EE 456- Artificial Neural Networks

12/15/2022

Sameer Khan, Allen Reed

Final Project- Object Detection

**Abstract:**

For this project, we challenged ourselves to create an object detection convolutional neural network. One of the most interesting and challenging aspects of this project was getting the neural network to predict the bounding boxes for the detected images. The goal of this project was to be able to detect vehicles in a video of a road with traffic, which it was able to do successfully.

**Outline:**

Creating an object detection convolutional neural network was pursued to obtain a better understanding of object detection used in applications such as autonomous vehicles. Our goal was to detect vehicles, bicycles, people, etc. so we could apply this to a video of a road with traffic and would be able to make predictions similar to how autonomous vehicles make detections.

In this project, TensorFlow and Keras were used for the low-level neural network functions. There were several attempts to create this neural network, however the script that ended up working was based on code written by Srihari Humbarwadi, which implemented a one-stage object detection model known as RetinaNet. Our previous attempts to complete this project was using the YOLO architecture; however, RetinaNet is simpler to implement within Keras and also runs faster. The RetinaNet model was trained using the popular COCO (2017) dataset.

**Discussion:**

First attempt - YOLO architecture:

As mentioned, a YOLOv1 architecture was originally planned and attempted. The YOLO script was written using TensorFlow and Keras, plus was based on many different examples and sources. Most of the examples we based our code from had become outdated, which made the code between examples and libraries difficult to integrate with each other (which is why the YOLO script didn’t succeed). Certain parts of the script came from a tutorial made by Vivek Maskara, such as model-related functions, loss, and other sections. Most of the higher-level code for the neural network script was either custom written or changed significantly from the original source. The code to load in the dataset was later changed and taken from the second attempt (the section below this), where it uses dataset VOC 2007.

The output of the current code for attempt one, using YOLO, is showed in figure 1. The official YOLOv1 architecture is showed in figure 2, so comparing our model to the official architecture is easier. The main difference between models is the yolo\_reshape function, which was written by Vivek Maskara and leveraged for our project.

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

Layer (type) Output Shape

Param #

=================================================================

conv2d (Conv2D) (None, 448, 448, 64) 9472

max\_pooling2d (MaxPooling2D (None, 224, 224, 64) 0)

conv2d\_1 (Conv2D) (None, 224, 224, 192) 110784

max\_pooling2d\_1 (MaxPooling (None, 112, 112, 192) 0

2D)

conv2d\_2 (Conv2D) (None, 112, 112, 128) 24704

conv2d\_3 (Conv2D) (None, 112, 112, 256) 295168

conv2d\_4 (Conv2D) (None, 112, 112, 256) 65792

conv2d\_5 (Conv2D) (None, 112, 112, 512) 1180160

max\_pooling2d\_2 (MaxPooling (None, 56, 56, 512) 0

2D)

conv2d\_6 (Conv2D) (None, 56, 56, 256) 131328

conv2d\_7 (Conv2D) (None, 56, 56, 512) 1180160

conv2d\_8 (Conv2D) (None, 56, 56, 256) 131328

conv2d\_9 (Conv2D) (None, 56, 56, 512) 1180160

conv2d\_10 (Conv2D) (None, 56, 56, 256) 131328

conv2d\_11 (Conv2D) (None, 56, 56, 512) 1180160

conv2d\_12 (Conv2D) (None, 56, 56, 256) 131328

conv2d\_13 (Conv2D) (None, 56, 56, 512) 1180160

conv2d\_14 (Conv2D) (None, 56, 56, 512) 262656

conv2d\_15 (Conv2D) (None, 56, 56, 1024) 4719616

max\_pooling2d\_3 (MaxPooling (None, 28, 28, 1024) 0

2D)

