DnD Dice Counter using Neural Networks

ECE4179 – NEURAL NETWORKS

ERIC HORNG

Introduction

Dungeons and Dragons is a popular tabletop roleplaying game played with a small group of people. It involves a player, the “Dungeon Master” creating a setting (such as a fantasy world) and a story, with the other players playing characters in this narrative to evolve the story. A core mechanic within the game is the use of different sided dice to dictate actions, such as the damage you do to a creature, the damage you may take from activating a trap, or simply whether lockpicking a door is successful. The game uses 6 kinds of dice, each with a different number of faces:

Figure 1. DnD Dice: (left to right) d4, d6, d8, d10, d12 and d20

At later stages of the story, you may end up having to roll dozens of dice at the same time as the amount of damage you dish out/take increases as you level up your character. For example, the maximum amount of damage my personal character can do to a creature is 21d6 (or 21 6-sided dice rolls). Counting this many dice at the same time can introduce small arithmetic errors and slows down the pace of the game. While electronic dice rollers exist where you can press a button and get a number, this doesn’t have the same feeling of rolling a physical dice. It would be like playing poker in a casino through an iPad.

This project involves the design of a neural network which detects and sums up the dice given an image. We’re only interested in detecting d6, d8, d10 and d12 dice, as these are the most used dice in the game and have a suitable shape for accurate top down classification.

YOLOv3 Object Detection

The first stage in the 2-stage pipeline is to identify the top of each dice. YOLOv3 (You Only Look Once) was used to detect the top of each dice. YOLOv3 is a fast and accurate object detection architecture that applies a neural network to the entirety of an image, divides the image into regions and predicts bounding boxes and probabilities for each region [1]. This contrasts with other object detectors which partition an image into thousands of sub-images and applies a classifier at different scales [1].

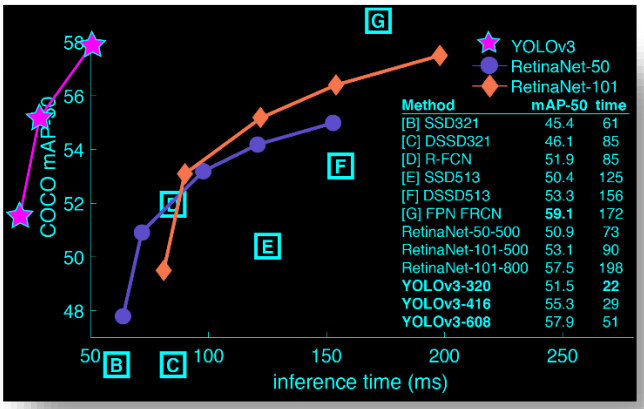


Figure 2. YOLOv3 network performance

For our network, we use Chris Fotache's implementation [1] in PyTorch with our own custom dataset. He rehashes the structure from training YOLOv3 on the COCO image dataset, hence why some filenames are in reference to COCO, and simplifies the data structure and provides a dataloader.

Dataset Generation

For training we use a combination of datasets available on the internet [3], Google images and images taken from a phone on different backgrounds and lighting. The online dataset alone was unsuitable for our network as each image only contained one dice type, taken from different angles. Our network needed to be able to detect multiple dice in an image hence why we supplement the training data with our own images. Our phone data is resized from 3000*4000 to 750*1000 to decrease training time. Our training dataset size is 852 images.

Figure 3. Kaggle Data (left), Our Data (right)

Once all the images have been aggregated, we need to annotate each individual image with the type of object we want to detect. For training, YOLO takes as input an image and a text file, with the text file named the same as the image. Each line in the text file is of the form:

$$class\ x\ y\ box\_width\ box\_height$$

$x, y\ box\_width\ and\ box\_height$ are in normalized coordinates (i.e. $x = 0.5$ corresponds to an x value halfway through the image, $x = 0.5, y = 0.5$ corresponds to the box starting in the direct centre of the image etc.) and dictate the location and size of a bounding box around the object. If there are 5 objects in the image, then the text file will have 5 lines. We use labelImg to draw a bounding box around each top-facing side of each dice and the program automatically generates each annotation file for us.

