

Multi-instance Object Detection in Food Images

By Tejas Rawal



Introduction

For this challenge, we were tasked with building a computer vision model capable of detecting food items given an image of food. The ability to recognize food from images can be a valuable tool for enhancing dietary awareness and facilitating more precise assessments of nutritional habits. Traditional methodologies, such as food frequency questionnaires, have long been known for their imprecise nature in capturing accurate dietary information within clinical studies.

This task could be boiled down to a multi-instance segmentations and detection problem. As a group, we chose to divide up the project work by each training and evaluating a separate model to detect food items in the images. In the end, we compared results and determined which approached worked the best. For my contribution, I traversed two different paths: on one, I attempted to create a model from the ground up and on the other, I utilized the Faster R-CNN architecture and complimentary pretrained models provided by PyTorch.

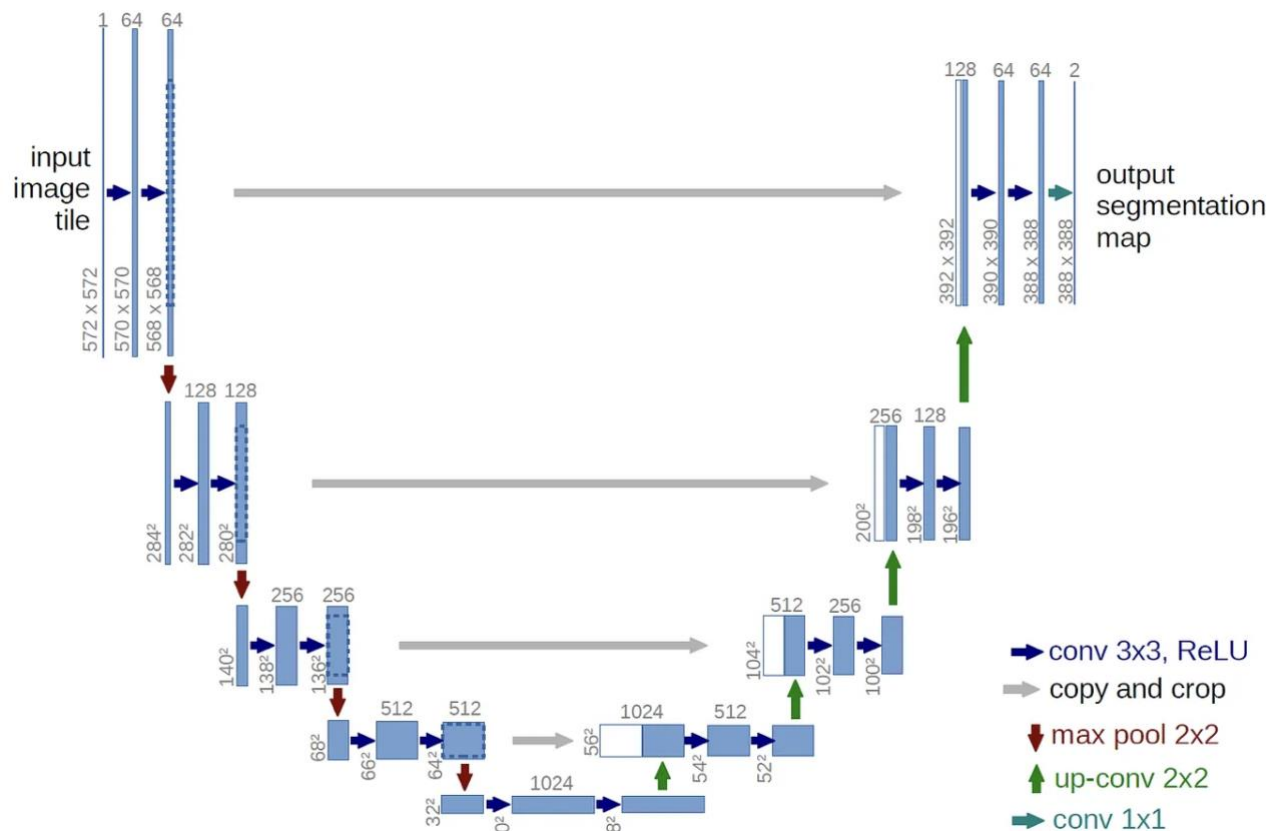
The dataset was provided with a COCO-style annotations format. COCO (or Common Objects in Context) is a large-scale object detection, segmentation, and captioning dataset widely used

throughout the industry as a benchmark for evaluating state-of-the-art model architectures design for object detection and segmentation.