conv2d\_16 (Conv2D) (None, 28, 28, 512) 524800

conv2d\_17 (Conv2D) (None, 28, 28, 1024) 4719616

conv2d\_18 (Conv2D) (None, 28, 28, 512) 524800

conv2d\_19 (Conv2D) (None, 28, 28, 1024) 4719616

conv2d\_20 (Conv2D) (None, 28, 28, 1024) 9438208

conv2d\_21 (Conv2D) (None, 14, 14, 1024) 9438208

conv2d\_22 (Conv2D) (None, 12, 12, 1024) 9438208

conv2d\_23 (Conv2D) (None, 10, 10, 1024) 9438208

flatten (Flatten) (None, 102400) 0

dense (Dense) (None, 512) 52429312

dense\_1 (Dense) (None, 1024) 525312

dropout (Dropout) (None, 1024) 0

dense\_2 (Dense) (None, 1470) 1506750

yolo\_\_reshape (Yolo\_Reshape (None, 7, 7, 30) 0

Figure 1. Output of YOLO attempt’s model summary

Diagram, engineering drawing

Description automatically generatedFigure 2. YOLOv1 model architecture

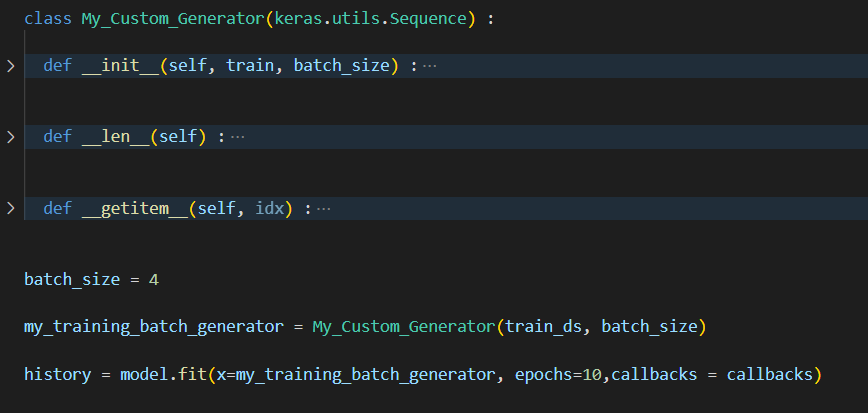


Figure 3. Code snipped of YOLO attempt

Figure 2 shows a section of the code from the YOLO attempt. To have a sequential model be fit with data of images and labels, a class (such as My\_Custom\_Generator, which Vivek Maskara used in his example) must be used for getting the data to be correctly retrieved by model.fit. The original function in the example was a lot different due to the dataset structure, so this function had to be rewritten for our case but is also the function that is causing the YOLO model to not work. The data base automatically batches the images and labels together, but TensorFlow needs the images and labels to be batched separately. We attempted to fix the batching issue; however, errors kept appearing that could not be easily explained or solved, due to the way the data was being batched.

Second Attempt – RetinaNet:  
We then tried to get RetinaNet to work, basing the script off a script created by Luke Wood, which uses KerasCV, however the libraries used were incompatible. Some of the incompatibility issues were able to be fixed or worked around, but other issues were too complicated to solve.

Diagram

Description automatically generated

Figure 4. RetinaNet architecture

Figure 4 shows the structure of the RetinaNet. It is comprised of four main parts: the bottom-up pathway (ResNet), feature pyramid, classification subnetwork and regression subnetwork. First the residual network backbone, the main feature extractor the RetinaNet is built on, extracts features from the images regardless of size of the image. Then these feature maps are up sampled and grouped among images of the same size and merged. Next, they are passed through to the classification network that computes probabilities that a classed object could be at any area in the image and outputs a probability map of the entire image. Simultaneously, the regression subnetwork does a similar calculation of the bounding boxes but outputs parameters of the size of the bounding boxes instead. Once all these calculations are made, a loss can be acquired from the predictions and backpropagated through the network.

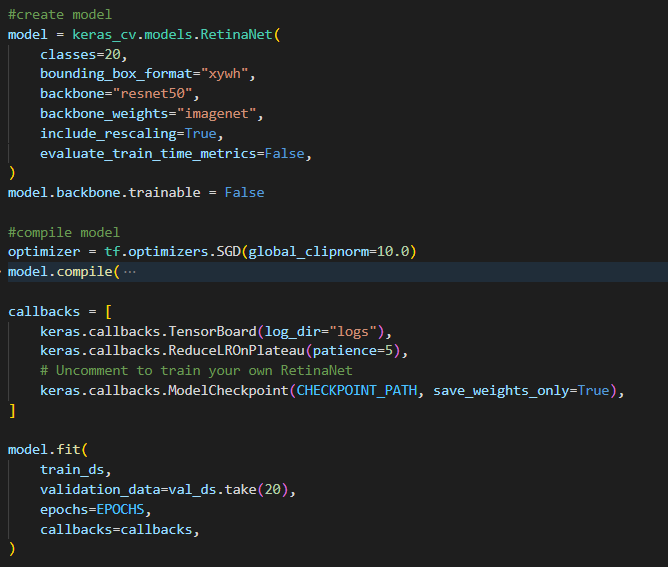


Figure 5. Code snippet of RetinaNet first attempt

Figure 5 shows our first attempt at a RetinaNet. Logically, the code seemed to be sound as it used the format and functions of a similar example. We start by creating the RetinaNet model with information on the backbone (ResNet), classes, bounding box format information, and more. Then we compile the model with scenarios for early stopping criteria. Finally, we try to fit the model to the VOC 2007 data set. Unfortunately, some of the functions used in the KerasCV library no longer were compatible with new version of TensorFlow API changes and made debugging the training of the model impossible therefore we had to scrap this attempt.

