FLAIROP Final Report

Vision and Image Processing Lab, University of Waterloo

Fall 2022

Kai Ma

Contents

1 Problem Overview

1.1 General Problem

The main problem I worked to address during this co-op term was **data imbalance for object detection** on the MetaGraspNet dataset. Specifically, we found that after training one network with all classes on the dataset and generating a confusion matrix, certain classes suffered from much lower accuracy. A particularly prevalent issue causing this was high similarity between classes. For example, the "*d-toy-airplane*" and "*j-toy-airplane*" classes represented two types of screws with identical heads but different tip, with d-toy-airplane having a flat tip and j-toy-airplane having a circular Phillips-style head. As we can see from the confusion matrix in Figure 1, our original model for all classes does badly on this task, frequently confusing the two classes. We chose this case as a representative example to do experiments on as it was clear, simple, and representative of the overall data imbalance problem.

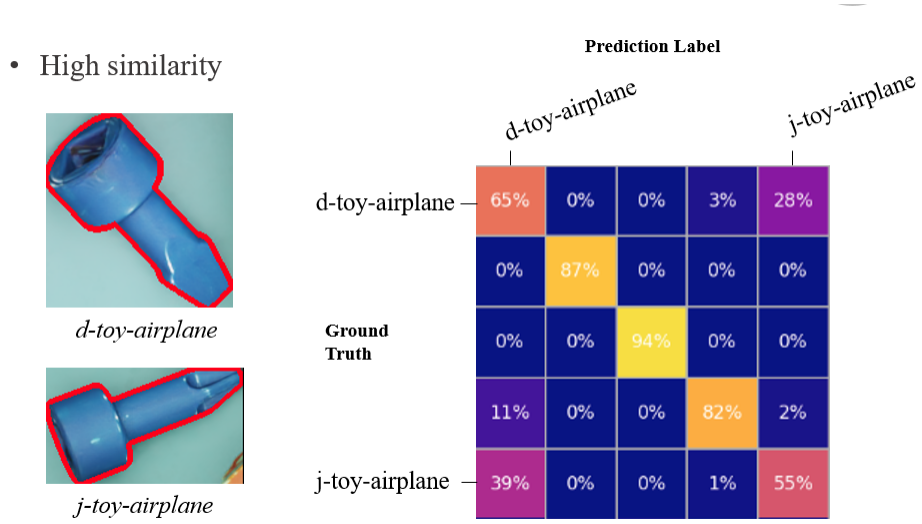

Figure 1: Toy airplane screw objects and confusion matrix

1.2 General Solution

We found that splitting the classes that are more difficult to train and training individual networks on these specific datasets led to an improvement in performance. For example, for our representative case of the toy airplane screws, we created a sub-dataset of MetaGraspNet, where every image contained at least one of the toy airplane objects. Then, we only ask the model to identify these two classes instead of all 97 in MetaGraspNet. Examining the confusion matrix for this expert model showcases an impressive improvement in performance (Figure 2). Other problematic classes showed similar increases in performance, showing that this approach generalizes well.

Note: The percentages in these confusion matrices often do not add up to 100; this is because the "background" class is also considered to account for objects that are not detected at all.

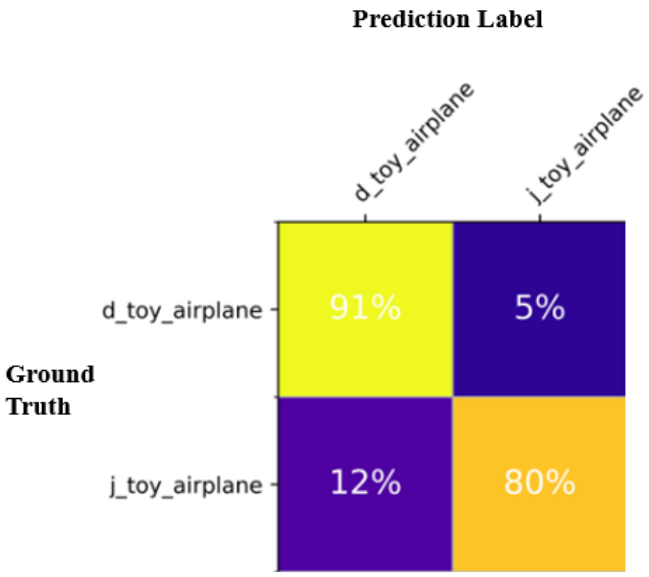

Figure 2: Confusion matrix for basic expert model

Since this expert model approach proved to be successful, our general solution was to use adaptive **mixture-of-experts** (MoE) model ensembles, which allow us to combine the predictions of trained models. We also considered other ways of combining expert predictions, such as bounding box ensembling, but found MoE to be the most versatile.

