        kpsa2=hull_to_kps(hulla2)
```

```
kpsb2=hull_to_kps(hullb2)
# Randomly transform 1st card
self.img1=np.zeros((imgH,imgW,4),dtype=np.uint8)
self.img1[decalY:decalY+cardH,decalX:decalX+cardW,:]=img1
self.img1,self.lkps1,self.bbs1=augment(self.img1,
                        [cardKP,kpsa1,kpsb1],transform_1card)
# Randomly transform 2nd card. We want that card 2
#                 does not partially cover a corner of 1 card.
# If so, we apply a new random transform to card 2
while True:
    self.listbba=[]
    self.img2=np.zeros((imgH,imgW,4),dtype=np.uint8)
    self.img2[decalY:decalY+cardH,decalX:decalX+cardW,:]=img2
    self.img2,self.lkps2,self.bbs2=augment(self.img2,
                         [cardKP,kpsa2,kpsb2],transform_1card)
    # mainPoly2: shapely polygon of card 2
    mainPoly2=kps_to_polygon(self.lkps2[0].keypoints[0:4])
    invalid=False
    intersect_ratio=0.1
    for i in range(1,3):
        # smallPoly1: shapely polygon of one of the hull of card 1
        smallPoly1=kps_to_polygon(self.lkps1[i].keypoints[:])
        a=smallPoly1.area
        # We calculate area of the intersection of
        #                         card 1 corner with card 2
        intersect=mainPoly2.intersection(smallPoly1)
        ai=intersect.area
        # If intersection area is small enough, we accept card 2
        if (a-ai)/a > 1-intersect_ratio:
            self.listbba.append(BBA(self.bbs1[i-1],class1))
        # If intersectio area is not small, but also
        # not big enough, we want apply new transform to card 2
        elif (a-ai)/a>intersect_ratio:
            invalid=True
            break
    if not invalid: break
self.class1=class1
self.class2=class2
for bb in self.bbs2:
    self.listbba.append(BBA(bb,class2))
# Construct final image of the scene
#                 by superimposing: bg, img1 and img2
self.bg=scaleBg.augment_image(bg)
mask1=self.img1[:,:,3]
self.mask1=np.stack([mask1]*3,-1)
```

```
        self.final=np.where(self.mask1,self.img1[:,:,0:3],self.bg)
        mask2=self.img2[:,:,3]
        self.mask2=np.stack([mask2]*3,-1)
        self.final=np.where(self.mask2,self.img2[:,:,0:3],self.final)
```

Ова функција рачуна нове позиције полигона, јер се сада на слици налази карта преко позадине, па је нова позиција збир растојања у односу на карту и координата карте, а она се налази на центру слике.

```
def hull_to_kps(hull, decalX=decalX, decalY=decalY):
    # hull is a cv2.Contour, shape : Nx1x2
    kps=[ia.Keypoint(x=p[0]+decalX,y=p[1]+decalY)
                        for p in hull.reshape(-1,2)]
    kps=ia.KeypointsOnImage(kps, shape=(imgH,imgW,3))
    return kps
```

Ван класе `Scene` су дефинисане координате које одговарају центру слике коју генеришемо, и то ће бити почетне позиције карата пре насумичног позиционирања. Димензије слике коју хоћемо да генеришемо су одређене са `imgW=720` и `imgH=720`, а координате почетне позиције карата за слику са две карте су дефинисане као:

```
decalX=int((imgW-cardW)/2)
decalY=int((imgH-cardH)/2)
```

Сада се у функцији `create2CardsScene()` прави нулта матрица за прву карту, димензија излазне слике, и чува у атрибуту класе `self.img1`, и на њеној централној позицији се уписује слика карте, а њено насумично померање је извршено функцијом `augment()`, а као аргументи јој се, поред слике карте, прослеђују и „кључне тачке” (keypoints) оквира око карте `cardKP`, дефинисаног на почетној позицији, и оба полиогона око знака и броја, као и трансформација коју је потребно извршити над картом, сачувана у променљивој `transform_1card`.

```
# imgaug keypoints of the bounding box of a whole card
cardKP = ia.KeypointsOnImage([
    ia.Keypoint(x=decalX,y=decalY),
    ia.Keypoint(x=decalX+cardW,y=decalY),
    ia.Keypoint(x=decalX+cardW,y=decalY+cardH),
    ia.Keypoint(x=decalX,y=decalY+cardH)
    ], shape=(imgH,imgW,3))

# imgaug transformation for one card in scenario with 2 cards
transform_1card = iaa.Sequential([
    iaa.Affine(scale=[0.65,1]),
    iaa.Affine(rotate=(-180,180)),
    iaa.Affine(translate_percent={"x":(-0.25,0.25),"y":(-0.25,0.25)}),
])
```

```
def augment(img, list_kps, seq, restart=True):
    # Make sequence deterministic
    while True:
        if restart:
            myseq=seq.to_deterministic()
        else:
            myseq=seq
        # Augment image, keypoints and bbs
        img_aug = myseq.augment_images([img])[0]
        list_kps_aug = [myseq.augment_keypoints([kp])[0]
                                                for kp in list_kps]
        list_bbs = [kps_to_BB(list_kps_aug[1]),kps_to_BB(list_kps_aug[2])]
        valid=True
        # Check the card bounding box stays inside the image
        for bb in list_bbs:
            if bb is None or int(round(bb.x2)) >= imgW or
                            int(round(bb.y2)) >= imgH or
                            int(bb.x1)<=0 or int(bb.y1)<=0:
                valid=False
                break
        if valid: break
        elif not restart:
            img_aug=None
            break

    return img_aug,list_kps_aug,list_bbs
```

У функцији `augment()` се дефинишу гранични оквири који окружују знак и број, позивом функције `kps_to_BB()` која прави гранични оквир на основу „кључних тачака” `imaug` библиотеке, тако што налази тачке са минималним и максималним `x` и `y` координатама и као излаз враћа гранични оквир одређен тим вредностима.

```
def kps_to_BB(kps):
    extend=3 # To make the bounding box a little bit bigger
    kpsx=[kp.x for kp in kps.keypoints]
    minx=max(0,int(min(kpsx)-extend))
    maxx=min(imgW,int(max(kpsx)+extend))
    kpsy=[kp.y for kp in kps.keypoints]
    miny=max(0,int(min(kpsy)-extend))
    maxy=min(imgH,int(max(kpsy)+extend))
    if minx==maxx or miny==maxy:
        return None
    else:
        return ia.BoundingBox(x1=minx,y1=miny,x2=maxx,y2=maxy)
```

Сада се у функцији `augment()` за сваки оквир врши провера да ли се налази унутар слике.

У функцији `create2CardsScene()` следи `while` петља унутар које се на почетку извршава функција `augment()`, али сада над другом картом, а њен излаз се прослеђује функцији `kps_to_polygon()`, која „кључне тачке” дефинише као координате.

```
def kps_to_polygon(kps):
    pts=[(kp.x,kp.y) for kp in kps]
    return Polygon(pts)
```

Даље унутар `create2CardsScene()` функције, врши се провера да ли је пресек граничних оквира око знака и броја у одређеним границама, јер је циљ да се избегне да друга карта прекрије знак и број прве карте, ако се то деси, улази се нову итерацију петље.

Уколико је услов задовољен, у листу оквира тј. атрибуту класе `self.listbba` се додаје инстанца класе `BBA`, при чијем прављењу се унутар конструктора у њене атрибуте уписују координате оквира око знака и броја, и назив карте.

```
class BBA:  # Bounding box + annotations
    def __init__(self,bb,classname):
        self.x1=int(round(bb.x1))
        self.y1=int(round(bb.y1))
        self.x2=int(round(bb.x2))
        self.y2=int(round(bb.y2))
        self.classname=classname
```

По изласку из `while` петље се листи оквира `self.listbba`, којој су додата два оквира око знака и броја прве карте, додају и оквири друге карте позивом `self.listbba.append(BBA(bb,class2))`.

На крају функције `create2CardsScene()` се спајају позадина и две карте у једну слику и коначна слика је уписана у атрибут класе `self.final`. На слици 4 је приказан пример слике са две карте.

Слика 4: Слика за тренирање која садржи две карте

Други део тренинг скупа, са три карте поређане као када се држе у руци, се извршава функцијом `create3CardsScene()`. Циљ је да се изврше две трансформације за две карте појединачно, а онда трећа трансформација све три карте као групе.

На почетку функције `create3CardsScene()` се извршавају позиви функције `hull_to_kps()` за полигоне око знака и броја две карте. Праве се нулте матрице за све три карте, димензија улазне слике, и уписују у атрибуте класе `self.img1`, `self.img2` и `self.img3`. Слике карата, сачуване као матрице у улазним променљивама `img1`, `img2` и `img3`, се уписују у наведене нулте матрице на почетним позцијама, тако да карте пре извршавања трансформација буду у центру слике, а координате почетне позиције карата за слику са три карте су дефинисане као:

```
decalX3=int(imgW/2)
decalY3=int(imgH/2-cardH)
```

Над `self.img3` и `self.img2` се позива функција `augment()` и прослеђују трансформације које желимо да извршимо над њима `trans_rot1` и `trans_rot2`, респективно. Уз то се функцији `augment()` прослеђују још и „кључне тачке” граничног оквира око карте `cardKP`, и изнад добијене „кључне тачке” позивањем функције `hull_to_kps()`.

```
trans_rot1 = iaa.Sequential([
    iaa.Affine(translate_px={"x": (10, 20)}),
    iaa.Affine(rotate=(22,30))
])
trans_rot2 = iaa.Sequential([
```

```
    iaa.Affine(translate_px={"x": (0, 5)}),
    iaa.Affine(rotate=(10,15))
])
transform_3cards = iaa.Sequential([
    iaa.Affine(translate_px={"x":decalX-decalX3,"y":decalY-decalY3}),
    iaa.Affine(scale=[0.65,1]),
    iaa.Affine(rotate=(-180,180)),
    iaa.Affine(translate_percent={"x":(-0.2,0.2),"y":(-0.2,0.2)})
])
```

Следи `while` петља у којој се извршава трансформација над све три карте, дефинисана у променљивој `transform_3cards`. Функција `augment()` се позива над `self.img1`, `self.img2` и `self.img3`.

Изван петље се уписују гранични оквири око знака и броја у атрибут класе `self.listbba`, а прослеђују се као инстанце класе `BBA`.

На крају функције `create3CardsScene()` се спајају позадина и три карте у једну слику и коначна слика је уписана у атрибут класе `self.final`. На слици 5 је приказан пример слике са три карте.

Слика 5: Слика за тренирање која садржи три карте

Наредни корак у прављену скупа за тренирање је да се, поред позива функција `create2CardsScene()` и `create3CardsScene()`, сачува и информација о позицији знака и броја на слици.

У следећој „ћелији” кода Jupyter свеске је дефинисан облик означавања података у XML језику, као и функција `create_voc_xml()` која чува вредности у већ одређеном

XML формату. Аргументи функције су излазна XML датотека, назив слике тј. путања и листа граничних оквира око знака и броја.

