GlizzyNet

Hot Dog, Not a Hot Dog Computer Vision Segmentation

Lucas Rea – aUToronto Application

Introduction and Approach

This project is part of my application to aUToronto, UofT's self driving car team. As outlined in the assignment doc, the goal is to develop and train a machine learning model to identify and segment hot dogs in images.

Image segmentation is a difficult task, with many models being developed over the years; RCNN, Mask R CNN, Fast RCNN, Faster RCNN…. The list goes on. These RCNNs utilize CNN backbones (ResNet, VGG16, ...) to extract high level image feature maps. These feature maps are responsible for identifying various edges, shapes, colours, etc.

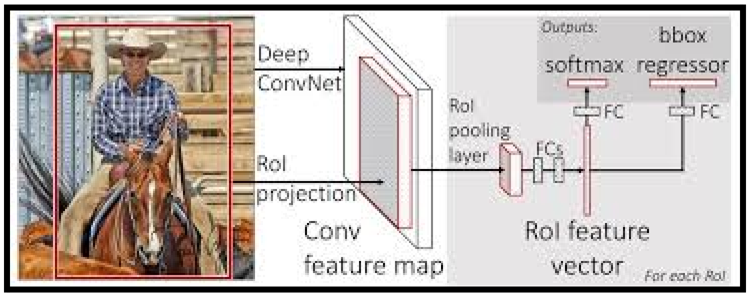


Figure 1: Faster RCNN Architecture

For this project I will utilize a pre-trained model, Faster RCNN with ResNet 50 backbone. I will then fine-tune the model using the provided dataset to identify and segment images of hot dogs. Faster RCNN uses feature maps to predict where an object may be, and identify it accordingly. The outputs of the model being the predicted label, and the bounding box of where the object lies.

My training process will focus on the Faster RCNN Predictor, as this is the part we want to fine tune. I will be training my models on Google Colab, utilizing an A100 to accelerate the training process. I compare the performance of 3 different training runs, with varying hyperparameters in my analysis.

In this informal report, I will include some formatted code snippets for class designs, and utility functions, though please see my notebook for the usable, working code.

Data Preparation

The data of this project was conveniently given in very organized formats. COCO and YOLO were the formats given, so the first thing I did was make myself familiar with both. COCO utilizes a large JSON file for ensuring each sample image has the appropriate properties stored. Whereas YOLO just has a txt file with the same name as its image counterpart. I found it easier to parse the COCO dataset, using PyTorch's dataset object. I found the data to be exactly the same so in my analysis I only used COCO. Though you could just as easily implement a transform function. I didn't find this part of the MVP.

```
datasets.CocoDetection(root=root, annFile=annFile)
```

In the first steps of processing the data, I wanted to take advantage of PyTorch's built-in functionality. The dataset and data loader. I created a custom dataset class which takes care of much of the normalized formatting into PyTorch, so the data is consistent with torch tensors.

```
class HotDogDataset(Dataset):
   def __init__(self, root, annFile, transform=None):
       super(HotDogDataset, self).__init__()
       self.root = root
       self.annFile = annFile
       self.transform = transform
       self.coco = datasets.CocoDetection(root=root, annFile=annFile)

   def __len__(self):
       return len(self.coco)

   def __getitem__(self, idx):
       img, target = self.coco[idx]

       # Apply transforms to the image
       if self.transform:
           img = self.transform(img)

       # Process target to handle bounding boxes
       boxes = []
       labels = []
       for obj in target:
           if 'bbox' in obj and len(obj['bbox']) == 4:
               x_min, y_min, width, height = obj['bbox']
               if width > 0 and height > 0:
                   x_max = x_min + width
                   y_max = y_min + height
                   boxes.append([x_min, y_min, x_max, y_max])
                   labels.append(obj['category_id'])

      # Convert lists to tensors
      # If no boxes were found, create an empty tensor with appropriate shape
       if len(boxes) == 0:
           boxes = torch.empty((0, 4), dtype=torch.float32)
           labels = torch.empty((0,), dtype=torch.int64)
       else:
           boxes = torch.tensor(boxes, dtype=torch.float32)
           labels = torch.tensor(labels, dtype=torch.int64)

       processed_target = {
           'boxes': boxes,
           'labels': labels,
       }

       return img, processed_target
```