Final Attempt - RetinaNet

The script based mainly from a network made by the Keras-io team was created next. This neural network uses RetinaNet and the COCO 2017 dataset. In an effort to make the code as easy as other aspects of TensorFlow, we opted to borrow some functions from the Keras example GitHub to get us moving and implemented them to our liking. This would include functions that would help us load the data, process the data into a more friendly format, create building blocks of the model, and help implement the model so we could expedite the process after so many failed attempts and hours.

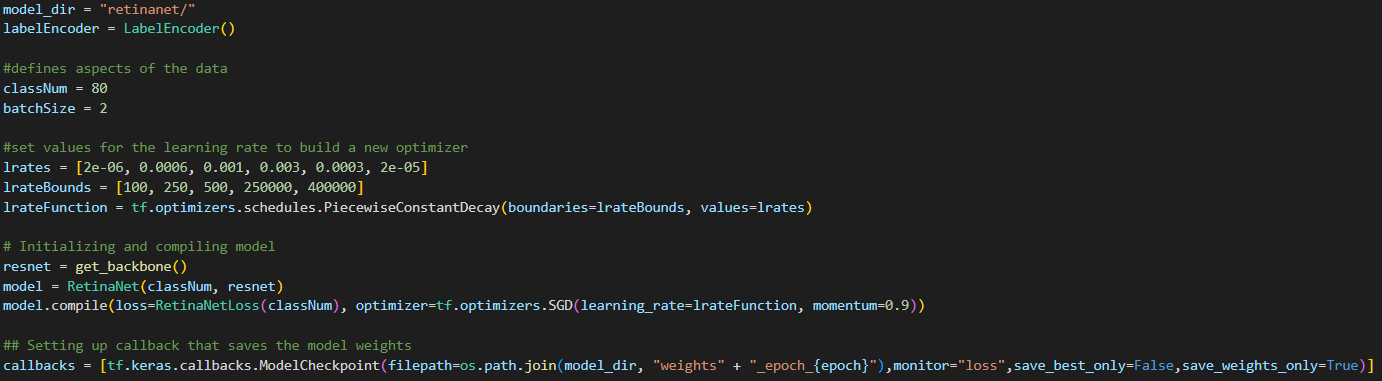


Figure 6. Code snippet of model prepping and data loading

Figure 6 shows a blend of the code we borrowed and how we implemented it. The top lines are an arbitrary set of parameters for training the model including things like the size of batches and learning rates. We decided to borrow this data from the source code as it would have taken longer to fine tune these arguments ourselves and are mostly arbitrary to the actually running of the model.

The next couple lines initialize the backbone of the model which is a feature extractor model that does the duty of taking images and extracting their important features. We chose to use a residual network because it was a widely used method of getting good results. Then we create the model, a RetinaNet, and compile it using the parameters and functions defined above.

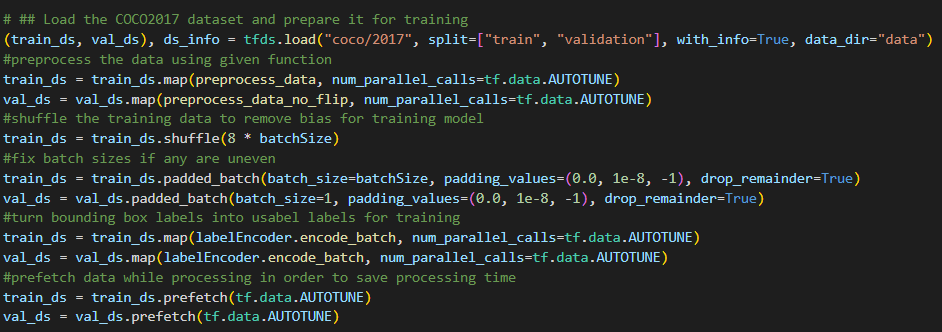


Figure 7. Code snippet of data preprocessing

Figure 7 shows a snippet of code that preprocesses the loaded data before we pass it through to train the model. In order, we first load the data and call the preprocess function which extracts the bounding boxes, image, and class from each data sample, performs an augmentation like flipping the image, resizes the image to a normalized pixel area, and fixes the bounding box format. After, we shuffle the data to remove any bias from the formatting of the dataset, make sure each batch is exactly its batch length, encode the batch and then prefetch it so it is ready to be used. A similar process happens to the validation set except shuffling is not necessary.