```
xml_body_1="""<annotation>
        <folder>FOLDER</folder>
        <filename>{FILENAME}</filename>
        <path>{PATH}</path>
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>{WIDTH}</width>
                <height>{HEIGHT}</height>
                <depth>3</depth>
        </size>
"""
xml_object=""" <object>
                <name>{CLASS}</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>{XMIN}</xmin>
                        <ymin>{YMIN}</ymin>
                        <xmax>{XMAX}</xmax>
                        <ymax>{YMAX}</ymax>
                </bndbox>
        </object>
"""
xml_body_2="""</annotation>
"""

def create_voc_xml(xml_file, img_file,listbba,display=False):
    with open(xml_file,"w") as f:
        f.write(xml_body_1.format(**{'FILENAME':os.path.basename(img_file),
                  'PATH':img_file,'WIDTH':imgW,'HEIGHT':imgH}))
        for bba in listbba:
            f.write(xml_object.format(**{'CLASS':bba.classname,
                'XMIN':bba.x1,'YMIN':bba.y1,'XMAX':bba.x2,'YMAX':bba.y2}))
        f.write(xml_body_2)
        if display: print("New xml",xml_file)
```

2.4 Превођење XML датотека у формат YOLO алгоритма

Поред Jupyter свеске `creating_playing_cards_dataset.ipynb` чији је садржај изнад описан, у GitHub репозиторијуму наведеном у литератури [1] се налази и Python датотека `convert_voc_yolo.py`. Ова датотека омогућава претварање података о објектима који се налазе на слици из дефинисаног XML формата у формат који одговара YOLO алгоритму, који ће бити примењен за тренирање.

Жељени формат је `<класа-објекта> <x> <y> <ширина> <висина>` [6]:

- `<класа-објекта>` - редни број класе, `0` до `(бројКласа-1)`
- `<x> <y> <ширина> <висина>` - рационалне вредности релативне у односу на висину и ширину слике, могу имати узети вредност из `(0.0,1.0]`
- на пример: `<x> = <апсолутно_x> / <ширина_слике>` или `<висина> = <апсолутна_висина> / <висина_слике>`
- `<x>` и `<y>` - су центар правоугаоника (не горњи-леви ћошак)

У датотеци `convert_voc_yolo.py` се пролази кроз XML датотеку и налазе потребни подаци, а кључна функција за превођење у YOLO формат је `convert()`. У аргументу `size` се налазе димензије слике, а у `box` су садржане `x` и `y` координате ћошкова граничног оквира око знака и броја карте.

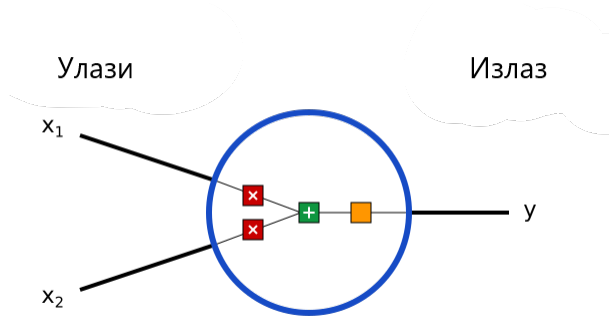
С обзиром да `<x>` и `<y>` у жељеном формату представљају центар оквира тј. правоугаоника, те вредности се добијају аритметичком средином улазних. Дељењем са ширином тј. висином слике се добијају релативне вредности.

