Jacob Sullivan Follow
Feb 18, 2020 · 9 min read · ✦ · Listen

# Machine learning LEGO image recognition: Using virtual data and YOLOv3

I have been working a lot with LEGO and 3D models lately. For my current project I am looking to build a LEGO image recognition program. My ideal scenario is to grab a handful of LEGO, toss them on the table, take a picture, and have the program catalog the pieces.

*(LEGO® is a registered trademark of the LEGO Group, which does not sponsor, endorse, or authorize this. Visit the official Lego website at http://www.lego.com.)*

The biggest challenge I encounter with any machine learning project is collecting and formatting the training data. I am pretty sure this is the biggest challenge everyone encounters with machine learning. Getting the right data in the right format is always the deal killer.

So I am going to use virtual 3D models of LEGO pieces to create the training data.

## LEGO Sorting Machines

There are several cool examples of people building LEGO sorting machines with LEGO image recognition and machine learning. My scenario is a bit different in that I want to recognize multiple LEGO at one time from one picture. Otherwise, my scenario is very similar to the sor 165 5 first example was here. They seemed to take a brute force approach. Manually labeling a bunch of pieces, training the algorithm, [illegible] ise, repeat … until you have working solution. [illegible] have employed before on other projects! But I have been playing with several LEGO 3D modeling programs, and I was thinking that it would possible to create virtual data. I recently came across a universal LEGO sorting machine that references "virtual" pieces, so I am guessing they are thinking something similar to me. [NOTE: After I finished this project I found a great video by Daniel West explaining that he does use synthetic LEGO data for his universal sorting machine. ]

## Virtual Training Data

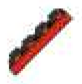
Ok, so what training data do I need? We need lots of pictures of LEGO, labeled with the LEGO id number. I would like pictures of each piece from multiple angles. When I toss a set of LEGO pieces onto the table, they could land in many different orientations. Right-side up, upside down, different rotations, etc. Usually I would want training pictures with multiple LEGO in one picture, with bounding boxes on each piece. But I am going to make several simplifying assumptions to get started.

As I mentioned, I have been spending a lot of time with LEGO modeling software. There are some amazing open source and freeware programs. I downloaded a set of these packages from LDRAW.org. LDRAW is a widely accept format for "bricks". (Bricks are LEGO, and LEGO-compatible, plastic parts.)You should really spend some time exploring the creations of the LDRAW community, super creative!

I ultimately settled on LDCad, an awesome program from Roland Melkert. Well maintained. Great tool. I had first considered using other options that were open-source. I initially planned on figuring out a command line tool, preferable with python, that would automate the image generation from LDRAW LEGO models. The WebGL stuff looked really promising. But ultimately I realized the investment to actually understand the underlying scripts was going to take more time then I had available. And LDCad uses LUA, a script format, with lots of scripting examples available. And that made it a lot easier to pick-up quickly.

## Using LDCad

Let's keep this simple. I wrote a LUA script that iterates through a list of pieces in an LDRAW file. For each LEGO piece, the script rotates the camera through a series of angles to capture images all around the piece. The animation actually looks like the piece itself is rotating,