My Contributions

Building a custom model

To understand the problem deeper and give myself a challenge, I decided to try and build a custom model using blocks of Convolution and Linear layers. For the backbone feature extractor segment of this model, I chose to use the U-Net architecture (Ronnenberger et al., 2015). This architecture consists of an encoder portion for down sampling images and a decoder portion for up sampling them. Both “halves” of the network are interconnected by skip connections which bridge the encoder’s feature maps directly to the decoder, ensuring that minute details from the input are retained in the output and helping to mitigate the vanishing gradient problem during training.



Once this backbone was constructed, its output was connected to three different heads:

- A bounding box head consisting of a sequential linear layer with the output features equal to $4 * \text{the number of classes}$. This layer would output the bounding box coordinate positions.
- A mask head consisting of a single 2D convolution layer with the output channels equal to the number of classes. This layer would output the binary segmentation masks.
- A classification head consisting of a sequential linear layer with the output features equal to the number of classes. This layer would output the class predictions for each image.

```
class CustomMultiLabelUNet(nn.Module):
    def __init__(self, output_classes):
        super().__init__()

        self.backbone = CustomUNet(input_channels=CHANNELS)
        self.bbox_head = BboxHead(in_features=64, num_classes=output_classes)
        self.class_head = ClassHead(input_features=64, num_classes=output_classes)
        self.mask_head = MaskHead(input_ch=64, output_ch=output_classes)

    def forward(self, x):
        features = self.backbone(x)

        # bbox prediction
        bbox_pred = self.bbox_head(features)

        # multi-label classification
        class_logits = self.class_head(features)

        # mask prediction
        mask_pred = self.mask_head(features)

        return {
            "boxes": bbox_pred,
            "labels": class_logits,
            "masks": mask_pred
        }
```

For the loss criterion, I combined the losses of each output layer by summing up their values.

```
def build_custom_criterion(output, target):
    bounding_box_pred, class_logits, mask_pred = output["boxes"], output["labels"], output["masks"]

    # Bounding box loss using Smooth L1 Loss
    bounding_box_loss = F.smooth_l1_loss(bounding_box_pred, target["boxes"])

    # Multi-label classification loss using Cross Entropy Loss
    class_loss = F.binary_cross_entropy_with_logits(class_logits, target["labels"])

    # Mask loss using Binary Cross Entropy
    mask_loss = F.binary_cross_entropy_with_logits(mask_pred, target["masks"])

    # Combine losses
    total_loss = bounding_box_loss + class_loss + mask_loss

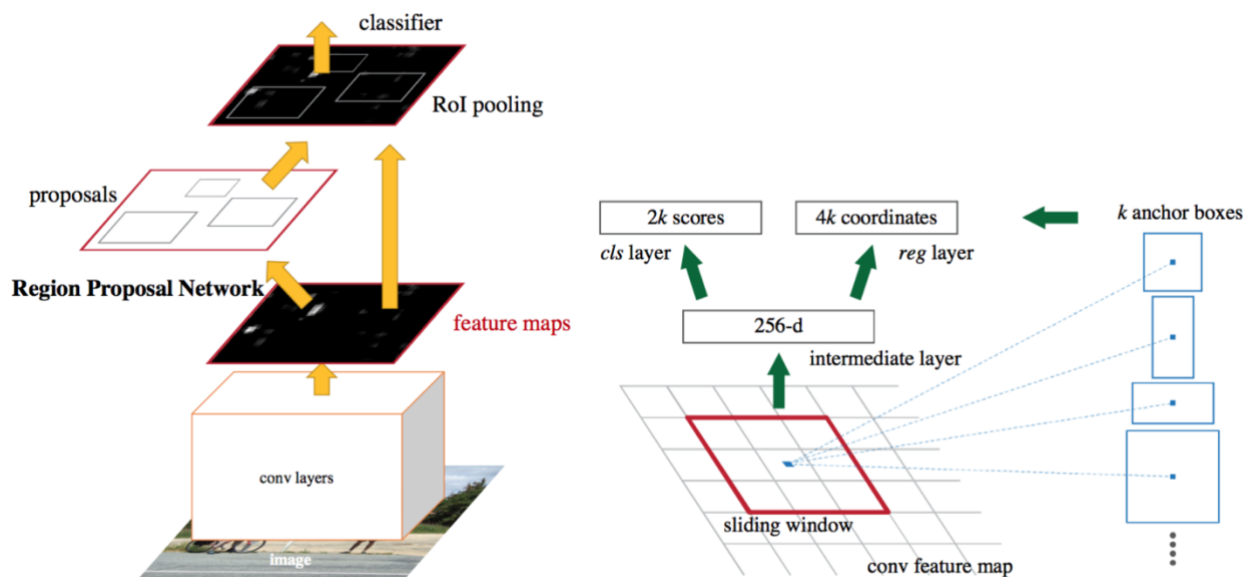
    return total_loss
```