```
def convert(size, box):
    dw = 1./size[0]
    dh = 1./size[1]
    x = (box[0] + box[1])/2.0
    y = (box[2] + box[3])/2.0
    w = box[1] - box[0]
    h = box[3] - box[2]
    x = x*dw
    w = w*dw
    y = y*dh
    h = h*dh
    return (x,y,w,h)
```

3 Вештачке неуронске мреже - теорија

3.1 Увод у вештачке неуронске мреже

Неурон је основна јединица вештачке неуронске мреже и има своје улазе, над којима извршава рачунске операције и добијену вредност уписује као излаз [7]. Овако изгледа неурон са два улаза:

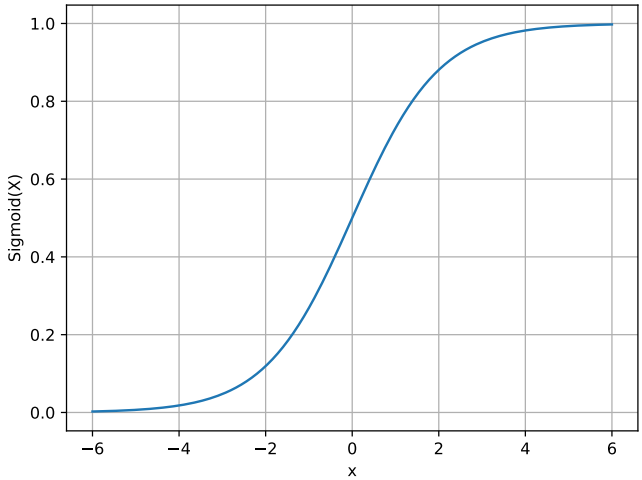


Слика 6: Неурон

Одигравају се три ствари. Прво се улази x_1 и x_2 множе тежинама веза w_1 и w_2, па се сви ови производи сабирају и том збиру додаје и праг активације b. На крају се ова вредноста проследи активационој функцији и добија се излазна вредност.

$$y = f(x_1 * w_1 + x_2 * w_2 + b) \tag{1}$$

Функција која се често примењује као активациона је Сигмоидна функција, дефинисана као $S(x) = \frac{1}{1+e^{-x}}$.



Слика 7: Сигмоидна функција

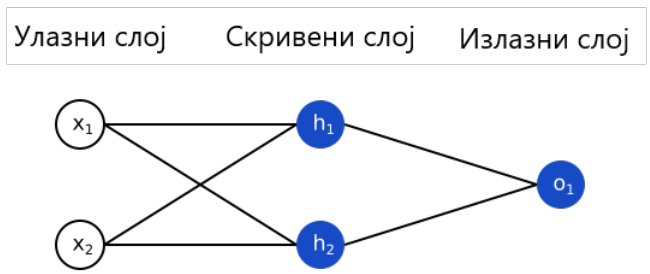
Ако би тежине веза имале вредности $w_1 = 0$ и $w_2 = 1$ тј. $w = [0, 1]$ (ако се представе као вектор), праг активације $b = 4$, а улази неурона $x_1 = 2$ и $x_2 = 3$ ($x = [2, 3]$), тада се применом изнад дефинисане формуле долази до вредности излаза:

$$\begin{aligned}(w \cdot x + b) &= ((w_1 * x_1) + (w_2 * x_2)) + b \\ &= 0 * 2 + 1 * 3 + 4 \\ &= 7\end{aligned} \tag{2}$$

$$y = f(w \cdot x + b) = f(7) = 0.999 \tag{3}$$

Када улази неурона износе $x = [2, 3]$, излазна вредност је 0.999. Овакав процес прослеђивања улазних вредност напред назива се **пропагација унапред**.

Неуронска мрежа је скуп међусобно повезаних неурона. На слици 8 је приказан пример вештачке неуронске мреже:



Слика 8: Вештачка неуронска мрежа

Ова мрежа има два улаза, скривени слој са два неурона (h_1 и h_2) и излазни слој који се састоји од једног неурона o_1. **Скривени слој** је било који слој између улазног и излазног слоја и могуће је да се вештачка неуронска мрежа састоји од много скривених слојева.

Претпоставимо да сви неурони имају исте тежине веза $w_1 = 0$ и $w_2 = 1$ тј. $w = [0, 1]$, исти праг активације $b = 0$ и сви неурони као активациону функцију користе Сигмоидну функцију; тада се следећим рачуном добија излазна вредност, ако се на улаз проследи вредност $x = [2, 3]$:

$$\begin{aligned}h_1 = h_2 &= f(w \cdot x + b) \\ &= f((0 * 2) + (1 * 3) + 0) \\ &= f(3) \\ &= 0.9526\end{aligned} \tag{4}$$

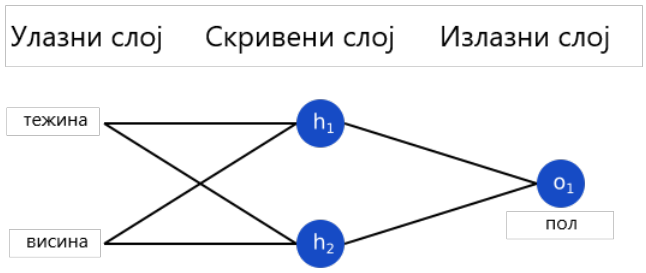
$$\begin{aligned}o_1 &= f(w \cdot [h_1, h_2] + b) \\ &= f((0 * h_1) + (1 * h_2) + 0) \\ &= f(0.9526) \\ &= 0.7216\end{aligned} \tag{5}$$

Вештачка неуронска мрежа може да има произвољан број слојева и неурона у њима. Користимо следеће податке за пример тренирања вештачке неуронске мреже:

Име	Тежина (lb)	Висина (in)	Пол
Наташа	133	65	Ж
Бајага	160	72	М
Ђоковић	152	70	М
Ивана	120	60	Ж

Табела 1: Пример тренинг скупа података

Сада следи тренирање мреже која треба да предвиди пол особе, а као улазне параметре има тежину и висину:



Слика 9: Вештачка неуронска мрежа која треба да се тренира

У наредној табели су *Мушкарци* означени бројем 0, *Жене* бројем 1, а остале вредности су подједнако смањене, да би израчунавање било једноставније:

Име	Тежина (минус 135)	Висина (минус 66)	Пол
Наташа	-2	-1	1
Бајага	25	6	0
Ђоковић	17	4	0
Ивана	-15	-6	1

Табела 2: Умањене вредности (припрема тренинг скупа)

Тренирање мреже је побољшање тачности предвиђања мреже. Да би се тачност повећала потребно је имати некакву мерило, па се дефинишу термини **функција губитка** L (Loss function) и **функција трошка** J (Cost function). Функција губитка се односи на један пример тренинг скупа, док је функција трошка средња вредност извршених функција губитка над свим примерима скупа, уз додатну „регуларизацију” [8].

Нека је на пример *средња квадратна грешка* (*mean square error*) функција трошка:

$$J = MSE = \frac{1}{m}\sum_{i=1}^{m}(y_{true} - y_{pred})^2 \tag{6}$$

Где је:

- m је број примера тј. 4 (Наташа, Ђоковић, Бајага, Ивана)
- y је променљива коју треба да предвидимо тј. Пол
- y_{true} је „тачан одговор”. На пример, y_{true} за Наташу би био 1 (*Жена*).
- y_{pred} је вредност коју смо предвидели, оно што је излаз неуронске мреже

Предвиђање је боље, уколико је функција трошка мања, па је **тренирање мреже поступак промене вредности тежине веза и прагова активације са циљем да се одреди што нижа функција трошка**.

Нека мрежа увек као излаз даје 0 тј. увек предвиђа да је особа *Мушкарац*; онда функција трошка тј. *средња квадратна грешка* износи:

Име	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Наташа	1	0	1
Бајага	0	0	0
Ђоковић	0	0	0
Ивана	1	0	1

Табела 3: Вредности функције губитка

$$MSE = \frac{1}{4}(1 + 0 + 0 + 1) = 0.5 \tag{7}$$

Нека се сада скуп састоји само из једног примера тј. особе:

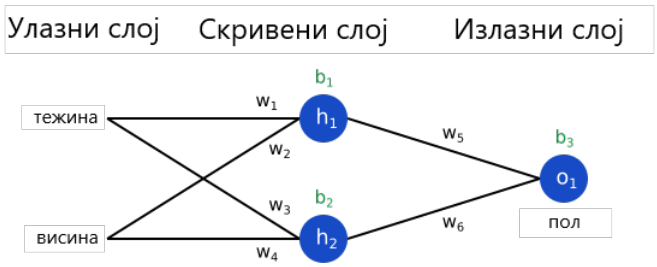
Име	Тежина (минус 135)	Висина (минус 66)	Пол
Наташа	-2	-1	1

Табела 4: Скуп који се састоји од једне *особе*

Средња квадратна грешка сада износи:

$$\begin{aligned} MSE &= \frac{1}{1}\sum_{i=1}^{1}(y_{true} - y_{pred})^2 \\ &= (y_{true} - y_{pred})^2 \\ &= (1 - y_{pred})^2 \end{aligned} \tag{8}$$

На слици 10 су означене ознаке тежине веза и прагови активације:



Слика 10: Тежине веза и прагови активације у вештачкој неуронској мрежи

Функција трошка може да се представи као функција са више променљивих:

$$J(w_1, w_2, w_3, w_4, w_5, w_6, b_1, w_2, w_3) \tag{9}$$

Промена вредности w_1, мења вредност J, па је потребно израчунати парцијални извод. Парцијални извод је представљен као:

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1} \tag{10}$$

Изнад је већ добијена вредност $J = (1 - y_{pred})^2$, па је могуће израчунати $\partial J / \partial y_{pred}$:

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = -2(1 - y_{pred}) \tag{11}$$

Потребно је одредити $\partial y_{pred} / \partial w_1$ и као у изнад наведеном примеру пропагације унапред, h_1, h_2 и o_1 су излази одговарајућих неурона. Тада је

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3), \tag{12}$$

где је f Сигмоидна функција.

Пошто w_1 утиче на h_1 (не h_2), може се написати

$$\begin{aligned} \frac{\partial y_{pred}}{\partial w_1} &= \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1} \\ \frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \end{aligned} \tag{13}$$

Урађено је исто и за $\partial h_1 / \partial w_1$:

$$\begin{aligned} h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\ \frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1), \end{aligned} \tag{14}$$

где је x_1 улаз неурона који представља тежину особе, а x_2 висину. Извод Сигмоидне функције је:

$$\begin{aligned} f(x) &= \frac{1}{1 + e^{-x}} \\ f'(x) &= \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x)) \end{aligned} \tag{15}$$

Сада је парцијални извод сведен на чланове које је могуће израчунати:

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1} \tag{16}$$

Поступак рачунања парцијалних извода идући од излаза ка улазу вештачке неуронске мреже назива се **пропагација уназад**.

Над скупом са једном *особом*, приказаним у табели 4, ће бити израчунати парцијални изводи.

Иницијализоване су вредности тежине веза на 1, а прагови активације на 0. Ако се изврши пропагација унапред, добија се:

$$\begin{aligned}
h_1 &= f(w_1x_1 + w_2x_2 + b_1) \\
&= f(-2 + -1 + 0) \\
&= 0.0474 \\
h_2 &= f(w_3x_1 + w_4x_2 + b_2) = 0.0474 \\
o_1 &= f(w_5h_1 + w_6h_2 + b_3) \\
&= f(0.0474 + 0.0474 + 0) \\
&= 0.524
\end{aligned} \tag{17}$$

Излаз мреже има вредност $y_{pred} = 0.524$, а то је вредност која не указује јасно да ли је *особа Мушкарац* (0) или *Жена* (1). Сада ће бити израчунато [1] $\partial J/\partial w_1$:

$$\begin{aligned}
\frac{\partial J}{\partial w_1} &= \frac{\partial J}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1} \\
\frac{\partial J}{\partial y_{pred}} &= -2(1 - y_{pred}) \\
&= -2(1 - 0.524) \\
&= -0.952 \\
\frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5h_1 + w_6h_2 + b_3) \\
&= 1 * f'(0.0474 + 0.0474 + 0) \\
&= f(0.0948) * (1 - f(0.0948)) \\
&= 0.249 \\
\frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1x_1 + w_2x_2 + b_1) \\
&= -2 * f'(-2 + -1 + 0) \\
&= -2 * f(-3) * (1 - f(-3)) \\
&= -0.0904 \\
\frac{\partial J}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\
&= 0.0214
\end{aligned} \tag{18}$$

[1] Извод $f'(x) = f(x) * (1 - f(x))$ за Сигмоидну функцију је израчунат у једначини (15).

Може се закључити да би се вредност функције трошка J врло мало повећала, уколико би w_1 било повећано.

Следи тренирање мреже и биће коришћен алгоритам *Стохастички градијентни спуст* којим се одређује како мењати вредности тежина веза и прагова активације, да би се минимизовала вредност функције трошка J.

Представљен је следећом једначином:

$$w_1 \leftarrow w_1 - \eta \frac{\partial J}{\partial w_1}, \tag{19}$$

где је η константа и представља **дужину корака при учењу** (**learning rate**), њиме се контролише колико брзо учимо. Одузима се $\eta \frac{\partial J}{\partial w_1}$ од w_1:

- ако је $\partial J / \partial w_1$ позитивно, w_1 ће се смањити, па ће се и J смањити
- ако је $\partial J / \partial w_1$ негативно, w_1 ће се повећати, па ће се и J смањити

Ако се ово уради над свим тежинама веза и праговима активације (у једначини (19) се b_1 замени уместо w_1), функција трошка ће се смањивати тј. неуронска мрежа ће се побољшавати.

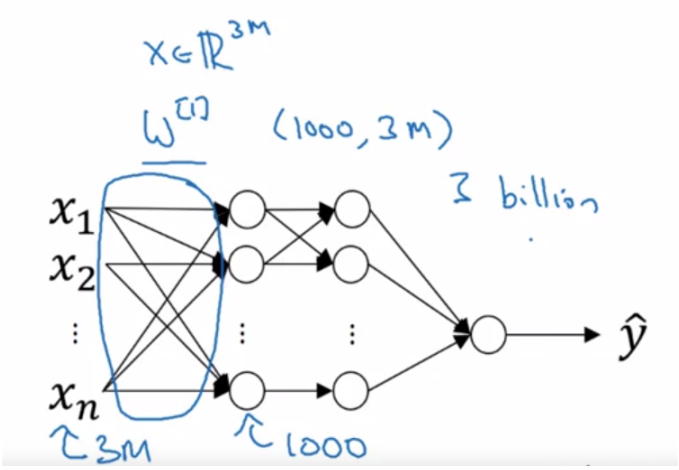