You may notice the manipulation of the x_min, y_min, width, height values. Upon research into the pre-trained model, I found that it requires the bounding box to be in the format of; x_min, y_min, x_max, y_max. To reiterate, this is why I built this class, so any data processing techniques can be performed.

An issue I ran into during training, I found that some labels were empty tensors. I believe this meant that some samples did not have any bounding boxes. To handle that I made sure that data without bounding boxes, received a zero tensor or the same length the model is expecting. Of course, I didn't have time to look at every single image in the training and validation set, but this bug was solved after I implemented this technique.

Fully utilizing the DataLoader class from PyTorch, my data becomes fully ingested.

```
# Data transformations with resizing
transform = transforms.Compose([
   transforms.Resize((256, 256)),  # Resize images to 256x256
   transforms.ToTensor(),          # Convert images to tensor format
])

# Custom collate function to handle variable-length targets (some images may have 0-N
bounding boxes)
def custom_collate_fn(batch):
   images, targets = zip(*batch)

   # Stack images to form a batch
   images = torch.stack(images, dim=0)

   # Since targets are already in a list, just return them as is
   return images, list(targets)

# Create instances of the dataset
train_data = HotDogDataset(root=COCO_TRAIN,
                             annFile=os.path.join(COCO_TRAIN, "_annotations.coco.json"),
                             transform=transform)

val_data = HotDogDataset(root=COCO_VALID,
                             annFile=os.path.join(COCO_VALID, "_annotations.coco.json"),
                             transform=transform)

# Create DataLoaders
train_loader = DataLoader(train_data,
                         batch_size=BATCH_SIZE,
                         shuffle=True,
                         num_workers=2,
                         collate_fn=custom_collate_fn)
```

```
val_loader = DataLoader(val_data,
                       batch_size=BATCH_SIZE,
                       shuffle=False,
                       num_workers=2,
                       collate_fn=custom_collate_fn)
```

Using a function I wrote, visualize_images, we can visualize a batch from these Data Loaders with the bounding boxes present.

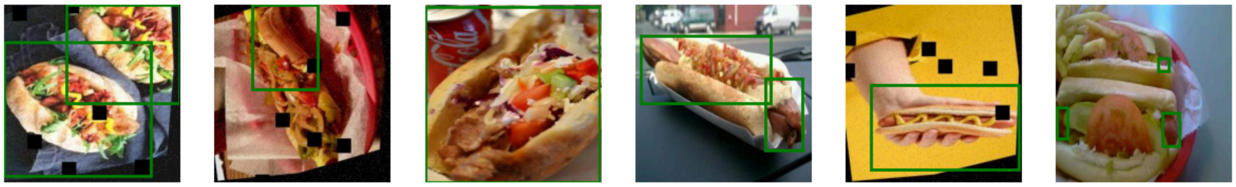


Figure 2 : Sample Hot Dogs with truth bounding box

After observing the data like this, I've seen that sometimes the data is labeled really well, and other times pretty poorly. Image 6 (from the left) thinks the hot dog is only the two ends sticking out. Image 4 also shows only 1 hot dog, yet 2 different bounding boxes. Nonetheless, this is our training data and our model should emulate this as well (a bonus if it performs better).