I quickly realized that this was a very naïve approach as the model did not output any predictions and the loss values had a very high frequency across the epochs. Although I could never get a working model, I came out of this exercise with a better understanding of the different components that factor into object detection/image segmentation and how they all fit together.

Faster R-CNN


To move forward with my objective, I decided on implementing a pretrained and well-known object detection model called Faster R-CNN (Ren et al., 2016). This foundational model was built upon the Region-Based Convolutional Neural Network (R-CNN) algorithm first introduced by Girshick et al. (2013). The original R-CNN architecture consisted of:

1. Accepting an input image
2. Extracting $\sim 2,000$ region proposals via the Selective Search algorithm (Uijlings et al., 2013)
3. Extracting features for each proposal ROI (region of interest) using a pretrained CNN
4. Classifying the features for each ROI using a class-specific linear layer



Faster R-CNN builds on top of the original R-CNN architecture (and the subsequent Fast R-CNN one) by creating an additional component called Region Proposal Network (RPN). The RPN generates a set of proposed bounding boxes to determine *where* an object could be in an image. These proposals are first generated using the anchors technique, where a set of bounding boxes at different scales from uniformly sampled points across the image. The goal of the RPN is to prune the number of proposals to a manageable size. For each proposal, the RPN module determines whether the anchor contains the foreground or background of the image and adjusts the anchor to better fit the object it is surrounding.

The anchors are classified as either foreground or background via the Intersection over Union (IoU) metric. The IoU is widely used to evaluate the performance of object detection models by comparing the ground truth bounding box to the predicted bounding box. The equation can be described as the area of the intersection of 2 bounding boxes divided by their area of union. The anchors are categorized using a default threshold of 0.5, where an $\text{IoU} \geq 0.5$ indicated foreground and an $\text{IoU} < 0.1$ indicates background.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


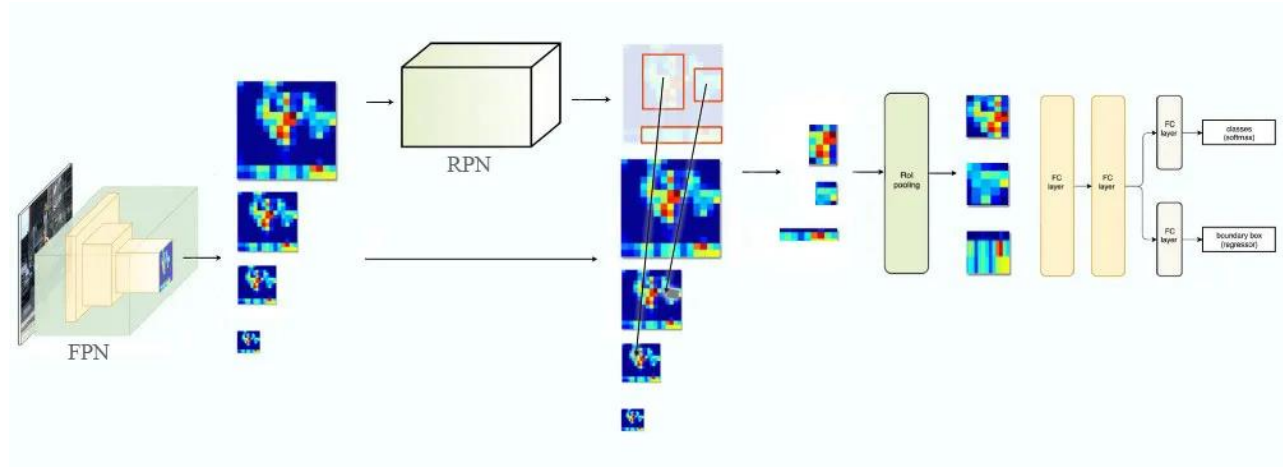
The proposals are then sent to the ROI pooling module, along with the features extracted by the backbone CNN. This module crops out feature vectors from the CNN feature map based on the proposed bounding boxes. The final feature vectors are fed into the R-CNN layer to obtain the bounding box coordinates and class labels.