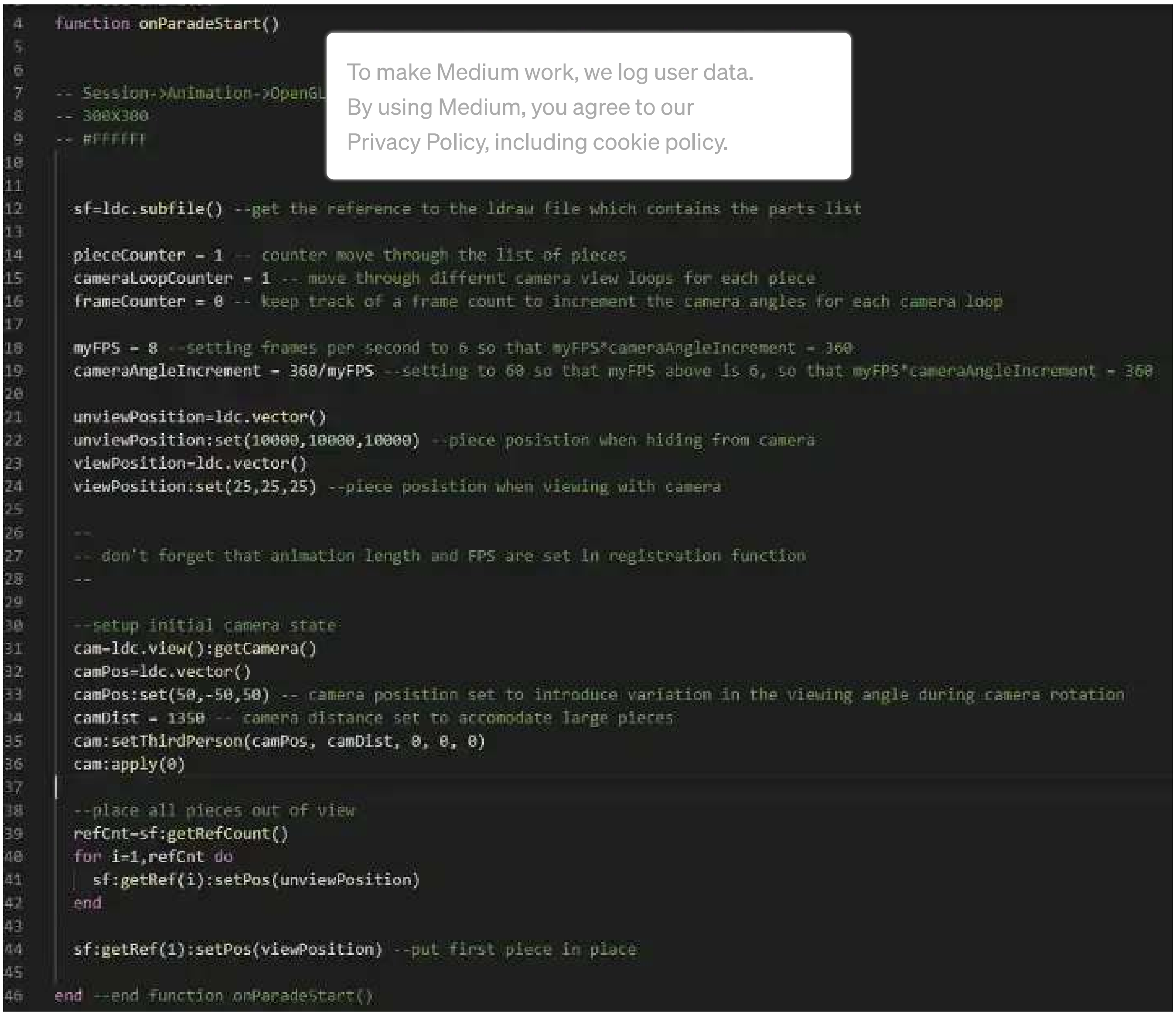


Lets walk through how that is actually accomplished.

LDCad refers to the scripts as “Animations”. I called my animation Parade. I put my animation in the samples.lua script file provided by LDCad for convenience.

The script is broken into two main parts; onParadeStart() and onParadeFrame(). The structure of scripting in LDCad reminds me of microcontroller state machines. Basically the onParadeStart() is run once when the user initiates the animation. And onParadeFrame() is run on each frame of the animation.

