Andrew ID: ckwizera 29/03/2024

ACV Homework 5

In this assignment, I'm diving into the world of object detection with the YOLOv3 model[1]. I'll be using the PASCAL_VOC dataset, which has a bunch of different objects of 20 classes. Plus, I'm adding our own dataset for detecting faces. The goal? Getting the model to spot faces accurately. And to top it all off, I'll be showing off the model live in action using OpenCV, making it all feel real and interactive.

Explanations of YOLOV3 object detection model utility functions:

(a) def iou(box1, box2, is pred = True)

The **iou** function calculates the Intersection over Union (IoU) score between two bounding boxes. The IoU score is a measure of the overlap between two bounding boxes. It is calculated as the area of the intersection of the boxes divided by the area of their union.

The function takes three arguments:

- **box1** and **box2**: These are the two bounding boxes for which the IoU score is to be calculated. Each box is represented as a tensor with four elements: [x, y, width, height]. Here, (x, y) is the center of the box, and width and height are the dimensions of the box.

- **is_pred**: This is a boolean flag that determines how the IoU score is calculated. If **is_pred** is True, the function assumes that **box1** is a predicted bounding box and **box2** is a ground truth bounding box. If `is_pred` is False, the function calculates the IoU score based solely on the width and height of the boxes.

The function works in two modes:

1. When **is_pred** is True: The function first converts the center coordinates and dimensions of the boxes to the coordinates of their top-left and bottom-right corners. It then calculates the coordinates of the intersection of the boxes and ensures that the intersection is at least 0. The area of the intersection is calculated as the product of its width and height. The function also calculates the areas of **box1** and **box2** and their union area. The IoU score is then calculated as the area of the intersection divided by

the area of the union. A small epsilon value is added to the denominator to prevent division by zero.

2. When **is_pred** is False: The function calculates the area of the intersection as the product of the minimum width and height of the boxes. It also calculates the areas of **box1** and **box2** and their union area. The IoU score is then calculated as the area of the intersection divided by the area of the union.

The function returns the IoU score, which is a value between 0 and 1. A score of 0 indicates no overlap between the boxes, and a score of 1 indicates a perfect overlap.

(b) def nms(bboxes, iou threshold, threshold):

Certainly! Here's an improved explanation:

In the task of object detection, Non-Maximum Suppression (NMS) serves as a crucial post-processing technique, designed to refine detection results by eliminating redundant bounding boxes and retaining only the most pertinent ones. By doing so, NMS effectively reduces computational complexity and minimizes false positives.

Here's a detailed breakdown of how the function operates:

1. **Initial Filtering**: The process begins by filtering out bounding boxes that fall below a specified confidence threshold. This step ensures that only detections with sufficient confidence are considered for further analysis.

2. **Confidence-Based Sorting**: Subsequently, the remaining bounding boxes are sorted in descending order based on their confidence scores. This ensures that detections with higher confidence levels are prioritized during subsequent processing steps.

3. **Iterative Selection**: Moving forward, the function initializes a list to hold the final selection of bounding boxes. It then iterates through the sorted list of bounding boxes. At each iteration, it selects the top-ranked bounding box and examines its overlap with other remaining boxes. If the boxes do not significantly overlap or if the first bounding box exhibits notably higher confidence, the second bounding box is added to the final selection.

4. **Output Generation**: Finally, the function returns the refined list of bounding boxes, representing the most relevant and non-overlapping detections.

(c) class YOLOv3(nn.Module):

This class defines an object detection architecture with configurable parameters for the number of input channels (defaulted to 3) and the number of classes to detect. Incorporating both CNN blocks and residual blocks, it effectively stacks multiple layers while mitigating gradient vanishing through the use of LeakyReLU activation functions. Additionally, the architecture employs ScalePrediction layers for accurate bounding box predictions and incorporates upsampling techniques to preserve and enhance feature representations.

Below is the structure of YOLO3 as presented in the paper[1]

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

(d) ScalePrediction module:

The ScalePrediction class defines a module that generates predictions for object detection at different scales within an input image. It takes in the number of input channels and the number of object classes as parameters. The module consists of convolutional layers, batch

normalization, and a leaky ReLU activation function. During the forward pass, the input tensor passes through these layers to produce an output tensor. This output tensor contains predictions organized in a grid, with each grid cell containing bounding box coordinates, objectness scores, and class probabilities for multiple anchor boxes. The dimensions of the output are reshaped and permuted to be compatible with the rest of the object detection architecture. Essentially, ScalePrediction generates the final predictions for detected objects at various scales in the input.

(e) ANCHORS
The anchor box values represent predefined bounding box shapes that serve as references for the model to predict object locations. They are chosen to cover different object scales and aspect ratios in the training data.