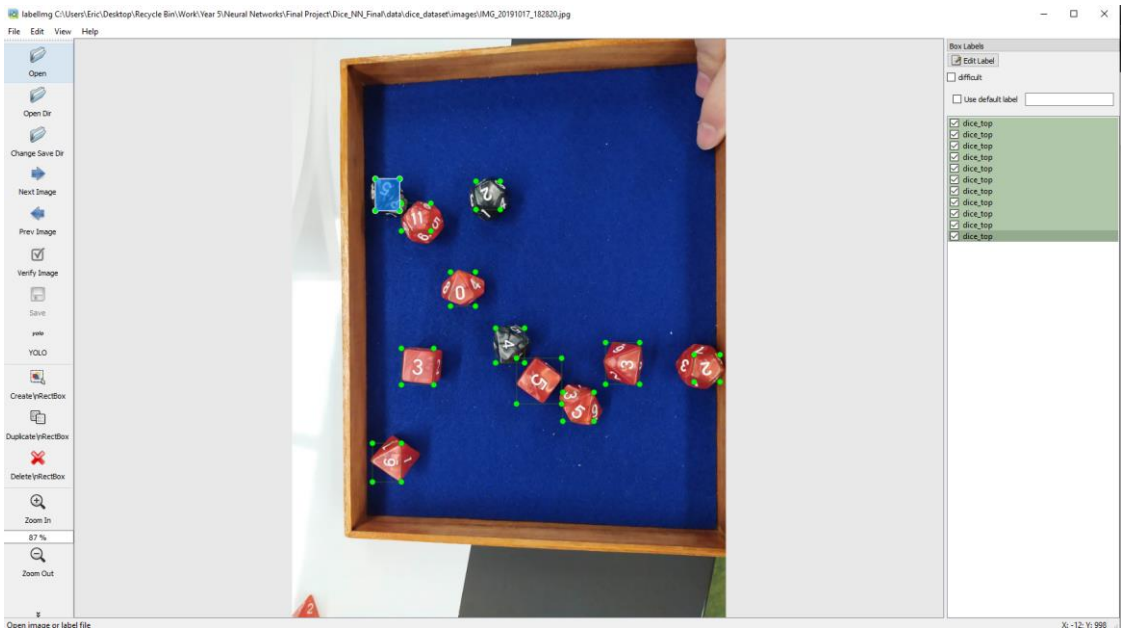


Figure 4. Annotating images through labelImg

Once all images have been annotated, we place them in a defined structure. We then run the script *create_list.py* to separate the data into a training (90%) and validation set (10%) and generate another two text files *train.txt* and *val.txt* containing the paths to images in each respective set.

```
Main Folder
--- data
    --- dataset name
        --- images
            --- img1.jpg
            --- img2.jpg
            ..........
        --- labels
            --- img1.txt
            --- img2.txt
            ..........
        --- train.txt
        --- val.txt
```

Configuring YOLOv3

The next step is to configure the settings of YOLO to correctly train on our data. Within the working directory of the application is a folder named config/ with *coco.data*, *coco.names*, *yolov3.cfg* and *yolov3.weights*.

In *coco.data*, we change the number of classes to 1, and point train and valid variables to our generated text files.

```
classes = 3
train=data/alpha/train.txt
valid=data/alpha/val.txt
names=config/coco.names
backup=backup/
```

In *coco.names* we only have the name of our class: DICE_TOP.

Yolov3.cfg contains the network definition and specifies hyperparameters such as learning rate and batch size. Within the first category [net], we change our batch to 16 (maximum batch size that can run on Google's 12GB K80 without running out of memory. We can also specify the subdivision, which breaks up the batch size into further batches however this was kept at 1. We can specify the width and height of the input to the network, changing it between 320x320, 416x416 or 608x608. Yolov3 automatically resizes the input images to these dimensions. Having a smaller image size can reduce accuracy and GPU memory usage, but also decrease training time and inference time and vice-versa for a large image size. A learning rate of 0.006 was left as default and steps and scales was changed to 500,800 and 0.1,0.1 respectively. These determine the reduction of learning rate during training i.e. multiply learning rate by 0.1 at 500 steps and a further 0.1 at 800 steps.

```
[net]
batch=16
subdivisions=1
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.006
burn_in=1000
max_batches = 500200
policy=steps
steps=500,800
scales=.1,.1
```

The next change we make is in the convolutional layers right before the [yolo] definition and the number of classes in each [yolo] section. We change the filters according to:

$$filters = (classes + 5) * 3$$

Hence for our dice detector, filters = 18.

Training and Results

Our computer doesn't have enough memory for a batch size of 1, hence we use Google Colab to perform training for us. We feed a pretrained model (available on the YOLO website), trained on the Coco dataset then alter these weights to fit our dataset, which is faster than training a network from scratch. We use Chris' *ListDataset* function to generate an appropriate dataset class to feed into PyTorch's dataloader. The training function is the standard training function used in previous neural network assignments. We train over 50 epochs, which takes approximately 3-4 hours and save the

minimum validation loss and minimum training loss weights for use in our program. Validation is tested every epoch. Minimum validation loss occurs at epoch 49.