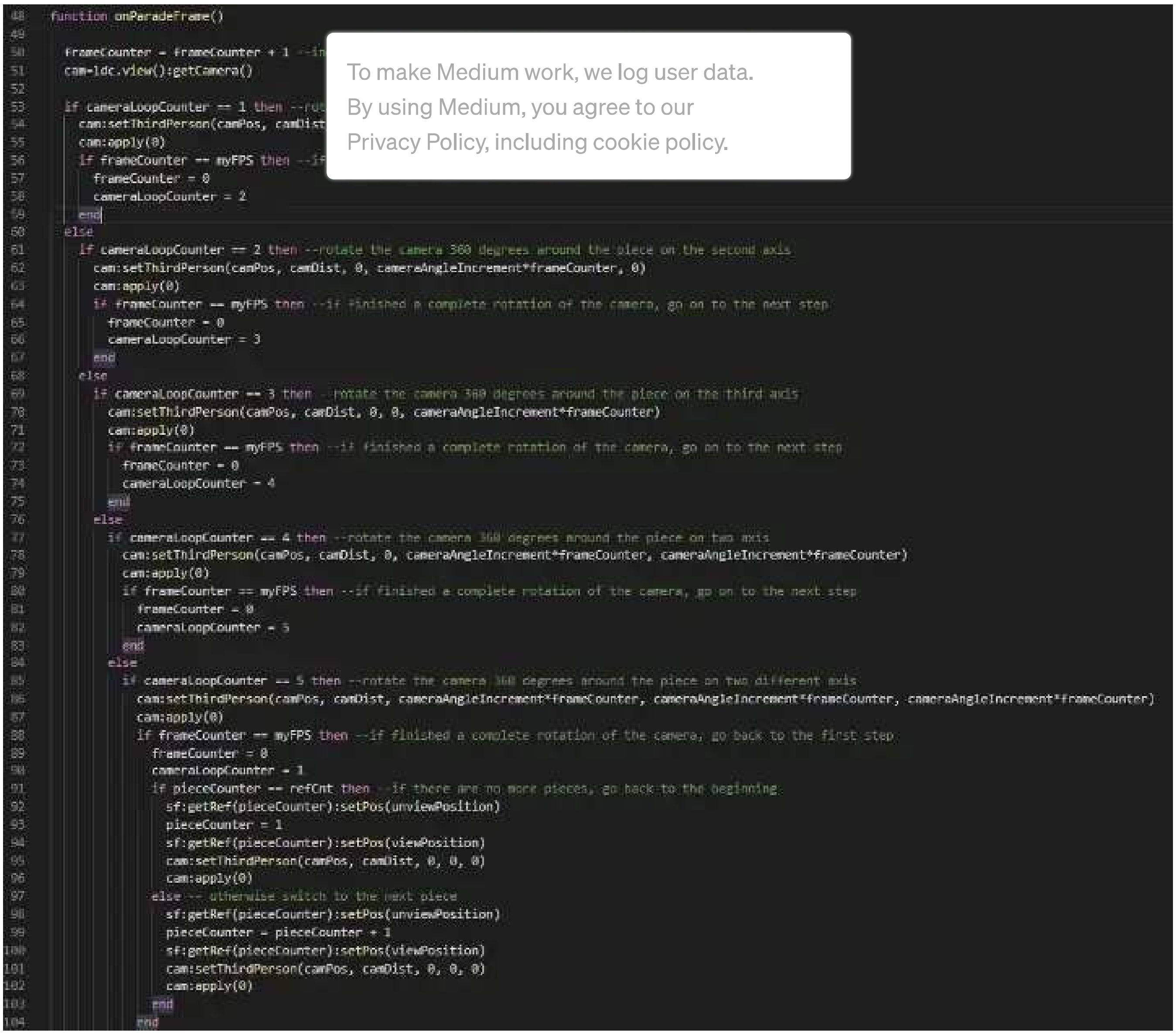
So in onParadeStart(), I initialize the LDRAW file and define constants like unViewPosition and viewPosition. I also initialize counters for looping through the pieces and camera viewing angles.



```
function onParadeStart()


-- Session->Animation->OpenGL
-- 300X300
-- #FFFFFF


  sf=ldc.subfile() --get the reference to the ldraw file which contains the parts list

  pieceCounter = 1 -- counter move through the list of pieces
  cameraLoopCounter = 1 -- move through differnt camera view loops for each piece
  frameCounter = 0 -- keep track of a frame count to increment the camera angles for each camera loop

  myFPS = 8 --setting frames per second to 6 so that myFPS*cameraAngleIncrement = 360
  cameraAngleIncrement = 360/myFPS --setting to 60 so that myFPS above is 6, so that myFPS*cameraAngleIncrement = 360

  unviewPosition=ldc.vector()
  unviewPosition:set(10000,10000,10000) --piece posistion when hiding from camera
  viewPosition=ldc.vector()
  viewPosition:set(25,25,25) --piece posistion when viewing with camera

  --
  -- don't forget that animation length and FPS are set in registration function
  --

  --setup initial camera state
  cam=ldc.view():getCamera()
  camPos=ldc.vector()
  camPos:set(50,-50,50) -- camera posistion set to introduce variation in the viewing angle during camera rotation
  camDist = 1350 -- camera distance set to accomodate large pieces
  cam:setThirdPerson(camPos, camDist, 0, 0, 0)
  cam:apply(0)

  --place all pieces out of view
  refCnt=sf:getRefCount()
  for i=1,refCnt do
    sf:getRef(i):setPos(unviewPosition)
  end

  sf:getRef(1):setPos(viewPosition) --put first piece in place

end --end function onParadeStart()
```

onParadeStart()