Model Architecture

As I mentioned in the introduction, I will be using a pre-trained model, RCNN with ResNet 50 backbone. Vision segmentation is difficult and I wanted to practice fine-tuning a complex and large model. I will only be training a portion of the base model. Since I don't want to modify any behavior in creating feature maps, or new ROIs, I will freeze these aspects of the model, as to only train the predictive power of bounding box regression, and category classification. However, I have passed these "freeze region" parameters into the model, so this is easy to adjust.

```
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection import FasterRCNN_ResNet50_FPN_Weights

class GlizzyNet:
   def __init__(self, num_classes=2, freeze_backbone=True, freeze_rpn=True,
freeze_fpn=True):
       self.num_classes = num_classes
       self.device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
       self.model = self._initialize_model(freeze_backbone, freeze_rpn, freeze_fpn)
```

```
    def _initialize_model(self, freeze_backbone, freeze_rpn, freeze_fpn):
        # Load pre-trained Faster R-CNN model with ResNet50 FPN backbone
        model = fasterrcnn_resnet50_fpn(weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT)

        # Modify the number of output classes (including background)
        in_features = model.roi_heads.box_predictor.cls_score.in_features
        model.roi_heads.box_predictor = FastRCNNPredictor(in_features, self.num_classes)

        # Freeze parts of the model as specified
        if freeze_backbone:
            for param in model.backbone.parameters():
                param.requires_grad = False

        if freeze_rpn:
            for param in model.rpn.parameters():
                param.requires_grad = False

        if freeze_fpn:
            for param in model.backbone.fpn.parameters():
                param.requires_grad = False

        # Freeze parts of the ROI heads, except the final classifier layers
        for name, param in model.roi_heads.named_parameters():
            if "box_predictor" not in name:
                param.requires_grad = False

        return model.to(self.device)
j in
    def get_model(self):
        return self.model
```

Training and Performance

In the early stages of training, I quickly realized my M1 MacBook pro simply doesn't have the horsepower to fine-tune a model like this. Unfortunately, PyTorch's MPS (Metal Performance Shaders) support wasn't fully configured for the pre-trained model I used. Thus, I had to change my training location to Google Colab.

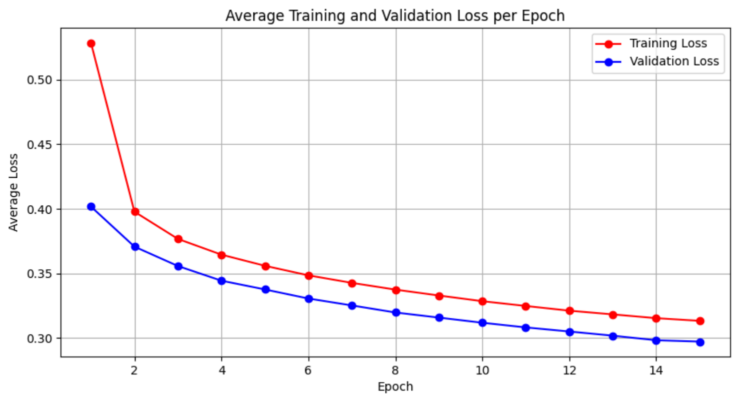
During training,I quickly noticed that the loss diminished very quickly, thus after some trial runs on Google Colab, I kept my total number of epochs at 15. This provided enough training time for the model to converge, while also not taking 30 minutes to train it once.

I performed 3 tests with slightly different hyperparameters.

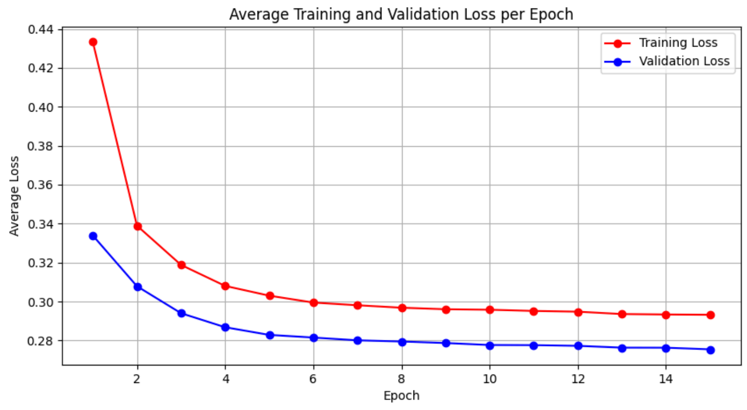
- Batch Size = 16, Learning Rate = 0.00005
- Batch Size = 32, Learning Rate = 0.0005