Since there are two outputs, two loss functions are required. For bounding box regression, a smooth L1 loss is applied, and the categorical cross-entropy loss function is used for classification loss. These losses are summed for each batch images and targets during training.

For training, I tested both the ResNet-backed and MobileNetV3-backed pretrained models. The difference between these models is in their name and coincides with the architecture used as the backbone feature extractor for the Faster R-CNN network.

Both backbone classification CNNs are modifications of the original architectures because they also include a Feature Pyramid Network (FPN). The FPN is a feature extractor that generates

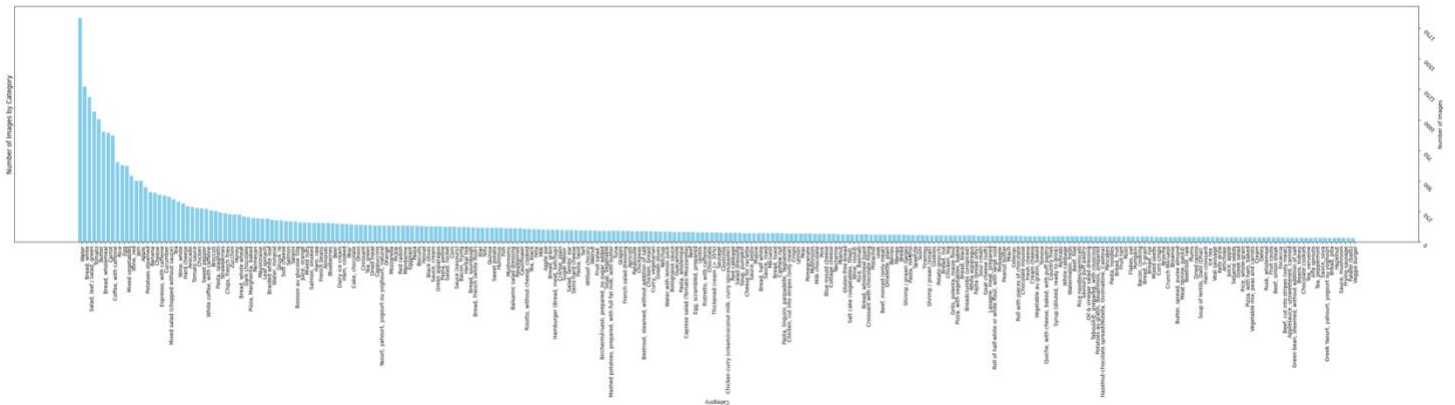
multiple feature map layers for an image using a bottom-up and top-down pathway. The bottom-up pathway is the normal convolutional network of the specific backbone and is used for feature extraction. The top-down pathway is composed of feature map layers that up sample the extracted features.