In onParadeFrame(), I built a series of if … else … if to act like a switch statement. It loops through all the pieces. Loops through all the animation patterns. Loops through all of the camera viewing angles. Increment the counters and updating the camera positions as it goes.



```
function onParadeFrame()

  frameCounter = frameCounter + 1 --in
  cam=ldc.view():getCamera()

  if cameraLoopCounter == 1 then --ro
    cam:setThirdPerson(camPos, camDist
    cam:apply(0)
    if frameCounter == myFPS then --if
      frameCounter = 0
      cameraLoopCounter = 2
    end
  else
    if cameraLoopCounter == 2 then --rotate the camera 360 degrees around the piece on the second axis
      cam:setThirdPerson(camPos, camDist, 0, cameraAngleIncrement*frameCounter, 0)
      cam:apply(0)
      if frameCounter == myFPS then --if finished a complete rotation of the camera, go on to the next step
        frameCounter = 0
        cameraLoopCounter = 3
      end
    else
      if cameraLoopCounter == 3 then --rotate the camera 360 degrees around the piece on the third axis
        cam:setThirdPerson(camPos, camDist, 0, 0, cameraAngleIncrement*frameCounter)
        cam:apply(0)
        if frameCounter == myFPS then --if finished a complete rotation of the camera, go on to the next step
          frameCounter = 0
          cameraLoopCounter = 4
        end
      else
        if cameraLoopCounter == 4 then --rotate the camera 360 degrees around the piece on two axis
          cam:setThirdPerson(camPos, camDist, 0, cameraAngleIncrement*frameCounter, cameraAngleIncrement*frameCounter)
          cam:apply(0)
          if frameCounter == myFPS then --if finished a complete rotation of the camera, go on to the next step
            frameCounter = 0
            cameraLoopCounter = 5
          end
        else
          if cameraLoopCounter == 5 then --rotate the camera 360 degrees around the piece on two different axis
            cam:setThirdPerson(camPos, camDist, cameraAngleIncrement*frameCounter, cameraAngleIncrement*frameCounter, cameraAngleIncrement*frameCounter)
            cam:apply(0)
            if frameCounter == myFPS then --if finished a complete rotation of the camera, go back to the first step
              frameCounter = 0
              cameraLoopCounter = 1
              if pieceCounter == refCnt then --if there are no more pieces, go back to the beginning
                sf:getRef(pieceCounter):setPos(unviewPosition)
                pieceCounter = 1
                sf:getRef(pieceCounter):setPos(viewPosition)
                cam:setThirdPerson(camPos, camDist, 0, 0, 0)
                cam:apply(0)
              else -- otherwise switch to the next piece
                sf:getRef(pieceCounter):setPos(unviewPosition)
                pieceCounter = pieceCounter + 1
                sf:getRef(pieceCounter):setPos(viewPosition)
                cam:setThirdPerson(camPos, camDist, 0, 0, 0)
                cam:apply(0)
              end
            end
```

onParadeFrame()

I set frames per second(FPS) to 8 because this give me 8 different pictures, each from a different camera angle, for each camera viewing loop. The length of the animation is based on the number of pieces multiplied by the number of camera viewing loops. And you will need all of this information when you are extracting pictures for the machine learning training phase in a bit.

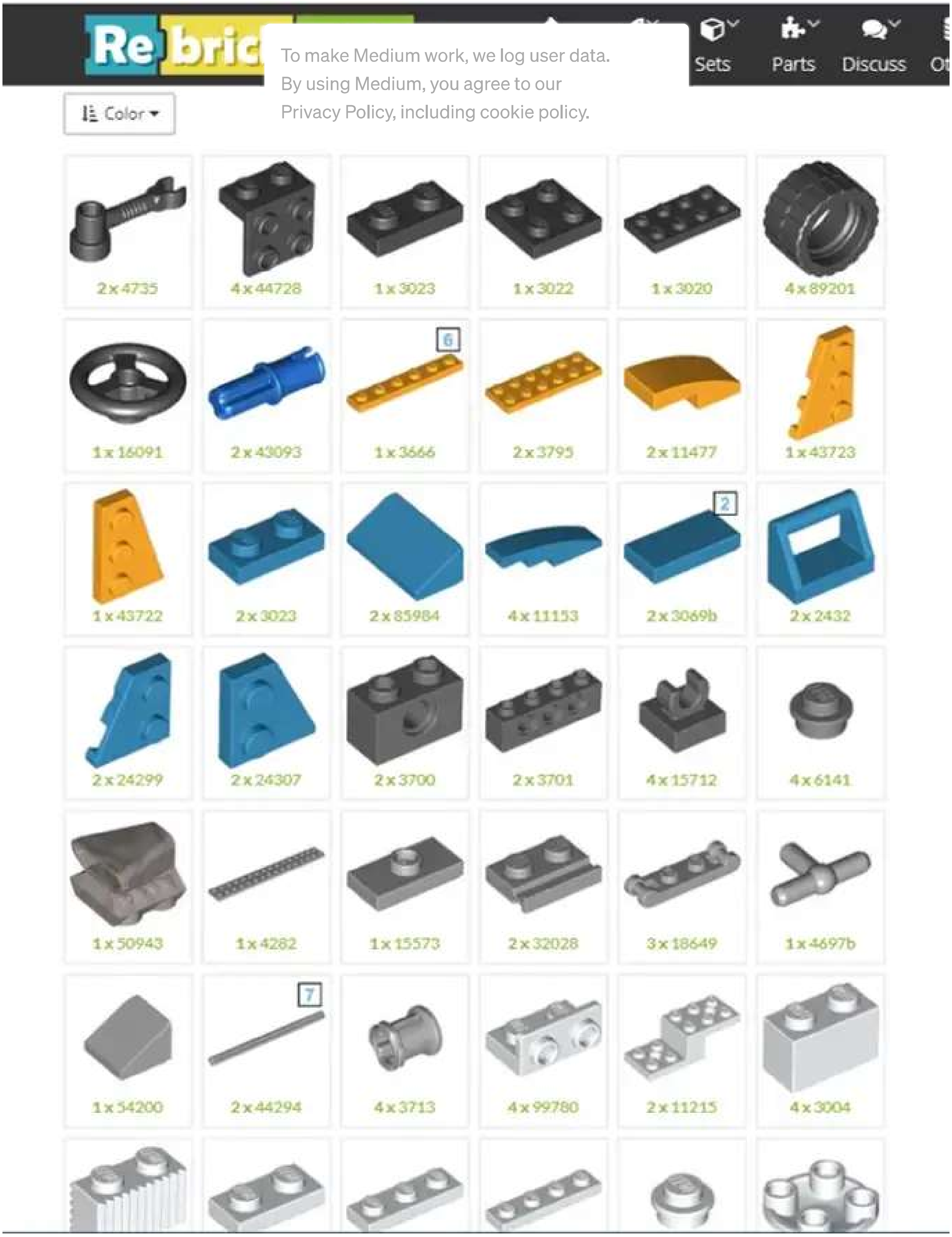
Once the Parade script is setup and registered, it will appear as a menu option in the LDCad program. You can open your LDRAW file in LDCad and then run the Parade animation. LDCad allows you to export the animation as a series of png image files. Nice!