- Batch Size = 64, Learning Rate = 0.001

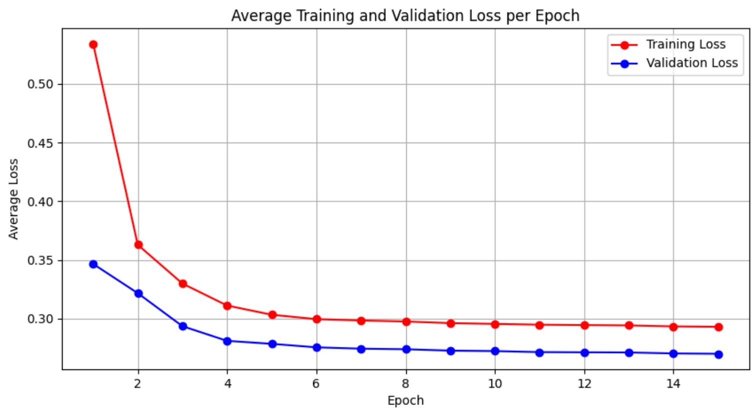
I chose these different tests to try and gauge if the loss being propagated back at different batch sizes had an effect on total model loss. I adjusted the learning rate somewhat accordingly. I developed the following train-validation curves.



Graph 1 : Batch Size = 16, Learning Rate = 0.00005



Graph 2 : Batch Size = 32, Learning Rate = 0.0005



Graph 3 : Batch Size = 64, Learning Rate = 0.001

Looking at the following curves, we can point out some obvious things.

1) The average validation per epoch is better than the training loss across graph 1-3. This was somewhat surprising, though we have to consider a few things.

   The model is already pre-trained, meaning it can generalize very well to the validation data already. There may also be some regularization effects (like dropout, weight decay) applied during training can increase the training loss but help the model generalize better, resulting in a lower validation loss. This is something I would have liked more time to investigate and research.

   Finally, we have to consider the fact that the validation dataset may be “easier” to predict. Perhaps the data in the validation set is cleaner and its bounding boxes are less cluttered.

2) It appears that the model with a batch size of 16 could have reduced its loss, if it kept training. By increasing the batch size, the model converges to a very small loss, very quickly. This makes sense, since the model is able to adjust the loss after seeing more data – it's able to learn faster.

   It’s also nice to observe that the training loss and validation loss decay in the same way. This gave me confidence that the model was learning and being fine-tuned to my data.

Model Evaluation

Of course there are other things we have to consider when evaluating the performance of the model. Loss is useful during training to ensure we’re on the right track, but the raw predictive power of the model comes from a few different figures of merit.

For the following model evaluation metrics, I will be using the model I saved (via checkpointing) from the 64 batch size training process. I’m using this one because the validation and training loss was the lowest. It is my best performing model. I call this, GlizzyNet! I wasn’t able to include the model in my submission (100MB limit) though with a batch size of 64, and learning rate of 0.001, you can emulate and train my model.