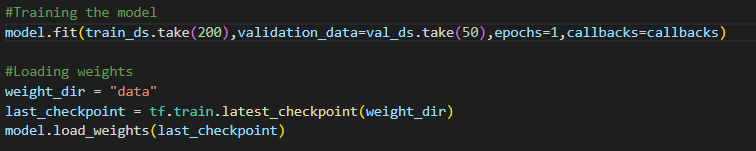


Figure 8. Code snippet of training and saving the model

After the data has been processed, we can finally train the compiled model. As you can see in Figure 8, we decided since the model has an efficient backbone, we do not need to run the entire dataset and instead ran about 200 images with multiple bounding boxes to help finish the training of the model. Afterwards we save the weights and can load them without retraining the model every run. The results were very good on practice validation images, so we moved on to using a video.



Figure 9. Code snippet showing the process for evaluating the video

Since the detection of this neural network was tested and proven to be a success, we now attempted to achieve our goal of detecting vehicles in a video of a road with traffic. The video was found online on video and loaded into our neural network script using OpenCV’s Video Capture. Each frame is sent to the visualization function for drawing the bounding boxes and labels. Inside the visualization function, the OpenCV image is converted to a Matplotlib pyplot figure. The pyplot figure couldn’t be updated using the typical draw() function in Matplotlib, so the figure was saved to a jpg file and then opened/displayed using OpenCV. The video is very slow, due to the prediction times being longer than expected, so we also save the video to a mp4 for later viewing. The code written to display and to create the video is shown in Figure 9.

**Results and Conclusion:**

We first tested our model by evaluating its accuracy on some test images form the validation test set.

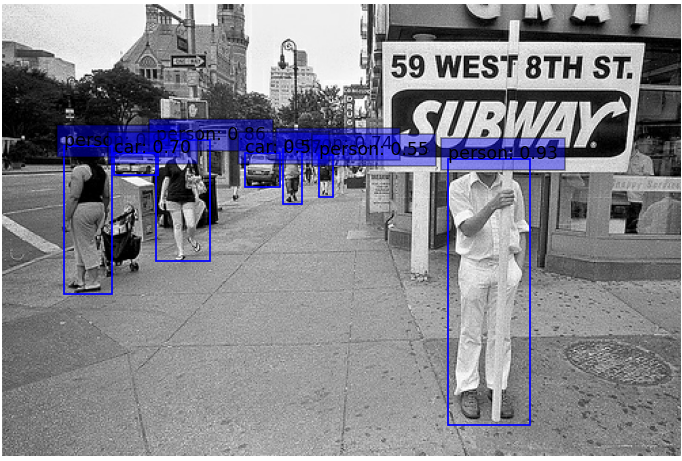
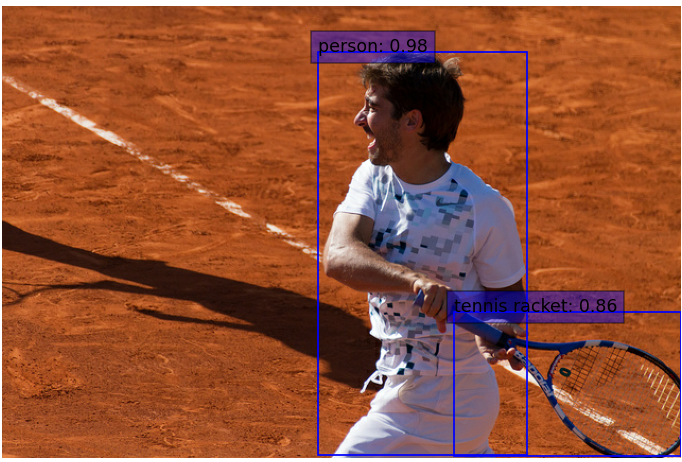
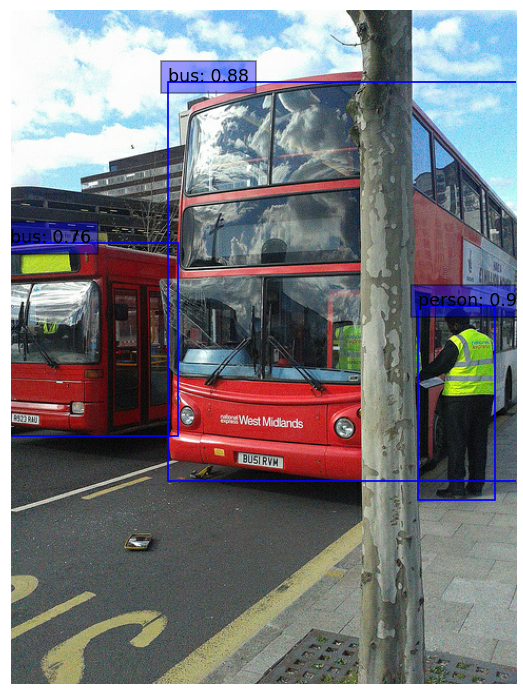


Figure 10,11,12. Validation test set images that have predicted bounding boxes with class and confidence percentage.