## Making LEGO images

I grabbed a small LEGO set I had unopened at the house. **31072 — Extreme Engines.**

31072 — Extreme Engines

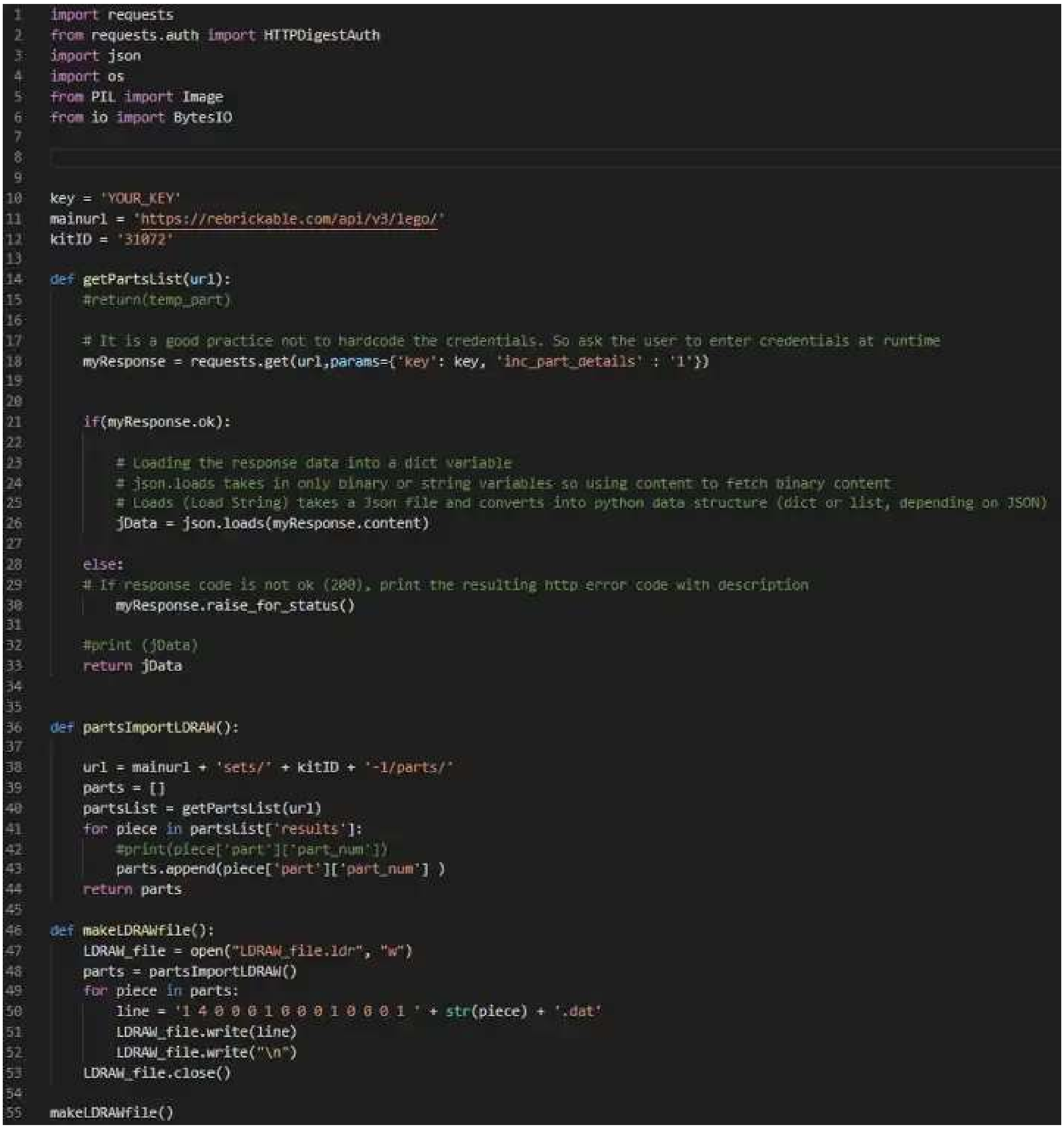
It is a smaller kit with 109 pieces total pieces, 49 unique pieces. I think it will be a good test of the project. I used a python script and the rebrickable.com API to download a parts list for that specific kit.