Hyperparameter	Value
Learning rate	0.006 (0.0006 at 500 batches, 0.00006 at 800 batches)
Network	Yolov3
Pretrained	Yolov3 (Coco)
Optimizer	Adam
Loss Function	CrossEntropy
Max Epochs	50
Min Validation Epoch	49

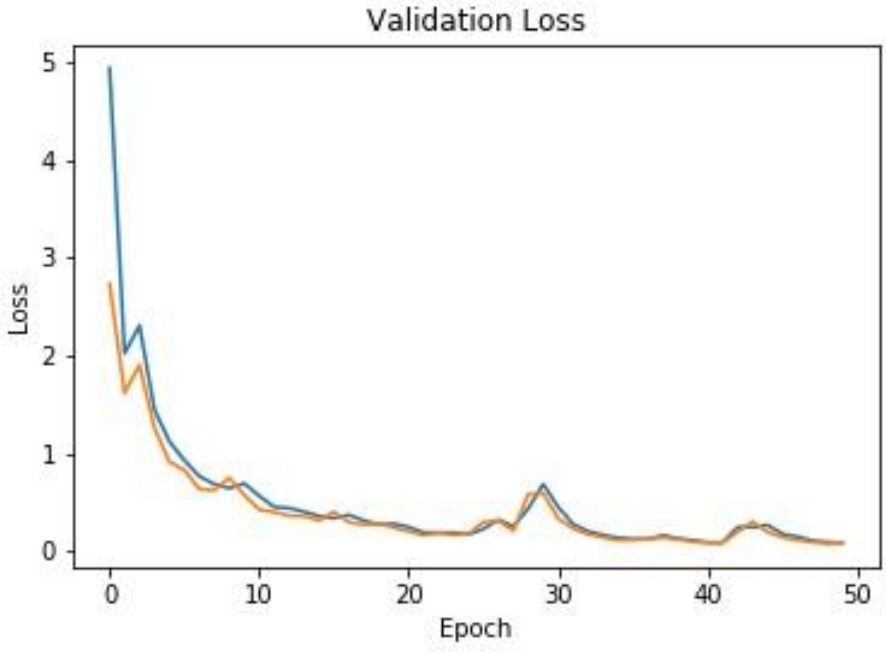


Figure 5. Object detector training and validation loss (50 epochs)

The loss plot shows that the network plateaus around epoch 20, with a large spike around epoch 30, possible where the learning rate is decreased.

Using the trained model on an image gives:

Figure 6. Object Detection model used on test images

Resnet101 Transfer Learning Classifier

For our classifier, we're using Resnet101's network and training the fully connected layer at the end. We're using transfer learning as it gives better accuracy and faster training than developing a CNN from scratch.

Creating classifier dataset

Since we couldn't find a proper dataset containing the top faces of dice, we generate our own. This is done by using our object detection network on all training and validation images from our YOLO custom dataset. The notebook *Generate_Number_Dataset.ipynb* [4] does this and saves cropped jpg images in the *generated_dataset* folder. The script first resizes the images into 416x416, maintaining the aspect ratio and padding if necessary. It then feeds it into the Darknet (Yolov3) network which returns $x, y, box\ width, box\ height$ values in a matrix. Bounding boxes with a lower confidence level than a threshold is then removed, and non-maximal suppression used on the remaining results to remove repeated bounding boxes on the same object.

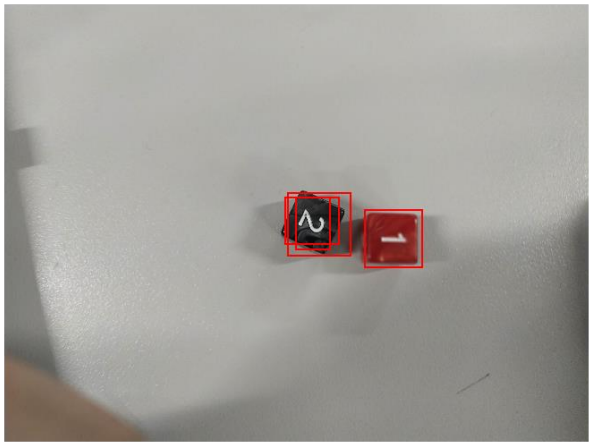


Figure 7. Non-maximal suppression removes repeated detections

After the script is finished, we manually go through each image, remove false detections and move the image to its corresponding class 1 to 12 in the train folder (and moving 10% of all images to val folder).

Figure 8. Output of Generate Dataset script

Training and Results

Once the data has been correctly set up, we can run the usual training function on the data and save the best validation accuracy model.