Latency and Inference

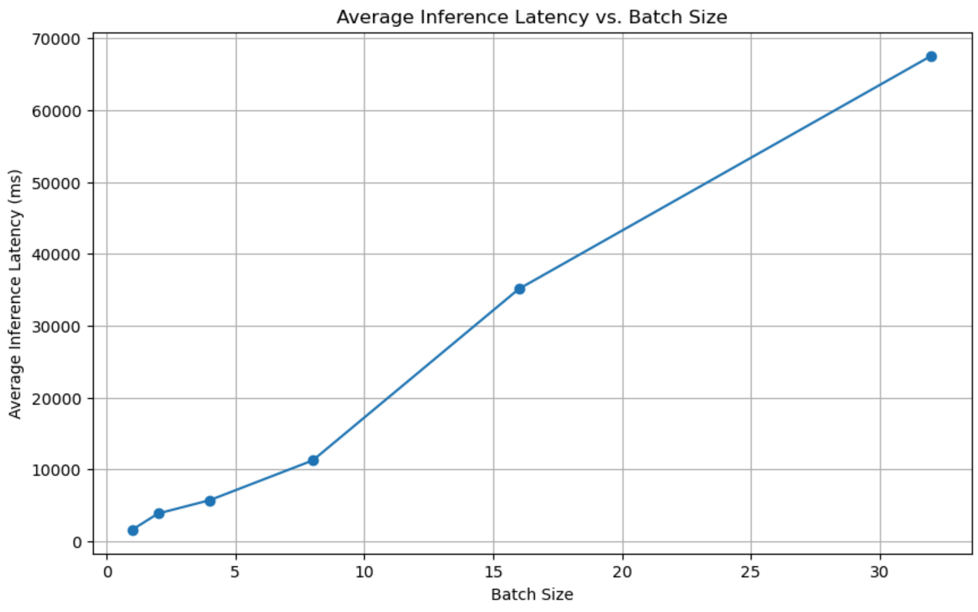
Working at AMD has taught me how important inference latency is, typically measured in milliseconds (ms). It’s a key figure of merit used by every company and LLM benchmark. It is also extremely important in the context of computer vision. The speed at which your model is able to infer predictions from an input video stream, while driving, means making life saving decisions, a lot faster.

I decided to perform inference testing across multiple batch sizes, to see if there is any dropoff with the model. I also used CUDA on Google Colab to modify the numerical precision at inference time.

The first test, batch size vs average latency time, was performed on my M1 Macbook pro (8GB Memory) – thus no CUDA acceleration for this first test. I decided to test the following batch sizes.

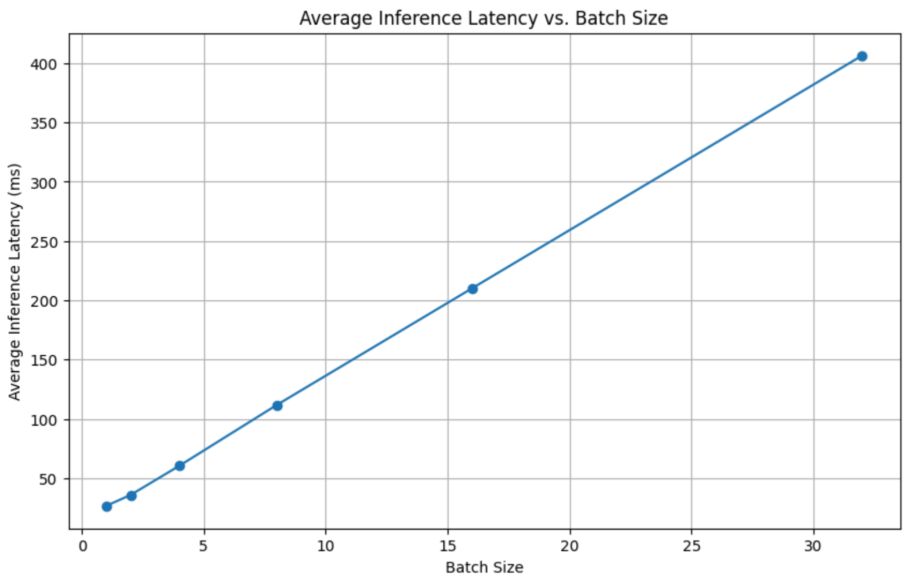
```
BATCH_SIZES = [1, 2, 4, 8, 16, 32]
```

For every batch size, I utilized a few warm-up iterations, followed by the test. When performing the test, I ran each batch (with the given batch size) 10 times, and computed the average latency. The results are as follows.