rebrickable.com

Within the JSON parts list returned by the API, are the LEGO parts ids and corresponding LDRAW parts ids. The rebrickable api documentation indicates that the rebrickable part number is "mostly" based on LDRAW. Digging into the JSON output of the api call, I found that the rebrickable part id was a more reliable

LDRAW id than the id listed under external_ids:LDraw. Not 100% sure why, but it worked well enough for a LDRAW file with a row for each unique LD



```python
import requests
from requests.auth import HTTPDigestAuth
import json
import os
from PIL import Image
from io import BytesIO



key = 'YOUR_KEY'
mainurl = 'https://rebrickable.com/api/v3/lego/'
kitID = '31072'

def getPartsList(url):
    #return(temp_part)

    # It is a good practice not to hardcode the credentials. So ask the user to enter credentials at runtime
    myResponse = requests.get(url,params={'key': key, 'inc_part_details' : '1'})


    if(myResponse.ok):

        # Loading the response data into a dict variable
        # json.loads takes in only binary or string variables so using content to fetch binary content
        # Loads (Load String) takes a Json file and converts into python data structure (dict or list, depending on JSON)
        jData = json.loads(myResponse.content)

    else:
    # If response code is not ok (200), print the resulting http error code with description
        myResponse.raise_for_status()

    #print (jData)
    return jData


def partsImportLDRAW():

    url = mainurl + 'sets/' + kitID + '-1/parts/'
    parts = []
    partsList = getPartsList(url)
    for piece in partsList['results']:
        #print(piece['part']['part_num'])
        parts.append(piece['part']['part_num'] )
    return parts

def makeLDRAWfile():
    LDRAW_file = open("LDRAW_file.ldr", "w")
    parts = partsImportLDRAW()
    for piece in parts:
        line = '1 4 0 0 0 1 0 0 0 1 0 0 0 1 ' + str(piece) + '.dat'
        LDRAW_file.write(line)
        LDRAW_file.write("\n")
    LDRAW_file.close()

makeLDRAWfile()
```

The LDRAW format requires a location and orientation for each piece on each row. Although these coordinates are not important to our animation, the format does need to conform to the LDRAW file format for a "sub-file reference line."

> *1 <color> x y z a b c d e f g h i <file>*

- 1 denotes "sub-file reference line"

- x y z is the location coordinates
- a b c d e f g h i is a t mogeneous transformation mat
- I picked 4(red) for the color. (The color shouldn't impact the recognition.)
- An example line would be … 1 4 0 0 0 1 0 0 0 1 0 0 0 1 3023.dat

Once I had the LDRAW file constructed, I set the animation length to 245 seconds in the Registration function of the samle.lua script. Because the duration should be (number of pieces)*(number of camera loops)

> *49 * 5 = 245*

In the LDCad program I loaded my LDRAW file and ran the animation to check it out. I then selected Session->Animation->OpenGL export option. I set 300X300 image size. Maybe bigger than needed, but I want enough pixels that works on a range of piece sizes. I set the Background to #FFFFFF (white). And exported to my folder of choice.

After I exported the animation, I had 1,960 images. 40 images for each of the 49 unique pieces. This may be overkill. Or it may have insufficient variation in the angles and lighting. Innovation and experimentation will be iterative.

Now that we have some images, let's move to the machine learning!

## YOLOv3 Object Detection

I am going to use the YOLOv3 algorithm for the training. Although YOLO specifically has limitations with "small objects" grouped "close together", I think it will be fine for this implementation. Small and close together are relative terms. If we were trying to discern LEGO on a table with bigger objects in the image too, these limitations would be a problem. But if the entire picture is a collection of similarly sized LEGO spread across the image … then it will be fine. FYI — YOLO is absolutely overkill.