Hyperparameter	Value
Learning rate	0. 001
Network	Resnet101 > FC(fc_features,12)
Optimizer	Adam
Loss Function	CrossEntropy
Pretrained	Resnet50
Max Epochs	100
Max Validation Accuracy Epoch	95 (75% Accuracy)

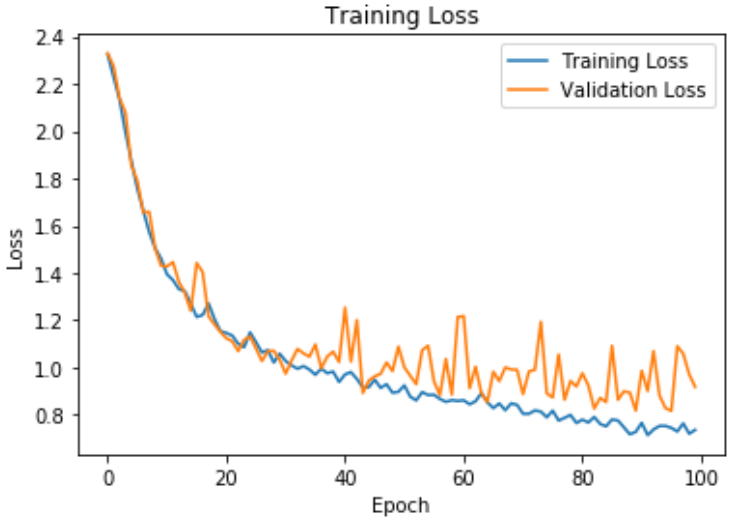


Figure 9. Classifier training and validation loss

Figure 10. Classifier accuracy

The loss and accuracy graphs for the classifier is very 'noisy' and chaotic. This is due to the low number of validation data, as a wrong prediction can wildly decrease the accuracy. From the plot, the network seems to plateau around epoch 40.

