FINAL-REPORT: You Only Look Once: Unified, Real-Time Object Detection

SongSeungWu

1 WHAT IS YOLO?

The YOLO (You Only Look Once) algorithm is a unified, real-time object detection model that allows for efficient and accurate object detection in a single forward pass. This efficiency makes YOLO suitable for applications where speed is critical, such as in real-time video analysis. The architecture divides the input image into an S×S grid of cells, where each cell predicts:

- B bounding boxes with coordinates (x, y, w, h)
- A confidence score indicating the likelihood of object presence
- Class probabilities for each detected object

For our model:

- S = 7: The grid size if 7×7
- B = 2: Each grid cell predicts 2 bounding boxes
- C = 20: There are 20 classes, implying each cell outputs 30 values

2 YOLO DATASET CLASS IMPLEMENTATION

The YoloDataset class is designed to handle data preprocessing for YOLO model training, preparing images and bounding box information in a format suitable for YOLO's output structure. The main functions of this class are as follows:

1. Data Loading and Initialization
    - Loads image paths, bounding box coordinates, and labels from a list_file.
    - Defines transformations such as ColorJitter and Resize to apply data augmentation.
2. Image Augmentation
    - Applies various augmentations, including random horizontal flipping, scaling, translation, and cropping, to generate diverse training samples.
    - Updates bounding box coordinates to match each transformation.
3. Target Generation
    - Converts bounding box information into a 7x7 grid format suitable for YOLO.
    - Calculates each grid cell's center coordinates, width, height, and sets bounding box and class label information according to the YOLO format.
4. Color and Blur Transformations
    - Applies random brightness, saturation, hue adjustments, and blur, enhancing image diversity to improve the model's robustness to varying image conditions.

This class provides optimized data preprocessing for YOLO model training, automatically handling image augmentation and target generation to prepare efficient and diverse training data.

3 YOLO NETWORK IMPLEMENTATION

The YOLO model built for this implementation consists of:

- 22 Convolutional Layers: These layers extract spatial features from the input images, with the first and 20th layers using a stride of 2 to halve the output size. The architecture uses four max pooling layers to down-sample feature maps.
- 2 Fully Connected Layers: These layers condense spatial information to predict bounding box coordinates, object presence confidence, and class probabilities.
- Leaky ReLU Activation: Applied to all layers except the final layer, with a slope of 0.1 for negative values, allowing some gradient flow even in negative regions.

4 YOLO LOSS FUNCTION IMPLEMENTATION

The yoloLoss class provides a comprehensive loss function tailored for YOLO model training, calculating multiple loss components to optimize object detection performance. This function compares the YOLO model's predictions to the target values, accounting for errors in position, size, class, and object presence. Key Components:

1. Classification Loss
    - Calculates classification loss by comparing predicted and actual classes for cells containing objects.
    - This loss encourages the model to correctly classify detected objects by computing the squared error only in cells where an object is present.
2. No-Object Loss
    - Penalizes high confidence scores in cells where no objects are present.
    - This loss helps the model learn to avoid falsely predicting objects in empty cells by reducing the confidence score for non-object cells.
3. Object Loss
    - Calculates losses related to bounding box center coordinates (x, y), dimensions (w, h), and confidence score in cells containing objects.
    - For each object-containing cell, two bounding boxes are predicted, and the one with the highest IoU with the ground truth is designated as the responsible bounding box.
    - The width and height loss calculations use a square root transformation to balance gradients for large and small boxes, emphasizing position and size accuracy for detected objects.
4. IoU Calculation
    - Computes the Intersection over Union (IoU) between predicted and ground truth boxes, used to determine the most accurate bounding box in the contain_obj_error function.

5 YOLO MODEL TRAINING IMPLEMENTATION

This code block outlines the entire process for training a YOLO model, including data loading, model and optimizer initialization, training loop, and learning rate adjustment.

1. Data Loading and Setup
    - Loads images from file_root and preprocesses them through the YoloDataset class, providing a train_loader for training.
    - The batch size is set to 10, data is shuffled randomly, and four worker processes are utilized.
2. Model and Device Setup
    - The YOLO model is instantiated, and pre-trained weights are loaded from state_dict.
    - If a CUDA device is available, the model is moved to the GPU to accelerate training.
3. Learning Rate and Optimizer Setup
    - Uses an SGD optimizer with an initial learning rate of 0.001, momentum of 0.9, and weight decay of 0.0005.
    - A ReduceLROnPlateau scheduler automatically reduces the learning rate by half if the loss stops improving, with updates printed to monitor learning rate changes.
4. Loss Function
    - Uses the yoloLoss function, which calculates losses for position, size, class, and object presence.
    - Settings of l_coord=5 and l_noobj=0.5 prioritize accurate bounding box coordinates and assign a relatively lower weight to the loss for non-object cells.
5. Training Loop
    - Trains the model over 15 epochs. For each epoch, it passes the input images through the model, computes predictions, and calculates the loss.
    - Initializes gradients with optimizer.zero_grad(), performs backpropagation with loss.backward(), and updates weights using optimizer.step().
    - Applies gradient clipping with a max norm of 5.0 to prevent gradient explosion.
    - Every 10 batches, it prints the current epoch, iteration, individual loss, and average loss to monitor training progress.
6. Learning Rate Scheduling
    - At the end of each epoch, scheduler.step() adjusts the learning rate based on the average epoch loss. If the loss stops improving, the learning rate is reduced to facilitate finer learning adjustments.