YOLO(you only look once) is optimized for speed. The most common applications are in real-time or near-time video analysis. We could have picked fast-CNN or other single shot detector for this project. But I want to use YOLOv3 because I have not used it before, and trying new things are fun!

I found a quick walk-through on "How to Train your own YOLOv3 detector from scratch".

Let's give it a whirl!

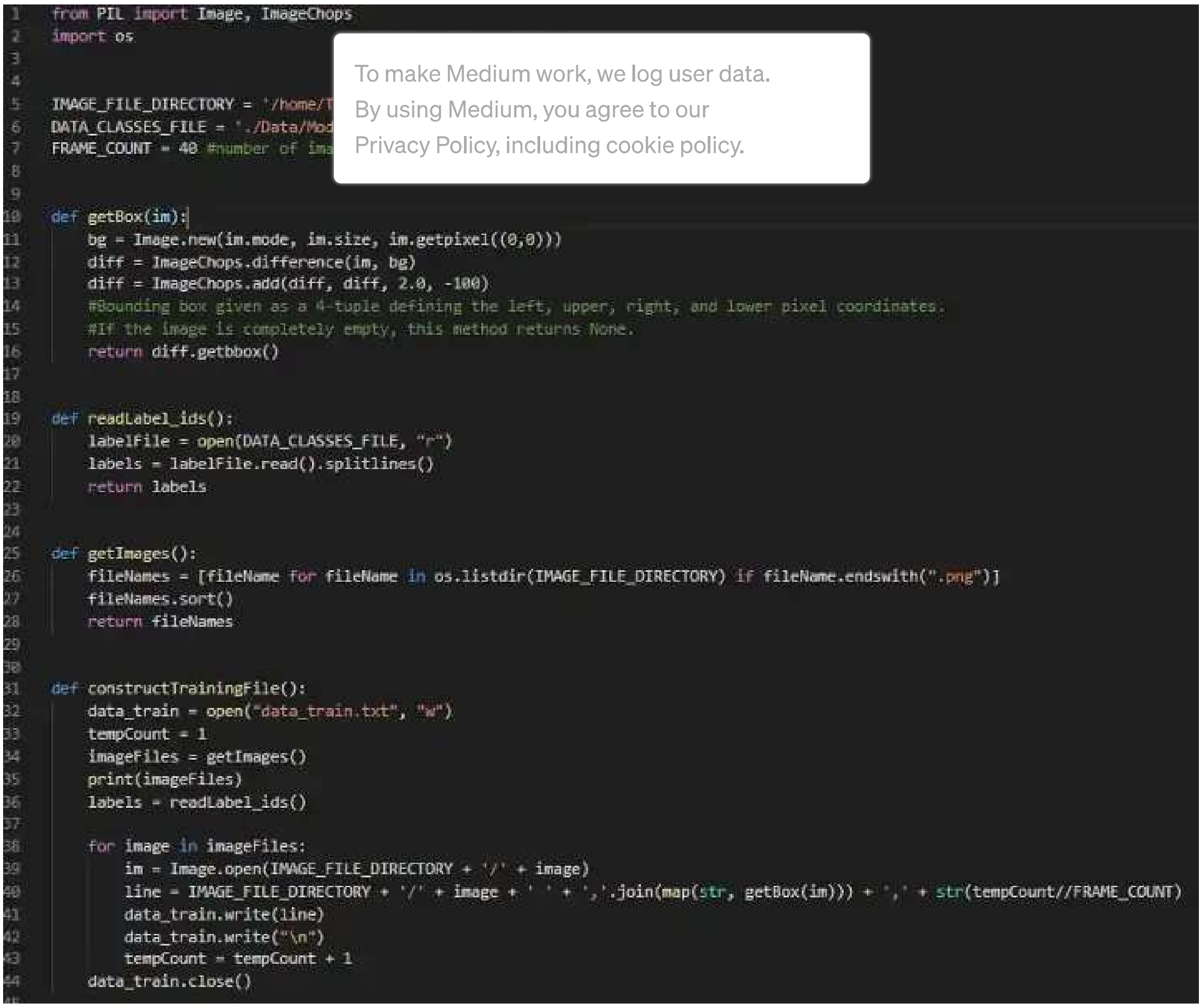
## Training Data Formatting

YOLOv3 needs a set of labeled training images with bounding boxes. And a text file describing the training data and images. The format required for the text file requires a line for each training image that contains … Absolute file path, xmin, ymin, xmax, ymax, label_id. The x and y coordinates are the bounding box of the object to be recognized. The label_id is a reference to data_classes. Ultimately the label_id will be a reference to the part id for each LEGO piece. But we will put the part ids in a Data_classes.txt file.