3.2 Конволуцијске неуронске мреже

Рачунарски вид (Computer vision) је грана која се изузетно брзо развија. Примена ове области је веома широка: детекција пешака и других аутомобила током аутономне вожње, препознавање лица примењено на откључавање телефона или врата, препознавање врсте хране итд. [9].

Један од проблема у овој области је да подаци (слике) на улазу могу да буду велики. На пример, над сликом 11 је потребно извршити класфикацију да ли се на њој налази мачка или не, а на излазу се добија вредност 0 или 1.

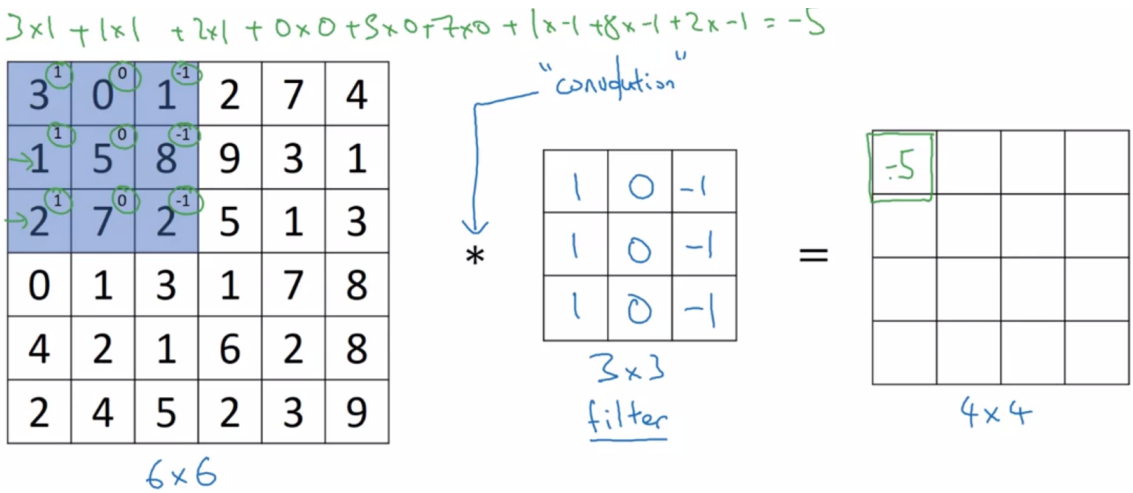
Слика 11: Слика великих димензија на улазу у мрежу

Уколико је слика 11 димензија $1000 \times 1000 \times 3$ (последња димензија 3 односи на три основне боје које се налазе у пикселу: црвену, зелену и плаву), то значи да је величина улазних података 3000000. Слика 12 приказује неуронску мрежу којој је прослеђена слика наведених димензија, па ако се претпостави да први слој има 1000 неурона, то значи да би само матрица тежина веза између прва два слоја имала димензије (1000,3000000), а тада израчунавање постаје неизводљиво.



Слика 12: Превелика величина података на улазу у мрежу

Операција **конволуције** је једна од основа за прављење конволуцијске неуронске мреже. Извршавање операције је приказано следећом сликом и на њој се види да се на улазу налази улазна матрица димензија 6×6 и над њом се изршава конволуција применом 3×3 филтера, да би се на излазу нашла матрица димензија 4×4.



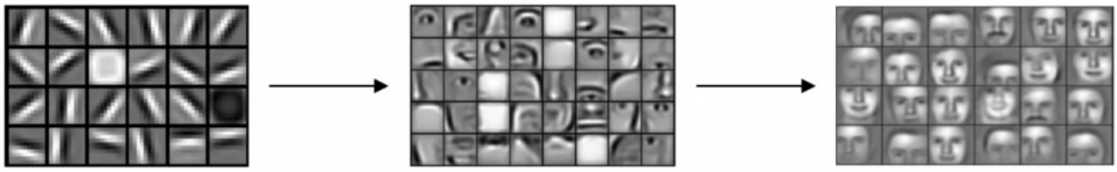
Слика 13: Операција конволуције

Операција се извршава тако што је вредност поља (1,1) у излазној матрици, једнак збиру производа чланова исечка (димензија филтера), улазне матрице и чланова филтера. Исечак (димензија филтера) се помера једно поље у страну (односно у следећи ред) и извршава се исти поступак.

Кроз пример *детектовања ивица* ће бити објашњен рад конволуцијске неуронске мреже [10].

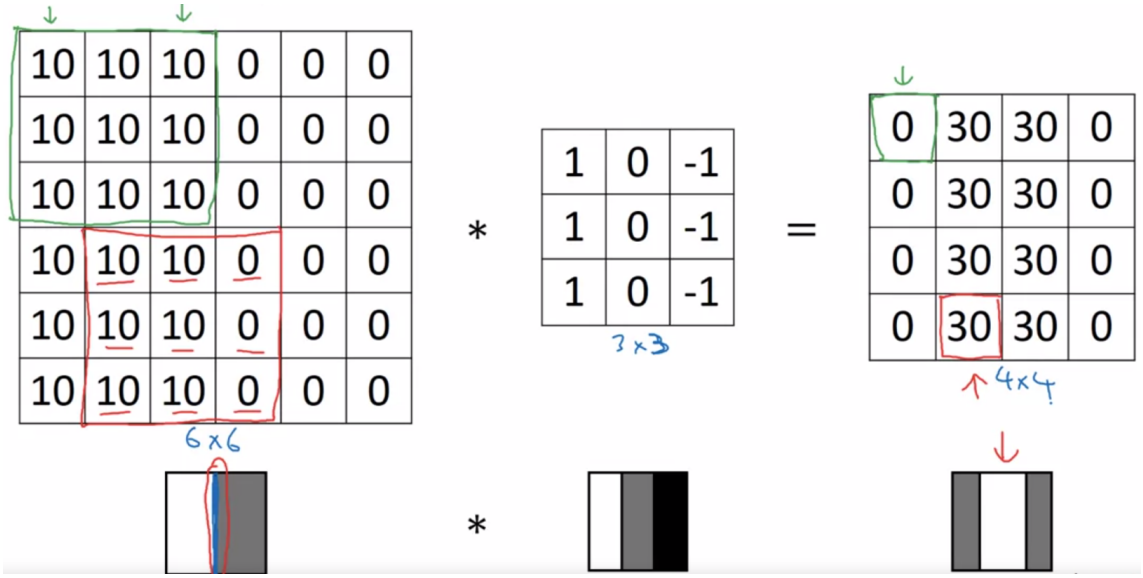
Применом много сложенијих неуронских мрежа у односу на оне које су приказане у претходном потпоглављу, могуће је решити проблем детекције лица. Први слојеви би детектовали *прелазе боја* (*ивице*), средњи слојеви налазе делове лица (очи, нос ...), док би виши слојеви на основу тих информација налазили лица.

Сада ће бити објашњено решавање првог случаја детекције ивица са слике 14, применом конволуцијских неуронских мрежа.



Слика 14: Поступак детекције лица

Нека је задатак да се детектују *вертикалне ивице*, а на улазу се налази једноставна матрица. Слика 15 приказује извршену операцију конволуције. Веће вредности у матрицама се односе на јаче осветљење пиксела, па се на излазу види да је у средини светлије, а са стране тамније, што се чита као вертикални прелаз **светло-у-тамно**.



Слика 15: Вертикална ивица, прелаз светло-у-тамно

Уколико би се на улазу уместо матрице где је светлија лева половина, нашла матрица која представља слику ивице где је десна страна светлија, излазна слика (после конволуције) би имала тамнији део у средини, а на крајевима би била светлија.

Поента је да је извршавањем конволуције коришћењем различитих филтера могуће детектовати различите ивице.

С обзиром да постоји много различитих ивица, а њихова детекција се врши применом различитих филтера тј. вредности поља матрице филтера су различите, могуће их је посматрати као параметре које је потребно проследити мрежи на

тренирање [11].

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} \tag{20}$$

При извршавању конволуције је могуће да исечак улазне матрице много пута покрије поља која се налазе у средини матрице, а само једном „посети” поља на ивици.

Уводи се **допуњавање матрице** (**padding**), а то је поступак проширивања матрице, чиме се добија да исечак, који се помера за један корак, посети поља на ивици више пута него пре допуне.

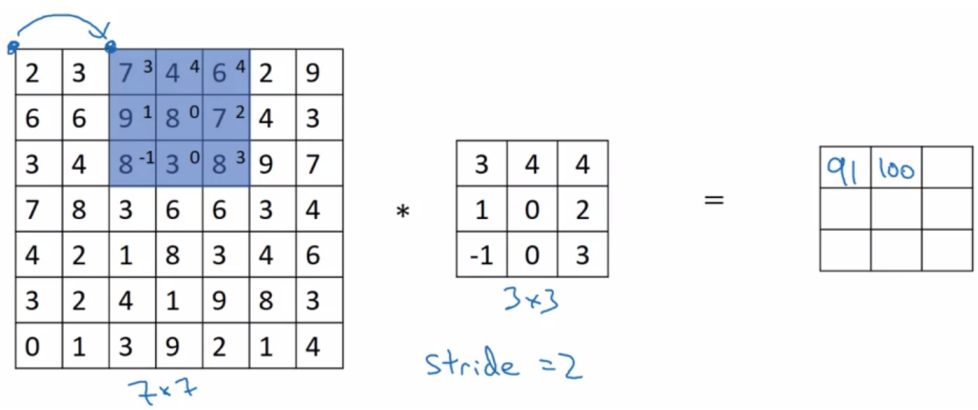
Овом техником се решавају два проблема и постиже се:

- да се на излазу нађе мање информација које се односе на средину матрице (слике)
- чување информација из поља са ивице слике

Поља којим се проширује матрица имају вредност 0, па не утичу на резултат.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & & & & & & 0 \\ 0 & & & & & & 0 \\ 0 & & & & & & 0 \\ 0 & & & & & & 0 \\ 0 & & & & & & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{21}$$

Још једна техника, којом се добија излазна матрица мањих димензија, је **Конволуција са већим кораком** (**Strided convolution**) и разликује се у *кораку* (*stride*) који исечак прави кроз улазну матрицу.



Слика 16: Конволуција са већим кораком исечка

Уколико се на улазу нађе слика димензија $n \times n$ и треба извршити конволуцију коришћењем филтера димензија $f \times f$, где је *корак* (*stride*) s, а *допуна* (*padding*) p; тада

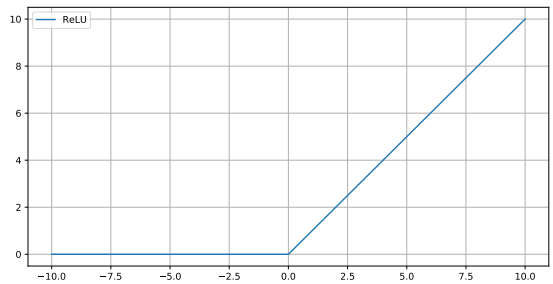
је димензија излазне матрице:

$$\left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor$$

Слика 18 приказује један слој конволуцијске неуронске мреже. На излазу се налази активациона функција ReLU, дефинисана као:

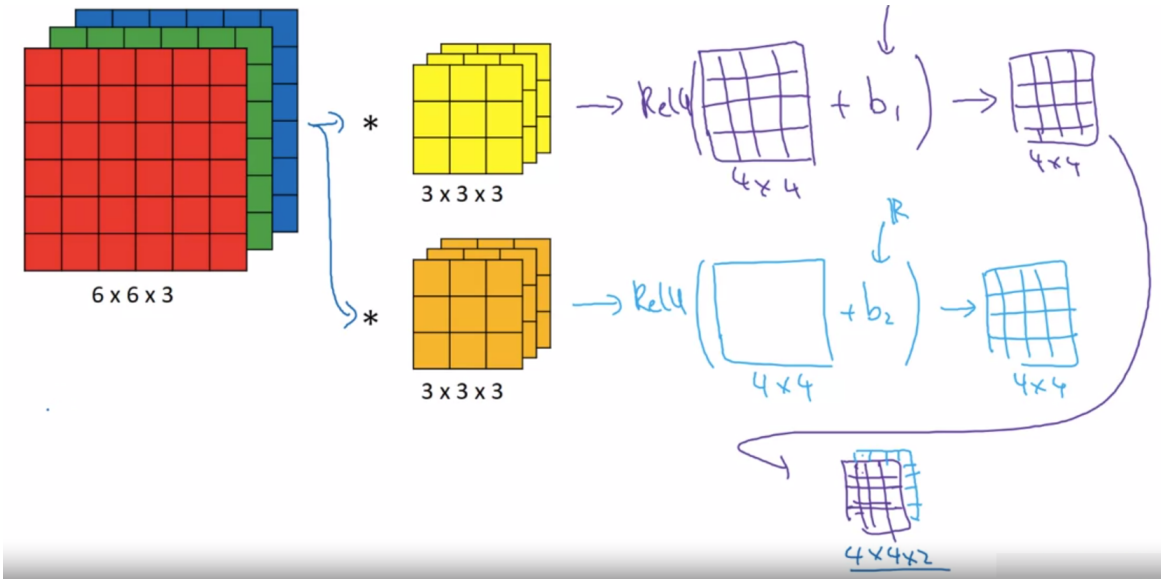
$$f(x) = max(0, x), \tag{22}$$

а приказана је на наредној слици:



Слика 17: ReLU функција

У примеру се на улазу налазе три матрице и показују вредности црвене, зелене и плаве боје у пикселима. Извршавају се две конволуције применом два разчичита филтера, а ReLU активационој функцији се прослеђује резултат конволуције увећан за праг активације. На излазу се налази матрица димензија $4 \times 4 \times 2$.

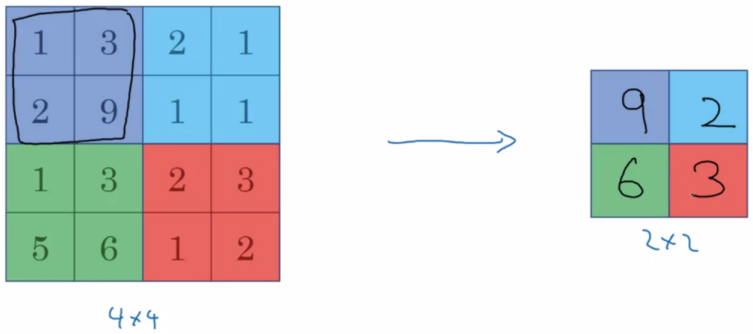


Слика 18: Пример слоја конволуцијске неуронске мреже

Оно што је битно да се примети је да колико год да је слика велика, број параметара за тренирање је дефинисан димензијама филтера, јер су то вредности које се прилагођавају [12].

На пример, ако постоји 10 филтера димензија $3 \times 3 \times 3$ (27 параметара) и праг активације b за сваки, значи да постоји укупно 280 параметара за тренирање, *без обзира на димензије слике на улазу*!

Врло често је матрица на излазу веома велика и можда ју је потребно смањити, а да притом главне *одлике* буду сачуване. Ово се постиже **Слојевима удруживања** (**Pooling layers**) и слика 19 приказује један тип који се назива *Удруживање максимума* (*Max pooling*). Процедура је слична као и са конволуцијом, јер се дефинише димензија филтера тј. исечка f и величина корака (stride) s.



Слика 19: Max pooling

Удруживање максимума се разликује од конволуције по томе што сада нема параметара које треба тренирати, него је операција фиксна. **Максимална вредност у исечку се уписује у излазну матрицу**.

Идеја је да уколико постоји већа вредност која указује на неки прелаз тј. *одлику*, она ће бити сачувана и у излазној матрици (као број 9 на слици), а уколико су вредности приближне, уписана вредност је само мало већа од осталих (број 2 са слике је био окружен 1, није велика разлика ...).

Две **предности коришћења конволуцијских неуронских мрежа** су:

- Дељење параметара: филтер који детектује одређене *ивице* у једном делу слике, врло вероватно ће подједнако добро радити и у другом делу слике. Ако би се на улазу налазила слика димензија $32 \times 32 \times 3$ и било примењено 6 5×5 филтера, то би значило да први слој има 3072 поља, а други 4704.

  Када би мрежа била дефинисана *потпуно повезаним слојевима* (*fully connected layers - FC*), димензије тежина веза између ова два слоја би биле $3072 \times 4704 \approx 14m$, што је превише.

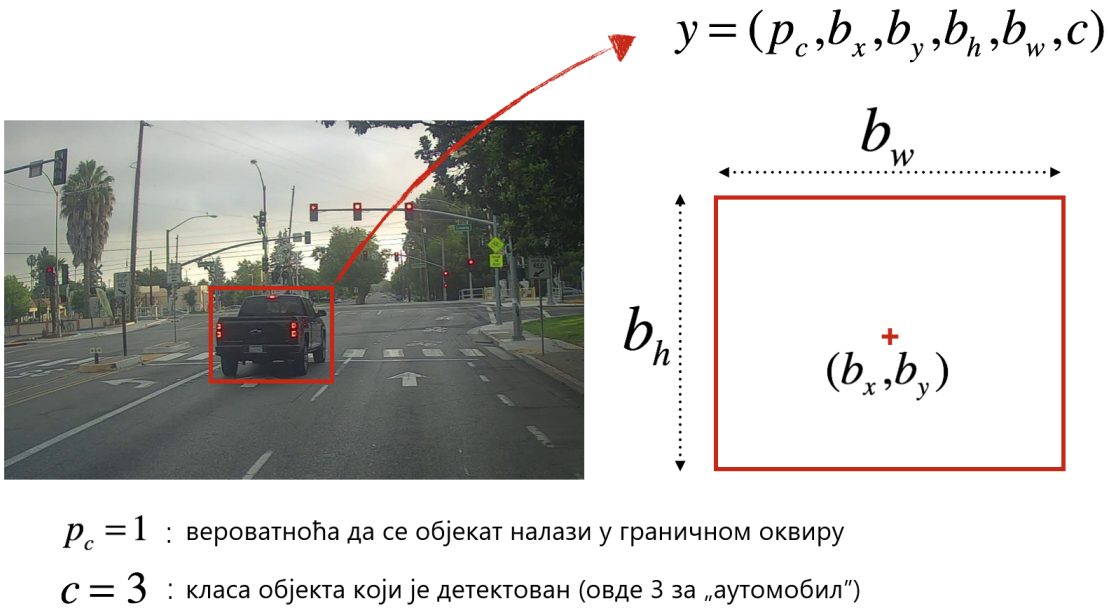
- Ређе зависности (Sparsity of connections): У сваком слоју, ниједна излазна вредност не зависи од много улазних. Свако поље матрице на излазу је израчунато из малог исечка слике са улаза и не зависи од осталих.

Конволуцијске неуронске мреже, овим одликама, омогућавају тренирање над мањим скуповима и теже долази до *претренирања* (*overfitting*).

3.3 YOLO алгоритам

Треба разликовати **класификацију** слике тј. одређивање класе објекта на слици и **локализацију** објекта тј. његово означавање (уоквиравање). Такође, сложенији проблем би била **детекција** објеката, када се на слици налази више објеката које треба класификовати и означити (локализовати) [13].

На слици 20 је приказан пример детекције аутомобила. Вектор у коме се чувају подаци о детекцији садржи: p_c - вероватноћу да је детектован објекат одређене класе; b_x, b_y, b_h, b_w - бројеви којим се дефинишу димензија и позиција граничног оквира; c - класа објекта који је детектован.



Слика 20: Детекција и дефинисање граничног оквира око објекта

Параметар c може да буде или целобројна вредност која представља класу или вектор димензија броја класа које детектујемо, где члан вектора који представља класу има вредност 1, а остали имају вредност нула.

Ако би требало да детектујемо три класе:

1. пешак
2. аутомобил
3. мотоцикл

вектор којим се представља детекција *аутомобила*, где c је представљено као вектор

тј. постоји посебан члан (бит) за сваку класу, изгледао би као:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{23}$$

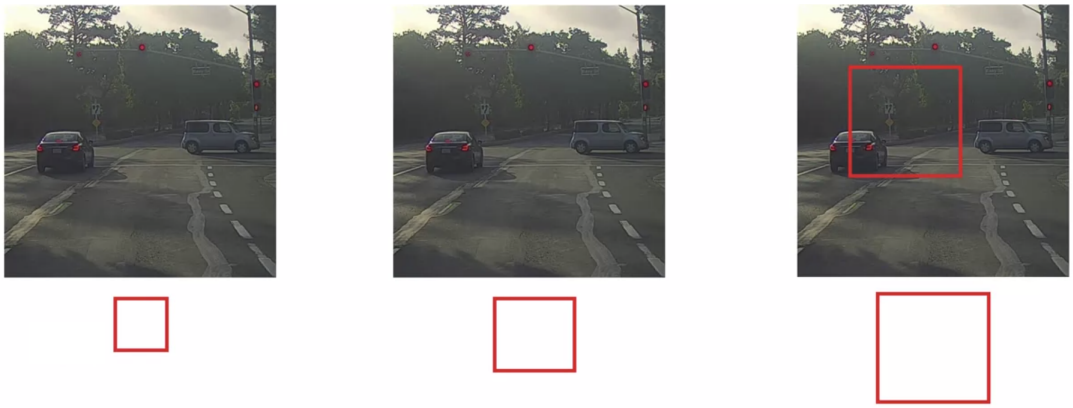
У случају да у посматраној слици нема објеката, вероватноћа p_c ће бити 0, па су остали чланови вектора небитни:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} \tag{24}$$

На слици 21 је приказан пример *Детекције објеката клизним прозорима* [14]. Сваки *прозор* се проследи конволуцијској мрежи, која даје резултат да ли има објекта у прослеђеном исечку слике или не. *Прозор* треба да „проклиза" кроз целу слику и овај поступак се може поновити и са већим *прозором*.