The given values are organized into three lists, each representing anchor boxes for a specific scale:

1. `[(0.28, 0.22), (0.38, 0.48), (0.9, 0.78)]`: Large anchor boxes for capturing larger objects.
2. `[(0.07, 0.15), (0.15, 0.11), (0.14, 0.29)]`: Medium anchor boxes for medium-sized objects.
3. `[(0.02, 0.03), (0.04, 0.07), (0.08, 0.06)]`: Small anchor boxes for smaller objects.

The values represent width and height ratios relative to the input image dimensions. For example, (0.9, 0.78) corresponds to an anchor box with 90% width and 78% height of the image.

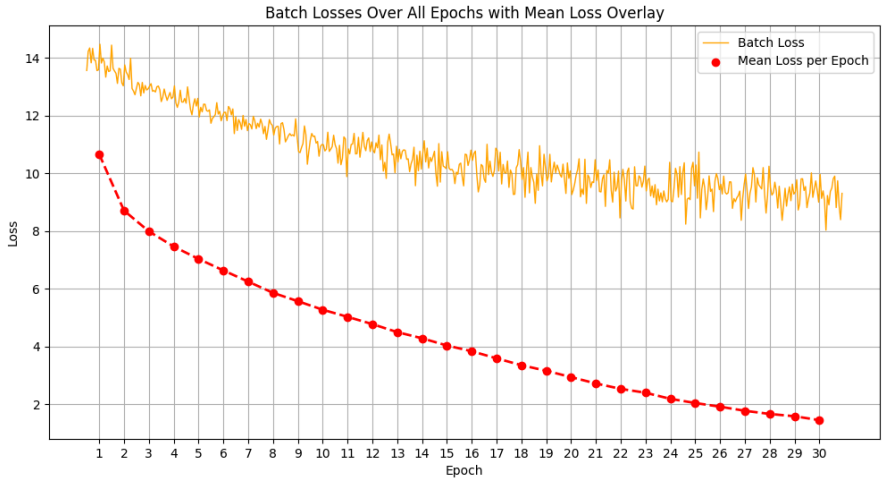
During training, the model learns to predict offsets from these anchors, objectness scores, and class probabilities for each anchor box, enabling accurate object detection for various scales and shapes.

(f) What happens in the training loop ?

The training loop function facilitates the training process by iterating through the dataset using a provided data loader. For each batch of data, it ensures both the input data (`x`) and target labels (`y`) are sent to the appropriate device, typically a GPU, to ensure efficient computation and avoid device conflicts. Utilizing `torch.cuda.amp.autocast()`, the function employs automatic mixed precision training, which optimizes training speed by scaling values from 32 to 16 bits. Within the autocast context, the model generates predictions (`outputs`) for the input data. Subsequently, the function calculates the loss for each scale of the YOLOv3 model by comparing the predicted outputs with the ground truth labels (`y`) using the provided loss function and scaled anchor boxes. These losses are appended to a list for monitoring the training progress. After resetting the gradients, the function performs backpropagation of the loss through the model's parameters, leveraging the `scaler` object to scale the loss. The optimizer then updates the model parameters based on the computed gradients, and the scaler is updated for the next iteration. Throughout the training loop, a progress bar visually tracks the

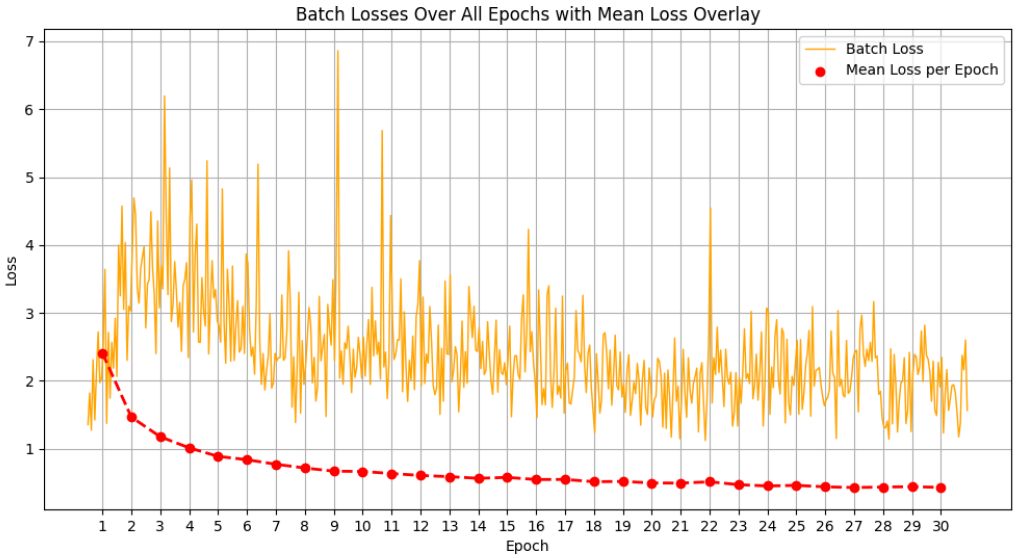
mean loss across batches, providing real-time feedback on the model's training performance. Finally, the function returns a list of losses and the mean loss for further analysis.