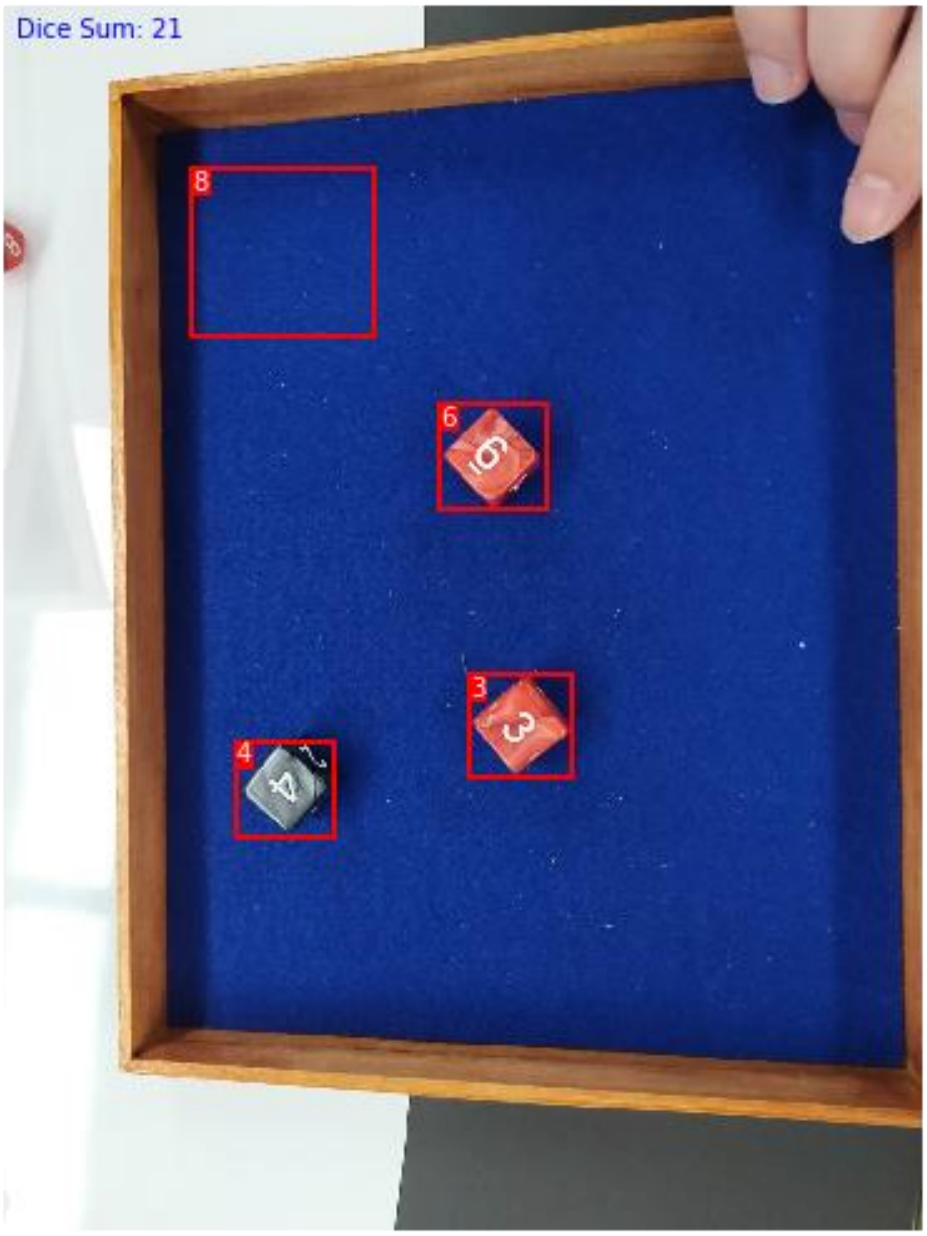
```
Training Confusion Matrix:
[[122.   0.   0.   1.   1.   3.   0.   8.   3.   3.   0.   0.]
 [  1.  42.   0.   0.   0.   4.   1.   2.   0.   2.   0.   0.]
 [  3.   0. 124.   4.   2.   1.   5.  16.   3.  57.   1.  13.]
 [  1.   0.   1. 192.   0.   3.   3.   1.   0.   3.   0.   1.]
 [  6.   1.   1.   1.  17.   0.   1.  12.   2.   5.   1.   1.]
 [  0.   1.   0.  14.   0. 171.   4.   3.   1.   4.   0.   2.]
 [  0.   1.   0.  13.   0.   4.  67.   1.   0.   2.   0.   0.]
 [ 19.   1.   9.   5.   2.   4.   1. 211.   4.  20.   0.   1.]
 [  1.   1.   3.   1.   0.   3.   1.   4.  44.   0.   0.   0.]
 [  1.   1.   4.   6.   1.   3.   1.   7.   2. 249.   0.   6.]
 [  1.   1.   0.   1.   0.   0.   2.   0.   0.   3.  23.   6.]
 [  1.   1.   8.   8.   0.   3.   4.   5.   2.  38.   0. 164.]]

Validation Confusion Matrix:
[[18.  0.  0.  1.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  6.  0.  1.  0.  0.  0.  2.  0.  2.  0.  0.]
 [ 0.  0. 17.  1.  0.  0.  0.  3.  0.  4.  0.  1.]
 [ 0.  0.  0. 27.  0.  0.  0.  0.  1.  1.  0.  1.]
 [ 1.  0.  0.  1.  1.  0.  0.  8.  0.  1.  0.  0.]
 [ 0.  0.  0.  3.  0. 12.  2.  1.  1.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  3. 10.  0.  0.  0.  0.  0.]
 [ 3.  0.  0.  2.  1.  1.  0. 30.  0.  5.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  2.  9.  1.  1.  0.]
 [ 0.  0.  0.  2.  0.  1.  0.  0.  0. 27.  0.  1.]
 [ 0.  2.  1.  0.  0.  0.  0.  0.  0.  2.  3.  2.]
 [ 0.  0.  0.  1.  0.  0.  1.  4.  0.  2.  0. 18.]]
```