Идеја је да ће се сигурно унутар неког *прозора* наћи објекат тј. да ће резултат прослеђивања исечка слике конволуцијској мрежи бити 1.

Проблем оваквог приступа је превелико време извршавања. *Детектор објеката клизним прозорима* се може направити коришћењем конволуцијских мрежа и таква имплементација је много ефикаснија.

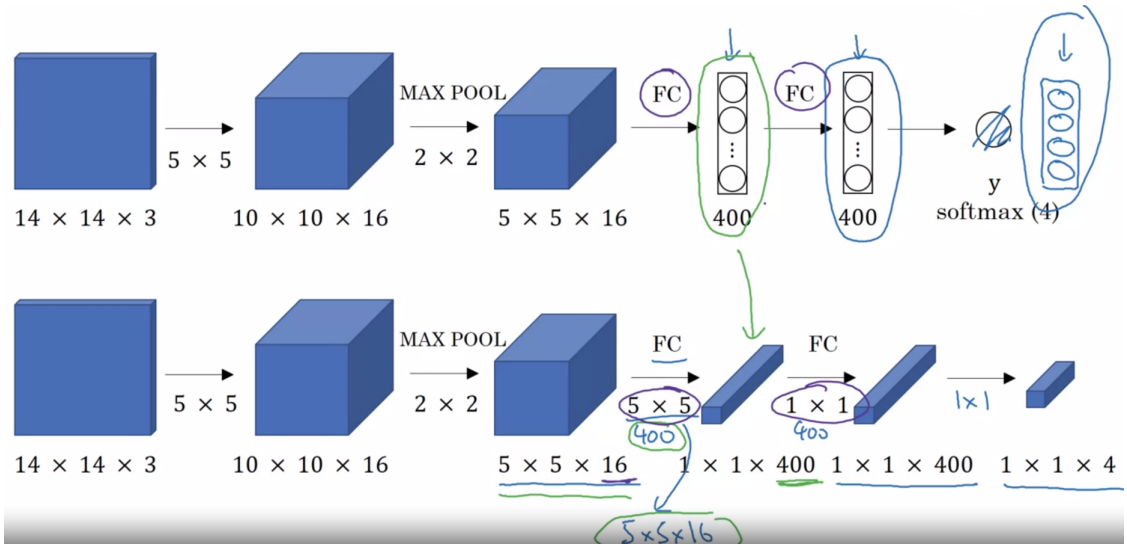


Слика 21: Детекција објеката клизним прозорима

Предност која се добија конволуцијском имплеметацијом је то што је у једној итерацији могуће добити више вредности тј. резултата за више *прозора*. Слика 22 приказује имплементацију *детекције објеката клизним прозорима* применом конволуцијских мрежа.

Прва мрежа на крају садржи два *потпуно повезана слоја* (*fully connected layers*). Ова два слоја праве проблем тј. успоравају детекцију, јер је потребно посебно извршавање за сваки *прозор* тј. исечак слике. На крају се налази Softmax слој, чији сваки члан представља вероватноћу да је детектована класа којој тај члан одговара.

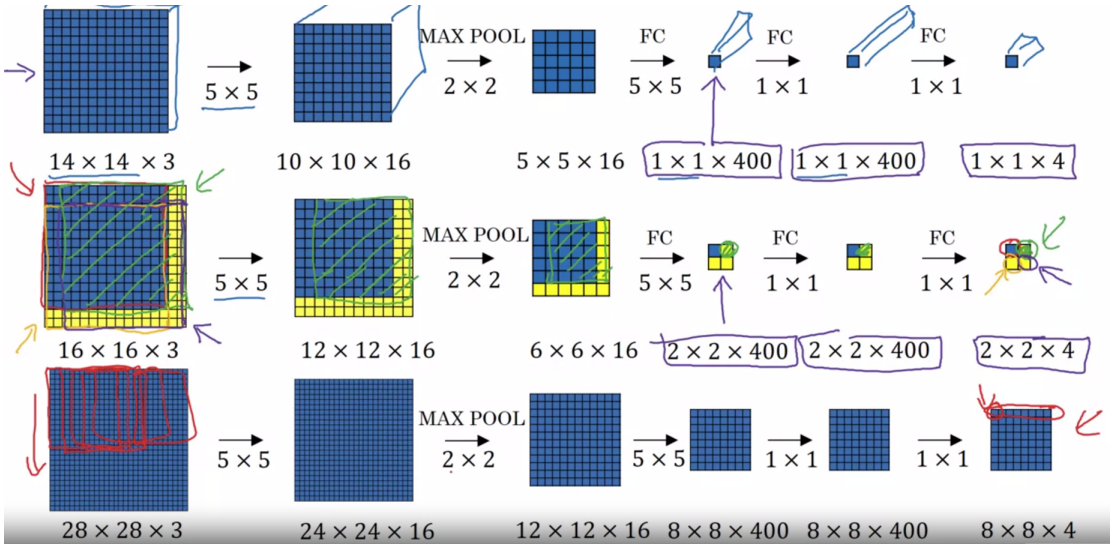
У другом случају се примењује филтер $5 \times 5 \times 16 \times 400$, па су излазне димензије $1 \times 1 \times 400$, а у наредном кораку се врши конволуција коришћењем 400 филтера димензија 1×1. На крају се примењује 1×1 филтер чиме се добија последњи слој димензија $1 \times 1 \times 4$.



Слика 22: Имплементација клизних прозора применом конволуцијских мрежа

Слика 23 приказује случај када на улазу нађе слика већих димензија од 14×14, као 16×16 или 28×28.

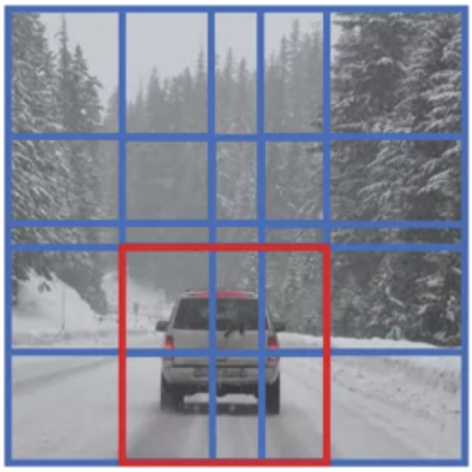
Сада се види предност у конволуцијској имплементацији *детекције објеката клизним прозорима*, где нпр. зелени (горњи-десни) 14×14 исечак улазне слике, као резултат даје има зелени означен део излаза. Поента је да су једним извршавањем добијени разултати за све исечке слике, исто је и за последњи пример када се на улазу нашла слика димензија 28×28.



Слика 23: Детекција над сликама већих димензија

Наредна слика показује све позиције квадрата којим се праве исечци слике унутар којих је потребно извршити детекцију. Као што је горе објашњено вероватноће да се унутар квадрата налазе тражени објекти (у овом случају аутомобил) се добијају истовремено за све квадрате (исечке слике).

Конволуцијска имплементација је много ефикаснија него када се посебно за сваки део слике врши тестирање.

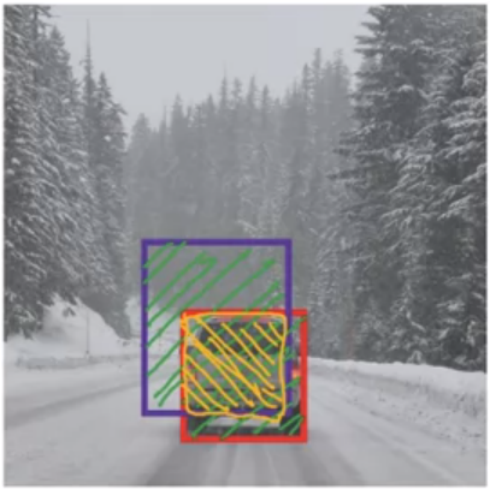


Слика 24: Истовремено извршавање детекција

Сада је потребно изабрати најбољи гранични оквир око објекта. Оквир припада оном исечку слике у коме се налази центар оквира. Дефинисан је параметар **ПпУ - Пресек преко Уније (IoU - Intersection over Union)**, којим је могуће установити преклапања два оквира. Представља однос површина пресека два оквира и њихове уније.

$$IoU = \frac{\cap}{\cup} \tag{25}$$

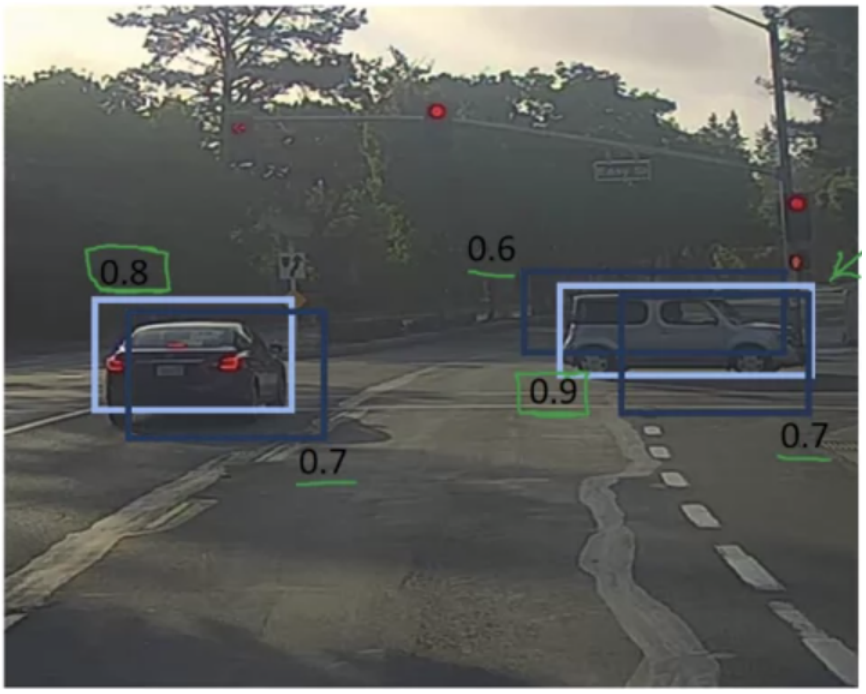
Слика 25 приказује преклапање два оквира, где је жутом бојом означен пресек, а зеленом унија, као у једначини (25). Дефинише се и задовољавајући праг, па ако је $IoU \geq 0.5$ нађени оквир се прихвата; наравно, овај праг је могуће повећати.



Слика 25: Преклапање два гранична оквира

Слика 26 приказује случај када се исти објекат нађе унутар више граничних оквира, али такође сваки са различитом сигурношћу указује да је тај објекат из одређене класе, у овом случају су различите вероватноће да је нађен аутомобил.

Узимају се у обзир оквири са највећом вероватноћом, а одбацују остали оквири који имају велико преклапање тј. IoU дефинисан у једначини (25).



Слика 26: Више граничних оквира око истог објекта

Овај поступак се назива **Не-Масимално потискивање (Non-Max suppression)** и одбацују се тј. *потискују* оквири који немају максималну вероватноћу детекције.

Нека је сада резултат израчунавања тј. провере да ли се објекат налази унутар прослеђеног исечка сличан као у приказаном вектору у једначини (23), али сада нека су изостављене класе, па је претпоставка да се врши претрага само за аутомобил и излаз је:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix} \tag{26}$$

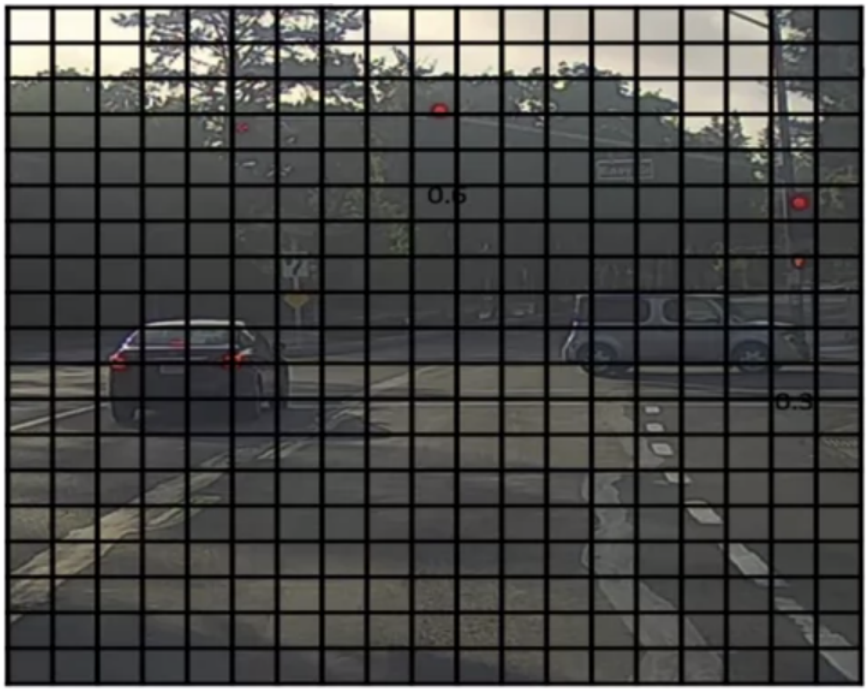
Претрага се врши над сликом 27, која је подељена *решетком* димензија 19×19. Поступак извршавања детекције је одређен псеудо-кодом:

1) Одбацити све оквире где је вероватноћа $p_c \leq 0.6$

2) Док год има непроверених оквира:

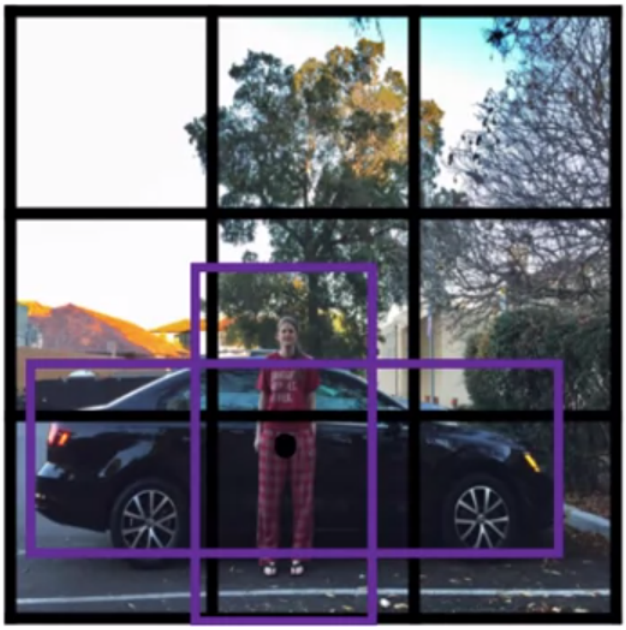
   (i) Изабрати оквир са највећом вероватноћом p_c

   (ii) Одбацити све преостале оквире где је $IoU \geq 0.5$ у односу на оквир из претходног корака



Слика 27: Слика подељена *решетком* димензија 19×19

Један од проблема детекције, који се може уочити до сада, је да се пољу *решетке* може доделити само један објекат [16]. Слика 28 приказује случај када центри граничних оквира *пешака* и *аутомобила* припадају истом пољу.



Слика 28: Класни оквири у случају више објеката додељеним једном пољу

Овакав проблем се решава дефинисањем граничних оквира који одговарају одређеној класи (**Anchor boxes**) и у овом случају постоје два, један за *пешака* и један за *аутомобил*.

До сада смо сматрали да свако поље решетке као излаз даје вектор као у једначини (23) тј. до сада је резултат израчунавања био $3 \times 3 \times 8$, где је сваком пољу био додељен вектор са 8 чланова који дефинишу детекцију.

Сада је излаз $3 \times 3 \times 16$, а вектор је дуплиран и првих 8 чланова се односи на

детекцију која одговара првом класном оквиру, а других 8 другом.

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ \dots \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 1 \\ 0 \\ 0 \\ \dots \\ 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{27}$$

Детектовани објекат се додељује пару `(пољеРешетке,класниОквир)` `(gridCell,anchorBox)`, а објекат, са слике над којом се врши тренирање, се додељује оном класном оквиру који има највеће преклапање са „правим” оквиром тј. највећи IoU (једначина (25)).

Уколико се на слици налази само аутомобил, тада ће прва половина вектора, која се односи на *пешака*, имати исте вредности као и раније у случају неналажења објекта, као у једначини (24).

Овако дефинисана примена класних оквира (Anchor boxes) не решава случај када се у пољу решетке налазе три (или више) објеката, али с обзором да YOLO алгоритам примењује 19×19 решетку над сликом, мале су шансе да ће се већи број објеката тј. центри граничних оквира око њих наћи у истом пољу решетке. Такође, могуће је да истом пољу припадају два објекта исте класе, па су им и класни оквири (Anchor boxes) исти. Одређивање класних оквира се може вршити произвољно (ручно) или применом K-means алгоритама, којим се групишу слични објекти.

YOLO (**You Only Look Once**) **алгоритам** функционише применом наведених техника. Нека се поново врши детекција *пешака* и *аутомобила*; а дефинисана су два класна оквира (Anchor boxes). Уопштено алгоритам функционише на следећи начин:

- за сваку *ћелију решетке* је потребно вратити 2 гранична оквира (слика 29, десно)
- занемарити оквире са малом вероватноћом детекције (слика 30, лево)
- за сваку класу је потребно извршити Не-Масимално потискивање (Non-Max suppression) (слика 30, десно)

Слика 29: Тест слика; Оквири за сваку *ћелију*

Слика 30: Оквири са великом вероватноћом поготка; Извршен Non-Max suppression

Резултат алгоритма би требало да буде да су сви *аутомобили* и *пешаци* на слици детектовани.

4 Тренирање - Darknet програмски оквир

Darknet је програмски оквир за детекцију објеката у реалном времену и представља имплементацију YOLO алгоритма. Написан је коришћењем програмских језика C и CUDA, одакле и потиче изузетна брзина овог система [17].

Тренирање је извршено на GoogleColaboratory виртуелној машини. Апликација се покреће тако што се *Jupyter свеска*, која се налази на GoogleDrive налогу, отвори помоћу GoogleColaboratory апликације. Када се датотека отвори у претраживачу, апликација у позадини алоцира ресурсе за виртуелну машину, после чега је могуће извршавати *ћелије* кода. Jupyter свеска, коришћена уз апликацију GoogleColaboratory, наведена је у литератури [18].

После извршавања првих *ћелија* где се клонира GitHub репозиторијум Darknet-а, подешавају његови параметри, извршава повезивање са GoogleDrive налогом, потребно је валидациони и тренинг скуп пребацити у виртуелну машину.

С обзиром да постоје 52 карте у шпилу тј. класе, параметре у конфигурационој датотеци `yolov4-obj.cfg` треба прилагодити тако да одговарају скупу који тренирамо. Приказана су правила за дефинисање параметара:

- `width = 416` и `height = 416`

  (ова два параметра треба да буду дељиви са 32, 416 је стандардна вредност, али се резултати могу побољшати са 608 и тренирање ће бити спорије)

- `max_batches = 104000`

  (#класа) * 2000 (не мање од 6000, па ако се тренира за 1, 2 или 3 класе биће 6000, али нпр. детектор са 5 класа има вредност `max_batches = 10000`)

- `steps=83200,93600`

  (80% `max_batches`),(90% `max_batches`) (у случају да је `max_batches = 10000`, онда је `steps = 8000,9000`)

- `filters = 171` [2]

  (#класа + 5) * 3 (ако се врши тренирање за само једну класу, онда је `filters = 18`, али ако се тренира за 4 класе онда `filters = 27`)

Датотека `cards.names` садржи називе класа (карата), а датотека `cards.data` садржи више информација (број класа; путању до датотеке са путањама до тренинг слика; путању до датотеке која садржи путање до валидационих слика; путању до датотеке са називима класа; директорујум где се чувају параметри истрениране мреже):