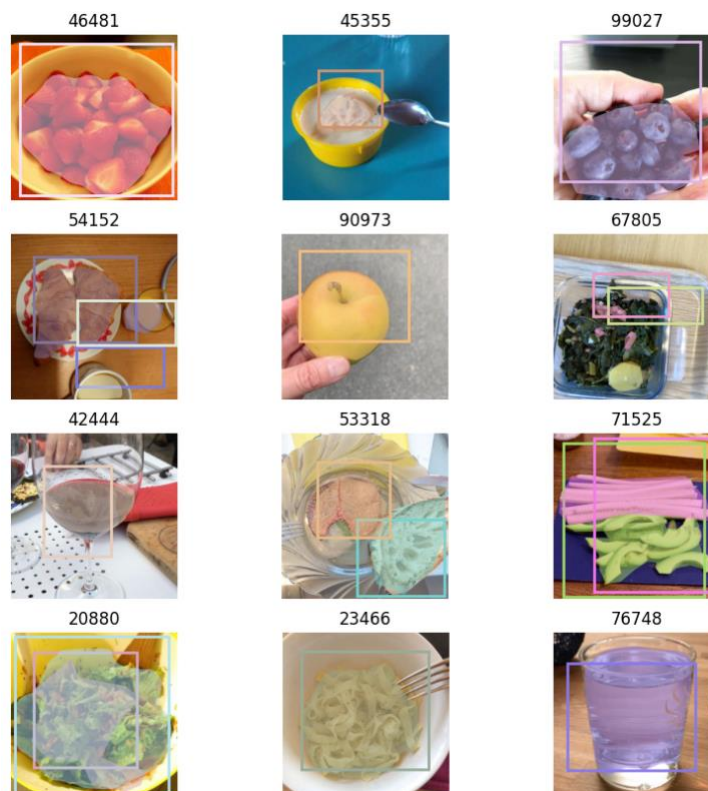
Exploratory Data Analysis

I performed some basic EDA on the dataset to visualize and gain an understanding of the distribution of our data.

The training set consisted of images containing labels from 323 different food item categories.



Based on the above distribution, I knew I was dealing with some imbalance. The water class was the most frequently occurring label in the data. Next, I visualized the ground truth labels and bounding boxes for a subset of images.



While developing this visualization, I learned that the ground truth bounding boxes in the provided annotations were in the [X, Y, Width, Height] format. I knew that I would need to convert these into the [Xmin, Ymin, Xmax, Ymax] format for model training.

Training

Training both models proved to be quite a challenge for me, and I was ultimately unsuccessful in obtaining meaningful results. However, I did learn a lot about constructing these types of models and working with the COCO data format.

The processes for training the custom and Faster R-CNN models were similar enough where I could structure their code in a concise manner. For the pretrained Faster R-CNN

Data Preprocessing

I used a few simple techniques and libraries to prepare the data for training. Once the image and annotations were read in, I applied some out-of-the-box transformations available from the torchvision transformations library. Since the annotations were provided in COCO format, I was able to use the Python COCO API to read in the data.

Image Transformations

For the images, I used the OpenCV Python library to read each file. Since OpenCV reads each image in BGR format, it had to be first converted to RGB before being input into the model. The training images consisted of a wide variety of widths and heights. To make it uniform, I applied a standard transformation to resize the images. Another technique I used right away is pixel normalization. For rescaling, I used mean and standard deviation values from ImageNet. Normalization helps to reduce the size of the logits and makes computations faster.

After running training for a few epochs and analyzing results, I decided to add a few more transformations, namely RandomHorizontalFlip and RandomRotation. These transformations help make the model more robust by expanding the bounds of its generalizing capabilities. After all transformations were applied, the image was converted to a tensor.

Annotation Transformations

There were a few key differences between the annotation format provided and the one required for the Faster R-CNN model:

- Annotations for individual images are provided as multiple entries in the annotations object. These must be grouped for each image. Luckily, the [COCO class](#) has that functionality built in.

- The bounding box arrays had to manually converted from [Xmin, Ymin, Width, Height] to [Xmin, Ymin, Xmax, Ymax]
- All possible category IDs had to be indexed in a list. A labels array was generated for each image containing the corresponding index value of each classified category ID from the list.

The new annotations object was returned along with the transformed image to be used by the training procedures.

Hyperparameters

In my code architecture, I extracted the hyperparameters configuration into a separate config file for more control over different training runs. The parameters I tuned included:

- **Detection score threshold:** This parameter was used to filter out detected annotations if their score fell below a specific value. The score was set at 0.5.
- **Batch size:** Used to control the batch size of the dataset during training and testing. Due to memory issues with the CUDA server, I landed on 16 as my final value.
- **Epochs:** I used 5 when testing out different configurations. After landing on the best, I used 20 for the MobileNet backbone model and 10 for the ResNet backbone model. The difference was due to required training time and memory constraints.
- **Optimizer:** Although not parameterized in the code, I tested the SGD and Adam optimizers available from PyTorch. I ultimately landed on using SGD with some additional specifications.
- **Learning rate (LR):** I used $1e-3$ as my starting value for this parameter, in conjunction with a LR scheduler to reduce the value by a factor of 0.1 every 5 epochs. I tested various values with and without the scheduler. The scheduler resulted in faster convergence since the initial LR was a higher value.
- **Momentum:** SGD with momentum is helpful for escaping local minima and faster convergence during training. I tested 0.9, 0.95, and 0.99, with the first value resulting in the lowest loss.
- **Weight Decay:** Along with momentum, I also chose to apply a weight decay to the optimizer and observed improved loss results. This adds a small penalty to the loss function. The penalty applied can be described by the following function: $\text{loss} = \text{loss} + \text{weight decay parameter} * \text{L2 norm of the weights}$.
PyTorch automatically applies the penalty to the weights and bias tensors. This is useful to prevent overfitting and results in smaller weights and bias values.

- **Gradient Clipping:** Used to force the step size of the gradient below a specific maximum so that the model is less likely to under- or over-fit. I started with a small value of 0.1 and worked up to final value of 1.0 in increments.

Results

As mentioned, I was unable to obtain quality results from my training and am still trying to figure out what is happening. Ideally, running inference on these models would have provided me with mean Average Precision statistic for different IoU thresholds so that I could compare each model's results. Without it, I was left to judge model performance by loss alone.

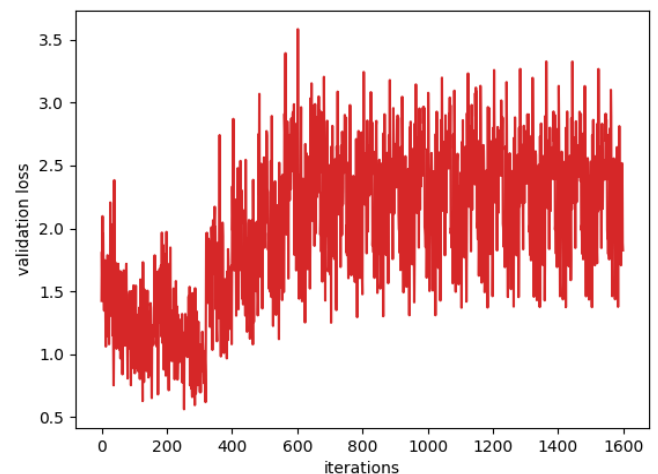
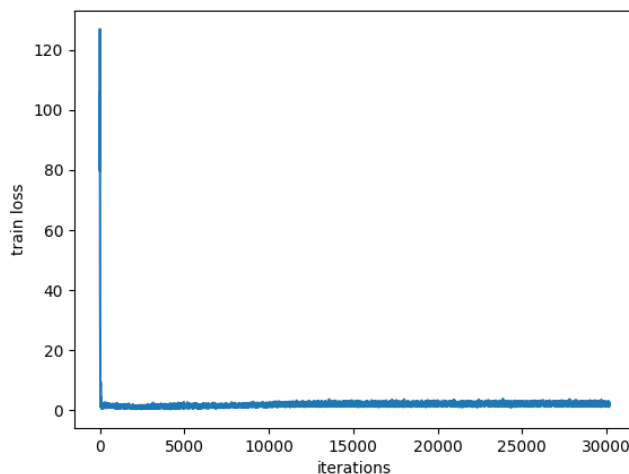
Custom

There are no evaluation metrics to show from training the custom model since I was unable to perform training on the model due to various issues.