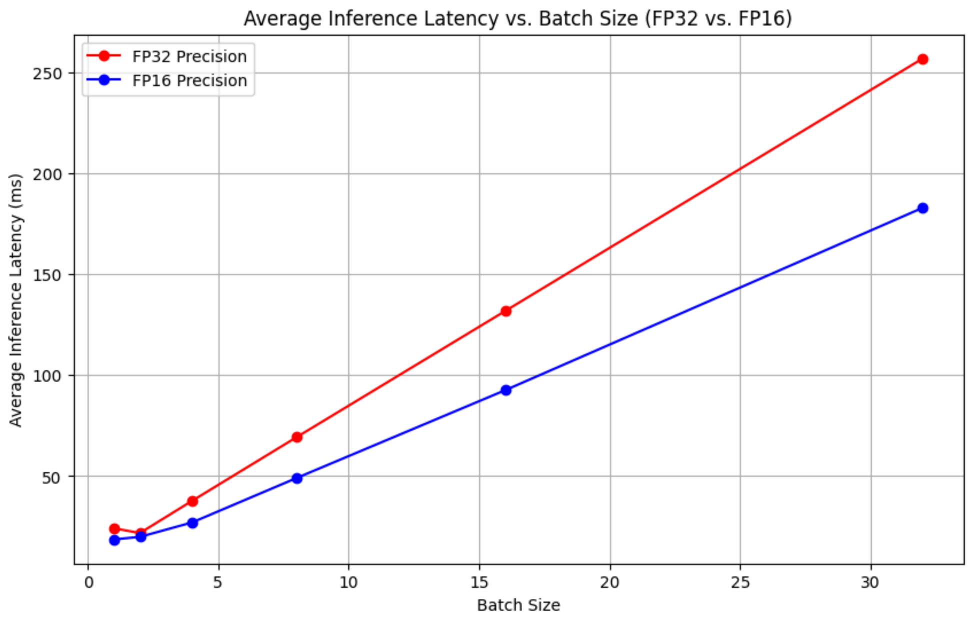
Graph 4 : Average Latency per batch (M1 Macbook pro, 8GB RAM)

This next test is exactly as outlined above, except I ran this on Google Colab (A100 with 40 GB of GPU RAM). We can see it takes significantly less time to perform inference on the A100, though it scales in the same way as on my Macbook pro.



Graph 5 : Average Latency per batch (A100, 40GB GPU Memory)

This final inference test uses CUDA’s ability to cast inputs to half the floating point precision (default is fp32). We can see significant speed ups while only using half the precision at inference time.



Graph 6 : Average Latency per batch, floating point quantization, FP32 vs PF16 (A100, 40GB GPU Memory)

IoU and mAP

Intersection over Union (IoU) measures how much the predicted bounding box overlaps with the truth bounding box.