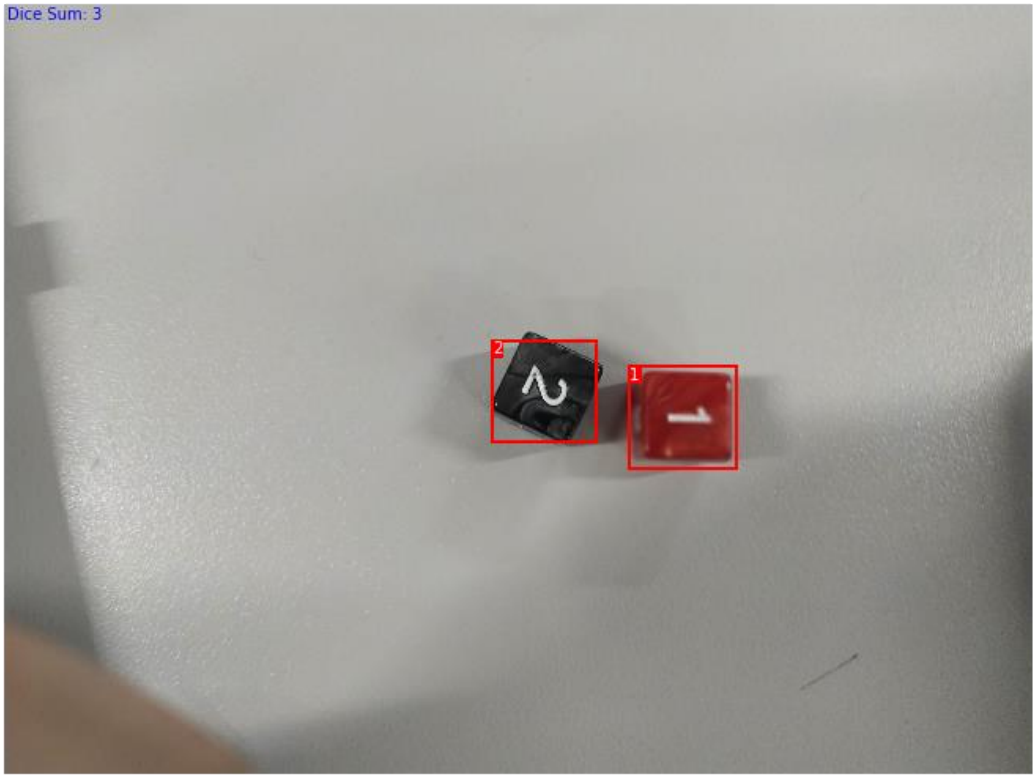
Dice Counting

Once both stages of the neural network are trained, we can run an image through the pipeline to get the sum of all dice.

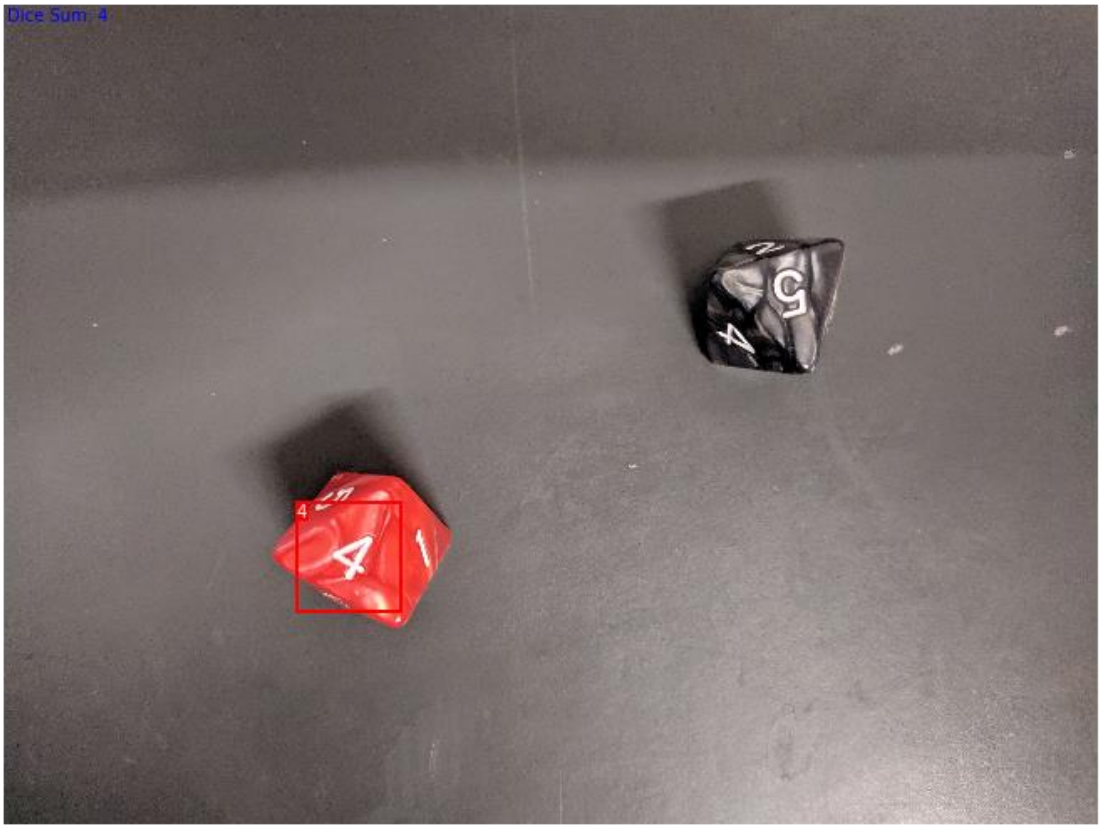
The program first enters the first stage: object detection. It loads the model with the best weights from training and processes the image. Then for each detected bounding box, draws a red box on the image and extracts the image within the box. In the second stage, the extracted portion is fed into the Resnet101 classifier to get a label between 0 to 11. This label is then translated into its respective dice number, written on the image and added to the total sum of dice. Finally, when all boxes have been processed, it writes the total sum in the top left of the image.