Faster R-CNN

MobileNet-V3-FPN

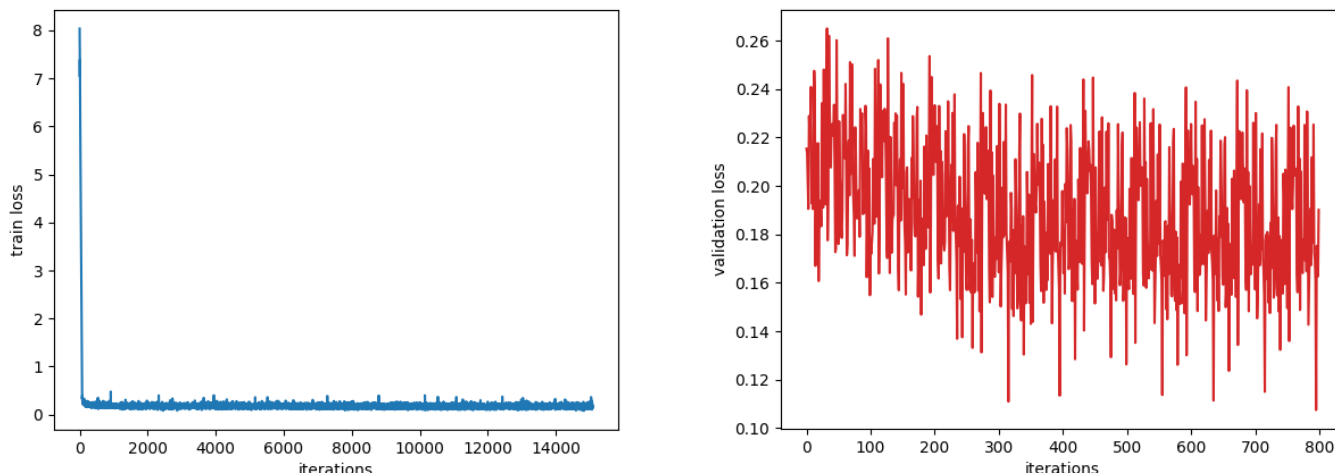
Best model loss:



The training loss quickly converged after a few iterations however, the validation loss – calculated using the provided test dataset – is erratic and displays an increasing pattern which is of concern. I attempted to tune this model by inspecting different stages of the pipeline to see if there were any discrepancies, but that effort was not fruitful.

ResNet50-FPN-V2

Best model loss



Once again, the training loss quickly converges after a few iterations by the validation loss remains erratic. This time, the validation loss is trending down which is an improvement at least. I tuned this model using the multiple combinations of hyperparameters but could not obtain metrics from inference.

Conclusion

Overall, this was an extremely difficult but rewarding experience. Instance segmentation and detection involves many different modules working in concert to product valuable outputs. As the composer of this model, we must be aware many different aspects of the data and training pipeline to ensure accurate, and in my case, useful results. I know there are problems in my data and model pipelines resulting in degraded performance. My goal is to continue working on these models and implementing an object detection model from scratch.

Some future considerations based on lessons learned:

- Data imbalance was present in the provided training dataset. I can use techniques such as weighted sampling to minimize the effects.
- Incorporating an FPN and anchor generator will yield better results.
- Test different backbones and use the FasterRCNN model constructor from PyTorch to customize the model for my specific needs.
- Get model inference to run correctly to assess the model's success more accurately.

Code Contribution

(485 lines from internet – 61 modified) / (485 + 670 own) = **36.7%**

Streamlit App

For my group's Streamlit application, I contributed a function that runs my Faster R-CNN model against an uploaded image and outputs the predicted labels along with an image annotated with the predicted bounding boxes.

References

1. Ronneberger, O., Fischer, P., & Brox, T. (2015, May 18). U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv.org. <https://arxiv.org/abs/1505.04597>
2. Ren, S., He, K., Girshick, R., & Sun, J. (2016, January 6). *Faster R-CNN: Towards real-time object detection with region proposal networks*. arXiv.org. <https://arxiv.org/abs/1506.01497>
3. ChatGPT by OpenAI
4. Martinez, H. (2023, November 13). Faster R-CNNs. PyImageSearch. <https://pyimagesearch.com/2023/11/13/faster-r-cnns/>
5. Weng, L. (2017, December 31). *Object detection for dummies part 3: R-CNN family*. Lil'Log (Alt + H). <https://lilianweng.github.io/posts/2017-12-31-object-recognition-part-3/>
6. Amidi, S. (2018). *Convolutional Neural Networks cheatsheet star*. CS 230 - Convolutional Neural Networks Cheatsheet. <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>
7. PyTorch. (n.d.). *Torchvision object detection finetuning tutorial*. TorchVision Object Detection Finetuning Tutorial. https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

Images

- Faster R-CNN Architecture: <https://lilianweng.github.io/posts/2017-12-31-object-recognition-part-3>
- FPN Network: <https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c>
- ROI pooling, IoU formula: <https://pyimagesearch.com/2023/11/13/faster-r-cnns/>