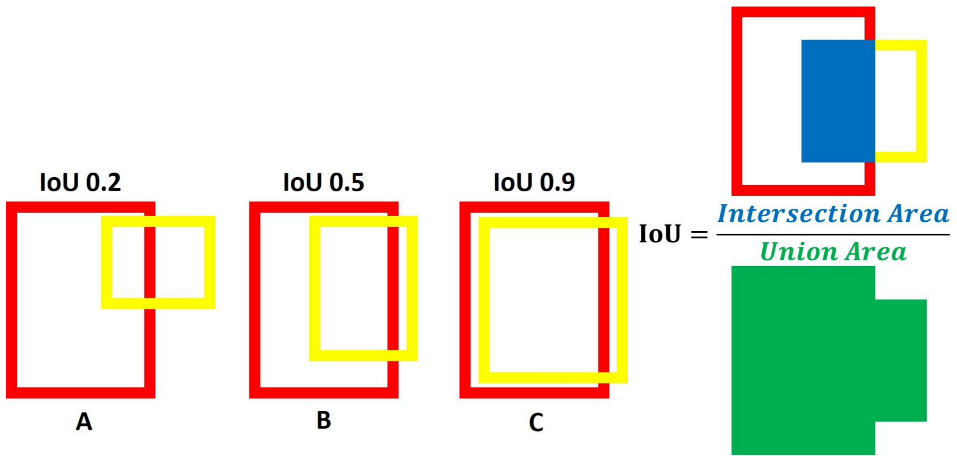
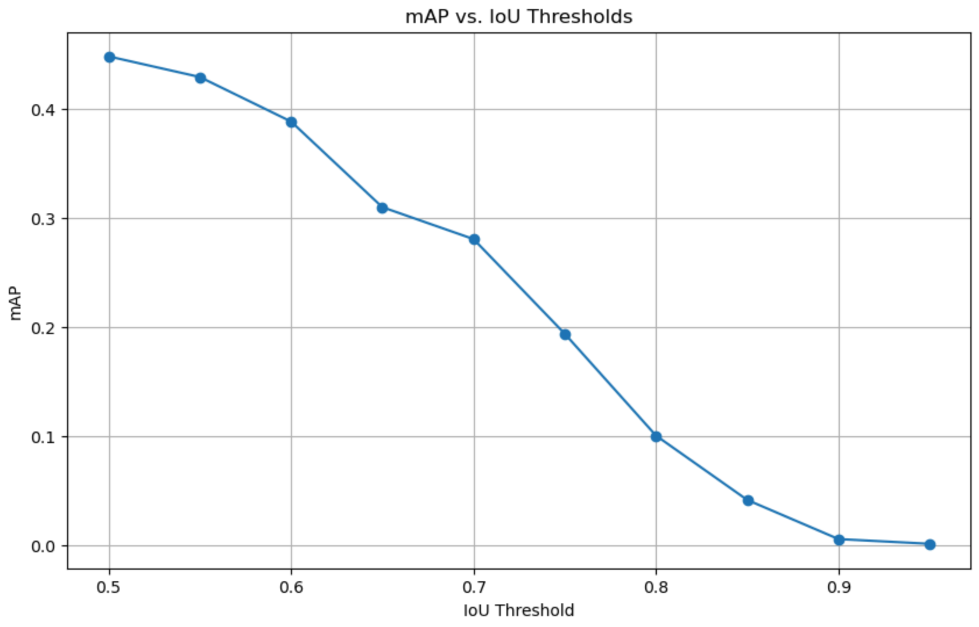


Figure 3 : IoU of prediction vs truth bounding boxes

The mAP, or mean average precision measures the average precision across all classes (2 classes in our case, hot dog, and background). As we raise the IoU, we demand that more of our model's predictions should lie within the truth's bounding box, and as a result we should see a drop in precision.



Graph 7 : Average Precision across varying IoU threshold

Looking at graph 7, we can see a steady decline from IoU threshold 0.5 to about 0.7. After this we start to see more significant drop offs. It may be interesting to see the bounding boxes it predicts with IoU between these values.

Qualitative and Robustness

All of this data above quantitatively shows how our model performed. But in practice we would use this to identify hot dogs, so let's see how it did. I will draw the bounding truth bounding boxes, and GlizzyNet's bounding boxes in the same image. The truth boxes in green, and GlizzyNet's in red. These images are with an IoU threshold of 0.5.

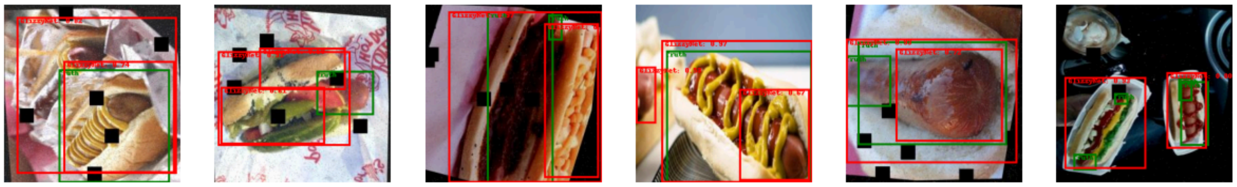


Figure 4 : Validation Hot Dogs, Truth boxes (green), GlizzyNet boxes (red)

We can clearly see that in figure 2, GlizzyNet sometimes does a better job at drawing the bounding boxes than our truth data displays. Image 1, GlizzyNet captures both. However image 2 it draws 3 boxes for only 1 hot dog – though the truth only captures the end of the hot dog. This may explain why our average precision was always calculated less than 0.5.