6 ISSUES AND SOLUTIONS IN THE TRAINING THE MODEL PROCESS

During the training process, an issue occurred where both the LOSS and AVERAGE_LOSS values were displaying as NaN. I identified that this issue was likely due to instability caused by high gradients or an inappropriate learning rate. To address this problem, I modified the training configuration as follows:

1. scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, verbose=True)
2. torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
3. scheduler.step(total_loss / len(train_loader))

This part is added to improve the stability of training and dynamically adjust the learning rate when the loss stops decreasing.

1. ReduceLROnPlateau Scheduler:
    - scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau automatically reduces the learning rate when the loss plateaus.
    - mode='min' monitors a decrease in loss, factor=0.5 reduces the learning rate by half, and patience=2 decreases the learning rate if there's no improvement for 2 epochs.
    - verbose=True provides updates whenever the learning rate is adjusted, allowing easier monitoring of the training process.
2. Gradient Clipping:
    - torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0) limits the gradient norm to a maximum of 5.0, preventing gradients from exploding.
    - This helps keep the training stable by restricting gradients that might otherwise become excessively large.
3. Scheduler Step:
    - scheduler.step(total_loss / len(train_loader)) adjusts the learning rate based on the average loss at the end of each epoch. By passing the epoch's average loss to the scheduler, the learning rate is reduced if the loss fails to improve.

7 PERFORMANCE IMPROVEMENT MEASURES

To enhance the performance of the YOLO model, the following changes were made:

1. Learning Rate Adjustment
    - The initial learning rate was increased from 1e-4 to 1e-3.
2. Optimization of the Loss Function Implementation
    - In compute_prob_error, the contain tensor was modified to match the specific target dimensions more closely. By adjusting contain = contain.unsqueeze(3).expand_as(target[:,:,:,10:]), we limited the expansion to only the required size [Batch_size, 7, 7, 30] rather than expanding across the entire tensor. This adjustment reduces memory usage and accelerates computation, optimizing the classification error calculation.
    - In not_contain_obj_error, the same dimensional optimization was applied, where not_contain = not_contain.unsqueeze(3).expand_as(:, :, [4, 9]]) was used to match the exact size needed for no-object confidence prediction. This adjustment reduces redundant calculations for cells where no objects are present.

These changes improve efficiency in calculating the loss by optimizing the memory footprint and computational load, ultimately enhancing the model's overall performance and convergence stability.

8 TRAINING RESULTS

After initializing the pre-trained weights (pre_train_complete), the YOLO model was trained for 15 epochs. Each epoch showed a significant reduction in both individual batch loss and average loss across iterations, indicating effective learning and convergence of the model. The key results per epoch include:

1. Initial Epoch (Epoch 1)
    - The initial average loss started high, at approximately 259.0364 after the first 10 iterations.

- As the training progressed within the first epoch, the average loss steadily decreased, reaching approximately 13.7393 after 230 iterations, demonstrating a rapid drop in loss during the early stages of training.

2. Final Epoch (Epoch 15)

   - By the last epoch, the model had largely stabilized with a low loss, indicating convergence.
   - The average loss by the end of training reached approximately 0.5285

9 INFERENCE(MAP) RESULTS

To evaluate the performance of the YOLO model, we calculated the mean Average Precision (mAP). Performace Results:

1. aeroplane: 63.96%
2. bicycle: 67.67%
3. bus: 61.95%
4. car: 70.21%
5. cat: 74.09%
6. dog: 77.07%
7. map: 69.16%

This result, with an mAP close to 70%,

10 CONCLUSION

In this project, we implemented an object detection system using the YOLO model and validated its effectiveness through training and performance evaluation. We applied several performance improvement measures, such as learning rate adjustments and loss function optimizations, which enhanced the model's convergence speed and stability. The final mAP achieved was 69.16%, with some object classes exceeding 70% accuracy.