As you can see in the above images, the net predicted images very well. It was especially good at detecting people have above a 90% confidence rate in all images for at least one person and having the ability to detect every person in the third image. It also shows expertise in detecting vehicles. Figure 10 shows its ability to detect a bus while Figure 12 shows its ability to detect cars even when far away and small in an image.

Once we had determined the model was good enough, we moved on to testing the model on a separate video not from the COCO data set.

We found a video on Videvo.com showing a large highway with many types of vehicles to detect and would act as a good metric to see if the model would work well for autonomous cars.

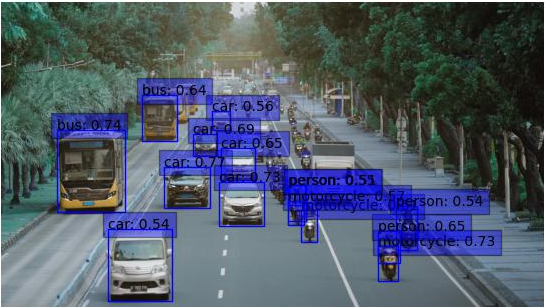


Figure 13. Frame from the highway video with predicted bounding boxes and confidence percentages.

Figure 13 shows 1 of almost 300 frames of the video that were evaluated. As you can see, the model was able to correctly identify nearly every object that it did detect. This included cars, buses, motorcycles, and people on motorcycles. In that regard, this was a great success. One point of failure are the objects that were not recognized at all like the many motorcycles on the right side of the image. This could be due to the data set not having many examples of small motorcycles or the clustering of the motorcycles not looking like many objects but just a single blob.

In general, we tested the accuracy of our model by the correctness of the classification and the confidence rate of the associated classification. A correct classification with high confidence is the best result. This model was tested using 100 training images initially and proved to work well on the validation data set images in Figures 8,9, and 10 but seemed to misclassify more objects on the video. One example was mistaking the buses on the left for trains. After training the model with 200 images, the results were more like Figure 11 with correct classifications and higher confidence rates. A fully predicted video of the test highway video can be found in the submission labeled output\_video.avi as well as a folder with all image frames with prediction values.

**Summary:**

The goal of this project was to make an object detection neural network which predicted bounding boxes around what the network detects. There were a couple failed attempts with YOLO and RetinaNet; however, we were eventually able to get the neural network to operate correctly by giving another attempt with RetinaNet. The desired result was a video of traffic on a road with detections of the vehicles, which our final attempt with RetinaNet was able to output successfully. The RetinaNet was trained on the COCO dataset, so the network can detect many different objects, not just detect vehicles. There were some errors with incorrect detections in the output video; however, that is to be expected in an object detection neural network.

**Contributions:**

Sameer worked primarily on finding functions to create the model and implementing them in TensorFlow. This includes training the model.

Allen worked primarily on the code that tests the model and including the code that evaluates the video and images.

All but two functions/classes were borrowed from Keras team GitHub in order to allow us to leverage the abilities of TensorFlow. Most of these functions are predefined in the API but were not compatible with the current version of TensorFlow so they had to be hard coded into the file to allow for proper usage. Attempt 2 of the ResNet shows how we tried to use similar functions to attempt the project. Most code after the functions is original code that follows function usage examples from Keras.

**Bibliography:**

Stock footage provided by Videvo, downloaded from videvo.net

Attempt 1 Code reference: Vivek Maskara, <https://www.maskaravivek.com/post/yolov1/>

Figure 2. YOLOv1 paper, Joseph Redmon and others, <https://arxiv.org/pdf/1506.02640.pdf>

Figure 4. Focal Loss paper, Tsung-Yi Lin and others, <https://arxiv.org/pdf/1708.02002.pdf>

Attempt 2 Code reference: Luke Wood, <https://github.com/keras-team/keras-cv/blob/master/keras_cv/datasets/pascal_voc>

Final code reference: Keras-io team, <https://github.com/keras-team/keras-io/blob/master/examples/vision/retinanet.py>