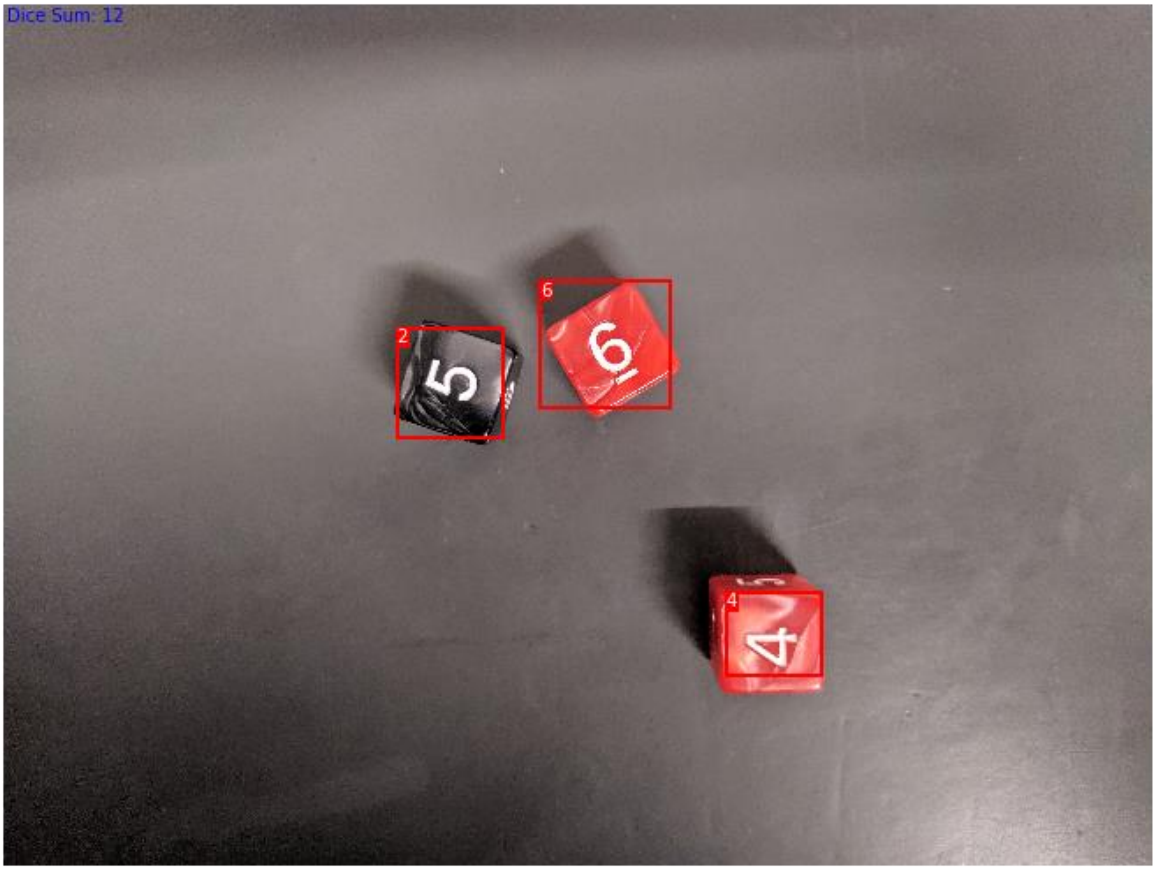
Final Network Results




Dice Sum: 3
2
1


Dice Sum: 6
2
4

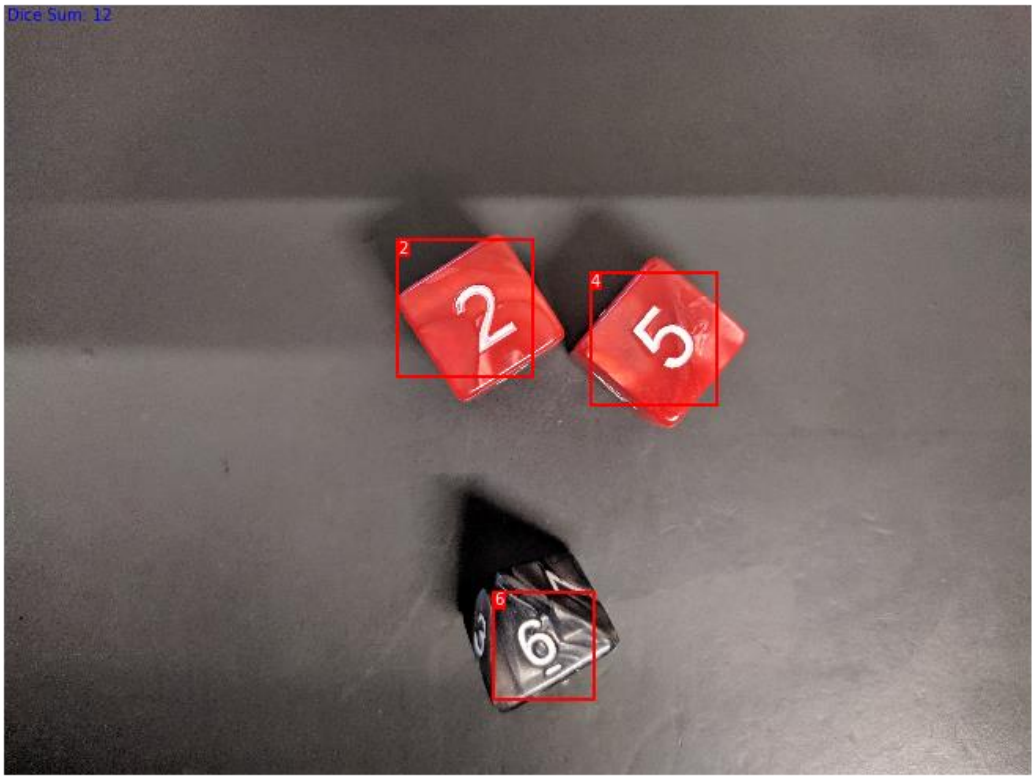

Dice Sum: 4
4
4


Dice Sum: 12
2
5
6
6
4
4


Dice Sum: 21
5
6
10


Dice Sum: 11
10
1


Dice Sum: 7
7
1
10
5


Dice Sum: 12
2
2
4
5
6
6


Dice Sum: 4
4


Dice Sum: 4
4

The network seems to work best on d6s but have is having trouble detecting the top faces for d8, d10s and d12s. The network often detects empty spaces as an object, or misses dice entirely. The

classifier mostly has trouble with 5, 2, 11, 12 and 1. There is a major imbalance between the number of data for ‘1’ and ‘12’, as all dice have a ‘1’ face, but only a d12 has a ‘12’ face. The network performs well on large, zoomed images of single dice, but has trouble with zoomed out pictures of multiple dice. Our dataset was small, and general, with images taken on different backgrounds such as carpet, a black table, a wooden table, a chessboard etc. This amount of variation between training images could mean that the network has trouble identifying features of the dice.

Improvements and Extensions

Since we didn’t have a proper dataset for both dice detection and dice numbers and had to generate our own, more data from different environments could increase the accuracy of our program. We can also increase accuracy by controlling the environment around the dice by rolling on a dice tray.