```
classes = 52
train = data/train.txt
valid = data/val.txt
names = data/cards.names
backup = /mydrive/Dokumenti/Fax/DiplomskiKarte/yolov4/backup
```

[2] У датотеци је потребно пронаћи све параметре **classes** и поставити их на **52**, а изнад сваког се налази коновлуциони слој који садржи параметар filters који треба изменити.

Тренирање почиње од иницијалних вредности тежина веза сачуваних у датотеци `yolov4.conv.137` и покреће се командом:

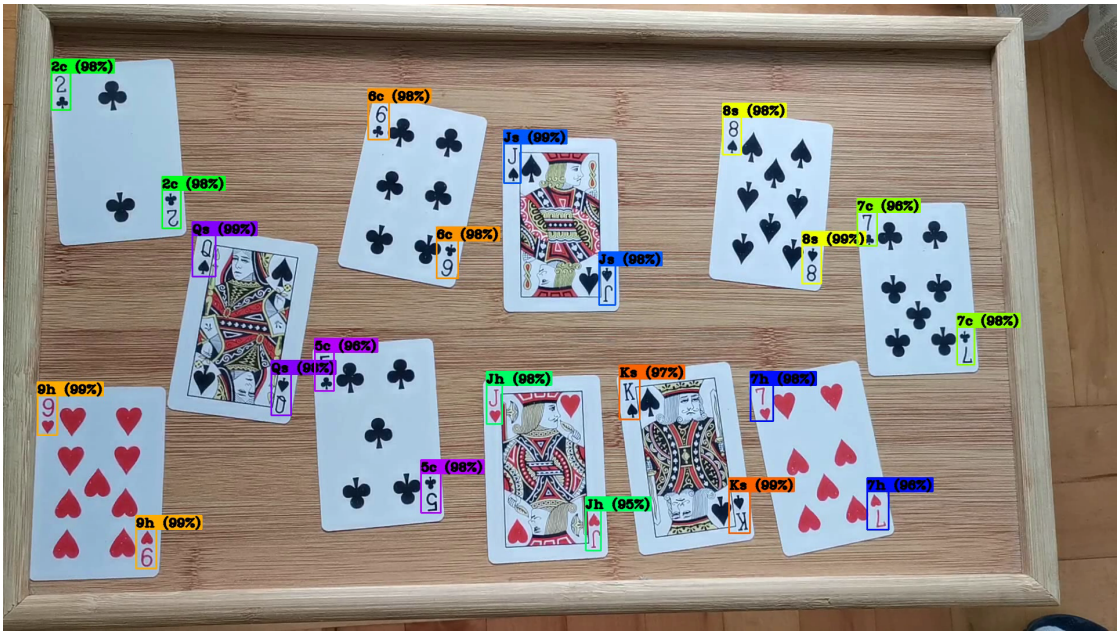
```
./darknet detector train data/obj.data cfg/yolov4-obj.cfg
                        yolov4.conv.137 -dont_show -map
```

Пре тестирања на слици или видеу, потребно је изменити конфигурациону датотеку `yolov4-obj.cfg` и поставити `batch=1` и `subdivisions=1`.

Тестирања над сликом и видеом се позивају следећим командама, респективно:

```
./darknet detector test data/obj.data cfg/yolov4-obj.cfg
                /mydrive/yolov4/backup/yolov4-obj_last.weights
                /mydrive/images/car2.jpg -thresh 0.3

./darknet detector demo data/cards.data cfg/yolov4-obj.cfg
                /mydrive/Dokumenti/Fax/DiplomskiKarte/yolov4/
                      backup/yolov4-obj_last.weights
                -dont_show /mydrive/Dokumenti/Fax/DiplomskiKarte/
                          video/videoTep.mp4
                -thresh 0.8 -i 0 -out_filename /mydrive/Dokumenti
                    /Fax/DiplomskiKarte/video/videoTep_08Thresh.avi
```

Параметар `-thresh 0.8` омогућава да се као резултат прикажу само оне детекције где је вероватноћа ≥ 0.8.



Слика 31: Тестирање над видеом

Детекција у релном времену са WEB камере се покреће командом:

```
./darknet detector demo data/cards.data cfg/yolov4-obj.cfg
                /mydrive/Dokumenti/Fax/DiplomskiKarte/yolov4/
                      backup/yolov4-obj_last.weights -c 0
```

4.1 Промене конфигурација током тренирања

Направљен је тренинг скуп који садржи 50000 слика и валидациони од 10000 (20% тренинг скупа).

Тренирање мреже је покренуто са изнад наведеним параметрима, али с обзиром да је *дужина корака при учењу* (*lerning rate*) била превелика, за финије подешавање је било потребно поставити овај параметар на вредност 0.0001 [19]. Тренирање са мањом вредношћу параметра *lerning rate* је покренуто од 3000. итерације и већ на 3400. итерацији резултати при тестирању су били врло добри.

На 3000. итерацији мера тачности мреже над тренинг скупом (Mean Average Precission - mAP) је била око 97.5% и при тестирању над видеом детекције нису биле тачне.

После тренирања од још 400 итерација са мањом вредношћу параметра *lerning rate*, Mean Average Precission - mAP је износио 100% над тренинг скупом. Команда за израчунавање mAP:

```
./darknet detector map data/cards.data cfg/yolov4-obj.cfg
      /mydrive/Dokumenti/Fax/DiplomskiKarte/yolov4/
                    backup/yolov4-obj_3400.weights
```

Извршене су измене параметара `width` и `height` и постављени на 608, да би мрежа била боље прилагођена већој резолуцији камере.

На крају су коришћени тежински коефицијенти после извршених 5400 итерација.

4.2 Резултати тренирања

У GitHub репозиторијуму *Diplomski rad - YOLO detekcija*, наведеном у литератури [20], се налазе:

- тежински коефицијенти истрениране мреже
- снимак над којим је извршена детекција
- пример снимка карте под различитим осветљењем, који је коришћен за прављење слика тренинг скупа
- LaTeX код овог рада

5 Закључак

Један од изазова оваквог пројекта је то што је програм потребно прилагодити стварним (физичким) објектима. При припреми тренинг скупа треба снимити карте под што разноврснијим осветљењем, шпил карата мора да буде прецизно одштампан, подлога на којој се налази карта треба да буде неутрална ...

Конфигурационе параметре тренирања над скупом је потребно мењати у складу са побољшањем, у виду смањивања функције трошка и повећања тачности над тренинг скупом. С обзиром да тренирање траје неколико сати, резултат промене неких параметара није одмах видљив и узима много времена.

Теоријски део у раду је наведен да би се стекао увид у начин рада алгоритма. YOLO (You Only Look Once) алгоритам је невероватно ефикасан, а то све због приступа да се слика посматра као целина и у једној итерацији извршава детекција (не извршавају се детекције над деловима слике, а у наредној итерацији збирно посматрају резултати, ово је спорије и тај приступ имају неки други алгоритми).

YOLO алгоритам за детекцију у реалном времену је изузетно моћан и може бити примењен над произвољним објектима. Треба имати у виду да може бити примењен и за посматрања на јавним местима. Главни аутор рада о YOLO алгоритму, Joseph Redmon, је радио и на развоју имплементације алгоритма, Darknet програмском оквиру, и повукао се са пројекта из етичких разлога, јер је био забринут за угрожавање приватности људи, а алгоритам је био употребљаван и у војне сврхе.

Задивљујућа је чињеница да је сада рачунар способан да *види*, а да су теоријске идеје изнете у овом раду и библиотеке заступљене у области Машинског учења први пут имплементиране у блиској прошлости.

Несумњиво је да ће, по речима мог ментора, *развој Вештачке интелигенције обележити наше каријере.*

Литература

[1] *playing-card-detection*, приступљено (септембар 2020.) на https://github.com/geaxgx/playing-card-detection.git

[2] api55. Одговор на: *What's the theory behind computing variance of an image?* [Интернет форум]. Приступљено (септембар 2020.) на https://stackoverflow.com/questions/48319918/whats-the-theory-behind-computing-variance-of-an-image

[3] *Contours : Getting Started*, приступљено (септембар 2020.) на https://docs.opencv.org/trunk/d4/d73/tutorial_py_contours_begin.html

[4] *Morphological Transformations*, приступљено (септембар 2020.) на https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html

[5] *pickle — Python object serialization*, приступљено (септембар 2020.) на https://docs.python.org/3/library/pickle.html

[6] AlexeyAB. Одговор на: *Specific format of annotation* [Интернет форум]. Приступљено (септембар 2020.) на https://github.com/AlexeyAB/Yolo_mark/issues/60

[7] *Machine Learning for Beginners: An Introduction to Neural Networks*, приступљено (септембар 2020.) на https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9

[8] lejlot. Одговор на: *Objective function, cost function, loss function: are they the same thing?* [Интернет форум]. Приступљено (септембар 2020.) на https://stats.stackexchange.com/questions/179026/objective-function-cost-function-loss-function-are-they-the-same-thing

[9] Andrew Ng, *Computer Vision*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/Ob1nR/computer-visions

[10] Andrew Ng, *Edge Detection Example*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/4Trod/edge-detection-example

[11] Andrew Ng, *More Edge Detection*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/8Donz/more-edge-detection

[12] Andrew Ng, *One Layer of a Convolutional Network*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/nsiuW/one-layer-of-a-convolutional-network

[13] Andrew Ng, *Object Localization*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/nEeJM/object-localization

[14] Andrew Ng, *Object detection*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/VgyWR/object-detection

[15] Andrew Ng, *Convolutional Implementation of Sliding Windows*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/6UnU4/convolutional-implementation-of-sliding-windows

[16] Andrew Ng, *Convolutional Implementation of Sliding Windows*, [Интернет курс] *Convolutional Neural Networks*, приступљено (септембар 2020.) на https://www.coursera.org/learn/convolutional-neural-networks/lecture/yNwOO/anchor-boxes

[17] *Darknet: Open Source Neural Networks in C*, приступљено (септембар 2020.) на https://pjreddie.com/darknet/

[18] *Running a YOLOv4 Object Detector with Darknet in the Cloud! (GPU ENABLED)*, приступљено (септембар 2020.) на https://colab.research.google.com/drive/1_GdoqCJWXsChrOiY8sZMr_zbr_fH-OFg?usp=sharing

[19] *Playing card detection with YOLO*, приступљено (септембар 2020.) на https://youtu.be/pnntrewHOxg

[20] *Diplomski rad - YOLO detekcija*, приступљено (септембар 2020.) на https://github.com/aleksavelickovic5762015/Diplomski-rad---YOLO-detekcija.git