So I built a short python script to make the Data_classes.txt file from the list of parts id. And placed that file in "**Data->Model_Weights->Data_classes.txt**"

I have the image file paths and can get the label_id from my array of part numbers I used to build the Data_classes.txt file. But I also need bounding boxes. I am sure there are many efficient ways to complete that task. But I used ImageChops from PIL. It is a crop utility. And since the training data is artificial and contains a uniform background, cropping the image provided the same coordinates as a tight bounding box.

I created a python script that puts all of this information together into a data_train.txt file. And then placed that file in the "**Training_Images->vott-csv-export->data_train.txt**".



```
from PIL import Image, ImageChops
import os


IMAGE_FILE_DIRECTORY = '/home/T
DATA_CLASSES_FILE = './Data/Mod
FRAME_COUNT = 40 #number of ima


def getBox(im):
    bg = Image.new(im.mode, im.size, im.getpixel((0,0)))
    diff = ImageChops.difference(im, bg)
    diff = ImageChops.add(diff, diff, 2.0, -100)
    #Bounding box given as a 4-tuple defining the left, upper, right, and lower pixel coordinates.
    #If the image is completely empty, this method returns None.
    return diff.getbbox()


def readLabel_ids():
    labelfile = open(DATA_CLASSES_FILE, "r")
    labels = labelFile.read().splitlines()
    return labels


def getImages():
    fileNames = [fileName for fileName in os.listdir(IMAGE_FILE_DIRECTORY) if fileName.endswith(".png")]
    fileNames.sort()
    return fileNames


def constructTrainingFile():
    data_train = open("data_train.txt", "w")
    tempCount = 1
    imageFiles = getImages()
    print(imageFiles)
    labels = readLabel_ids()

    for image in imageFiles:
        im = Image.open(IMAGE_FILE_DIRECTORY + '/' + image)
        line = IMAGE_FILE_DIRECTORY + '/' + image + ' ' + ','.join(map(str, getBox(im))) + ',' + str(tempCount//FRAME_COUNT)
        data_train.write(line)
        data_train.write("\n")
        tempCount = tempCount + 1
    data_train.close()
```

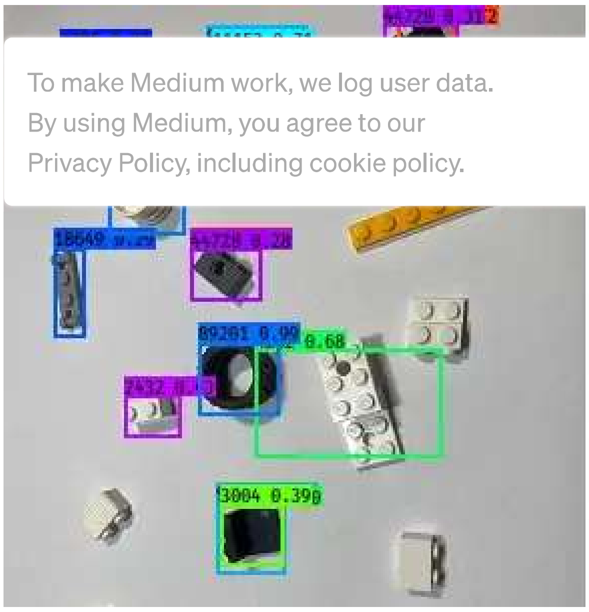
There are a near infinite number of tweaks and optimizations that can be attempted on a model like YOLOv3. Setting up these two files will allow you to train the first iteration if you start with the reference implementation linked above.

You can now run Train_YOLO.py from the command line.

## Results!

So the model trained for a long time. 102 Epochs. It was like 48 hours on my home server. Again, I am sure there are many many optimizations to be made.

I opened my box of LEGO, grabbed a handful of pieces, tossed them on the table, took a picture, and ran the inference.



Inference Detection

My oh my. Not too bad. It got a couple of these correct on the first training run!

As to be expected, lighting and shadows play a big role. Orientation of the pieces factored as well. The more unique pieces like the wheels are picked out more accurately and frequently.

I think it is a heck of a start! Without any model tweaking, and using the simple virtual data camera rotation animation program. Towards the end of this project, I found a video by Daniel West explaining that he does use synthetic LEGO data for his universal sorting machine. He talks about the challenges of virtual data and tricks for improving the outcome by mixing in some real data too. Maybe I'll try that out. But I've also got another idea for a cool project using LEGO and machine learning. And never enough time in the day.

Machine Learning Lego Image Processing STEM Toys

## Sign up for The Variable

By Towards Data Science