If we increase the IoU threshold of GlizzyNet to 0.7, we can observe the following.

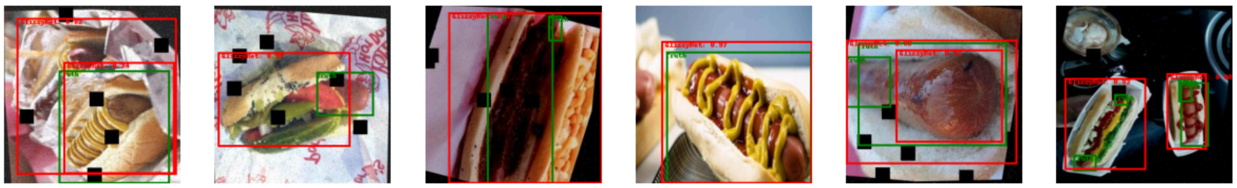


Figure 5 : Validation Hot Dogs, Truth boxes (green), GlizzyNet boxes (red)

We can see in figure 5 that by increasing the IoU threshold, GlizzyNet's bounding boxes are more consolidated and do a better job at segmentation. This does lead to mismatches with the truth boxes – which once again explains why the mAP was lower than expected. Though GlizzyNet does a fantastic job and appears to be very robust.

Future Considerations

The goal of the project was to correctly segment images of hot dogs, and in that sense this project has been a success. I wanted to create a MVP, so as to submit my application to aUToronto in a timely manner. However, there are quite a few things I've thought about, that given much more time, I would be able to take a closer look at.

Training other aspects of the model

As I mentioned in the Model Architecture section, I only trained the Faster RCNN Predictor. This was to have a faster training process, and not re-learn various feature maps and ROIs. It would be interesting to see if training other, deeper parts of the model result in better performance and predictive power. This is why I allowed the freeze parameters into the constructor of my model.

A potential Micro RCNN

I found this project rewarding, in that fine-tuning a pre-trained model comes with some of its own challenges. Though another approach I could've taken was to design the RCNN from scratch. This would have involved a lot of extra work and time – the payoff would be the experience in designing my own RCNN. However due to my own time constraints I quickly realized this was not feasible.

Data Augmentation

A common technique used when developing vision based systems is to augment training data. This may be rotations and other transforms to the data which can make the model more robust. After observing some of the training data, I saw that some images of the hot dogs were in many different positions (birds eye, side view, top down), as well as some pixels blacked out to act as noise. I realized that this is probably good enough, and to achieve a MVP, this was certainly not a requirement.

Loss Function

The pre-trained model I used already returned the loss scores during training. These are training scores from the category classification, as well as the bounding box regression score (utilizing their respective loss functions). Since we are only evaluating our model on hot dogs, a future test would be for giving less weight to the classification loss value.

Conclusion

I found this challenge very fun and I certainly learned a lot. It gave me a good idea of what work would be like with aUToronto (though with less hot dogs I hope). I believe I have achieved a MVP for this challenge in utilizing the training and validation datasets. GlizzyNet appears to have achieved better qualitative performance than the data it was used to fine-tune with. This is almost certainly due to the fact that I utilized transfer learning for this challenge.

References

Faster R-CNN: Features and Classification in One Stage - Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017. https://ieeexplore.ieee.org/document/7485869

Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation - Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik. CVPR 2014.
https://ieeexplore.ieee.org/document/6909475