Taking a photo, moving it from your phone to the root folder and changing the image path within the program is a tedious process and can be slower than just counting it up yourself. Instead, we can use a webcam to capture video, feed each frame through the network and display it in real time. The inference time for object detection and classification take less than a second, with most of the overhead from displaying an image using matplotlib. Hence, we can create a rig, with a camera pointed towards a dice tray to maximise accuracy.

Conclusion

The network has major problems detecting dice and sometimes detects non-dice as dice. The classifier has average performance and has trouble identifying 5, 2, 1,11 and 12. The poor performance of the network can be attributed to small and poor training data. More specific data is needed, possibly in one location against a single background. A possible extension to this project would be to integrate a webcam and video feed to get real time detection and classification.

References

[1] https://pjreddie.com/darknet/yolo/

[2] https://towardsdatascience.com/training-yolo-for-object-detection-in-pytorch-with-your-custom-dataset-the-simple-way-1aa6f56cf7d9

[3] https://www.kaggle.com/ucffool/dice-d4-d6-d8-d10-d12-d20-images

[4] https://towardsdatascience.com/object-detection-and-tracking-in-pytorch-b3cf1a696a98

[5] https://towardsdatascience.com/a-two-stage-stage-approach-to-counting-dice-values-with-tensorflow-and-pytorch-e5620e5fa0a3

[6] https://towardsdatascience.com/object-detection-and-tracking-in-pytorch-b3cf1a696a98

[7] https://towardsdatascience.com/calculating-d-d-damage-with-tensorflow-88db84604f0a