The following is the losses graph after training 30 epochs of the PASCAL_VOC datasets.



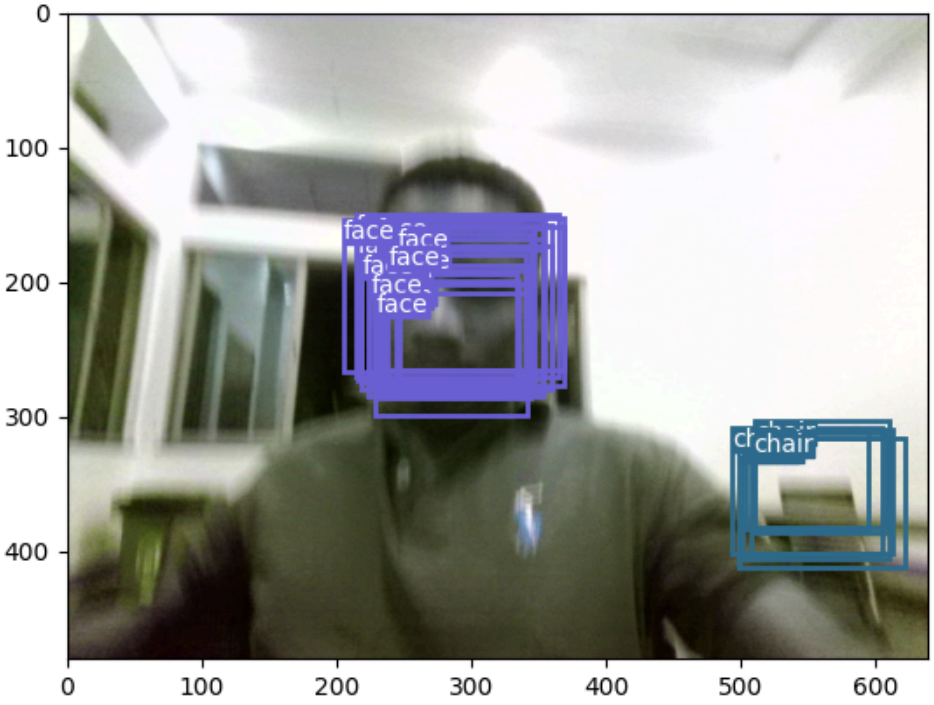
The above figure shows the change of model loss in the training process over 30 epochs.

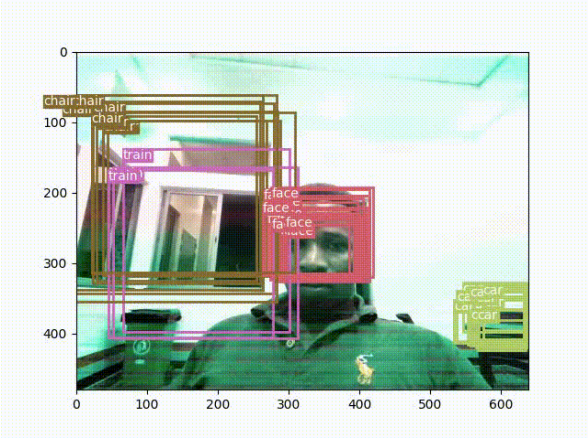
The following figure shows the model training loss after adding the new class (face) starting from other saved checkpoints:

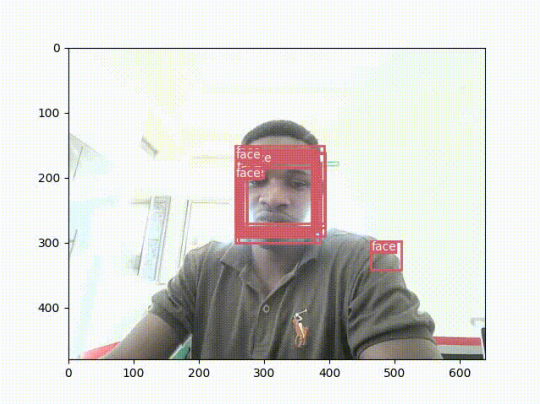


It can be inferred that the model got too much noise in batch loss. Due to the new dataset I have added which might not be well distributed.

Bonus: To increase the model accuracy in predicting faces, I removed the person class. As the model was biased on person class due to many persons compared to our few faces class. Even if I removed the person class, the model was still inaccurate in detection of faces. Due to that case, I added a new annotated face dataset from /kaggle/input/face-detection-dataset which are publicly available. The following is the result of the model in action.







The above demonstration shows how good the model has become after training 30 epochs with the additional dataset of faces.

The best checkpoints are located here

References:

[1] J. Redmon and A. Farhadi, ‘YOLOv3: An Incremental Improvement’. arXiv, Apr. 08, 2018. doi: 10.48550/arXiv.1804.02767.