MoE is an ensemble learning technique classically used for classification tasks. A diagram illustrating the MoE architecture is shown below in Figure 3. It follows this general approach:

- Decompose task into sub-tasks, which are delegated to experts
- Use "gating network" that learns which expert to trust for a given task
- Either directly which expert to use or combine expert predictions

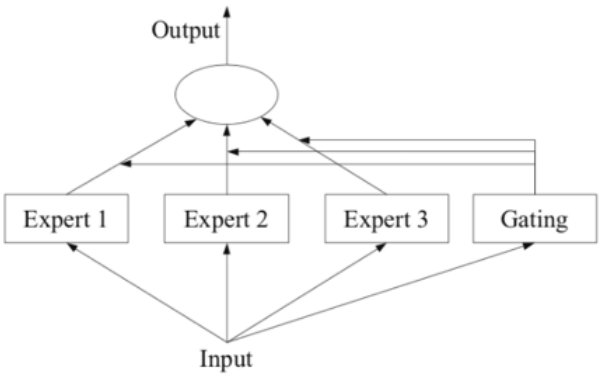

Figure 3: MoE Architecture

2 Experiments

2.1 Superclass Experiment

While we have shown that the expert is able to differentiate between the two expert classes, it is important that it can first identify objects that belong to these two classes. Thus, we combine the two classes into a "superclass". Thus, we train a model with 95 regular classes and a superclass ("d_j_superclass" = d_toy_airplane + j_toy_airplane"), to validate this. This experiment returned an excellent result — 99% accuracy.

Figure 4: Superclass Experiment Confusion Matrix

2.2 Architecture Experiments

While the expert model shows great improvement from the original, it is still far from perfect. Thus, an important consideration is identifying the bottleneck for classification; specifically, are the feature layers the bottleneck or simply the last classification layer in the bounding box head?

To determine this, the first experiment we conduct is **freezing the feature backbone** and training the classifier head to see if there is improvement. We observe no significant change, indicating that training the classifier further has no effect and that the feature backbone is the crucial component.

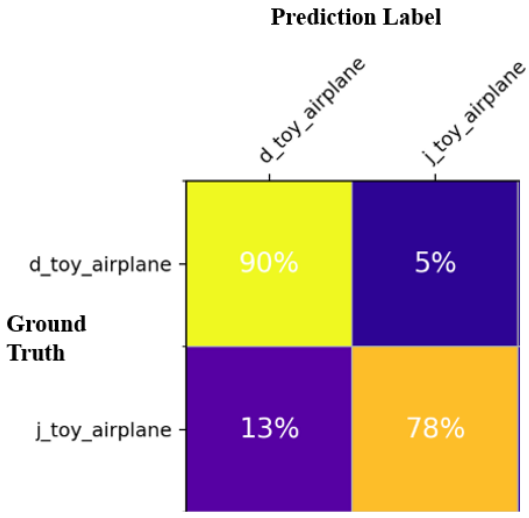

Figure 5: Freezing Backbone: No significant change

To further validate this result, we also try training with a **different bounding box head** to see if there is improvement. Instead of Mask-RCNN's standard bounding box head, we try a different architecture ("Shared4FC1Conv"). This also has a similar result to the original expert model, which implies that our previous conclusion is likely correct — the crucial component for classification is the feature backbone.

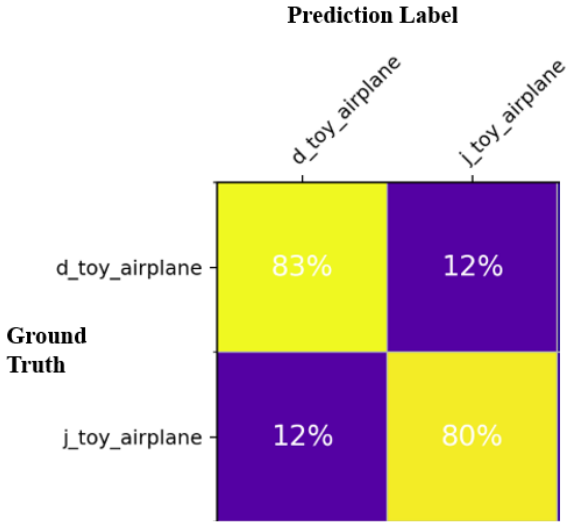

Figure 6: Alternate BBox Head + Frozen Backbone: No significant change

Since we have determined that the feature backbone is the crucial component to classification, can we use a larger backbone for better feature learning? To determine this, we replace the standard ResNet-50 Mask-RCNN with a ResNet-101 backbone. Since there is **no improvement**, we see that the standard feature extractor backbone is performing adequately and the lackluster performance is likely due to the dataset itself.

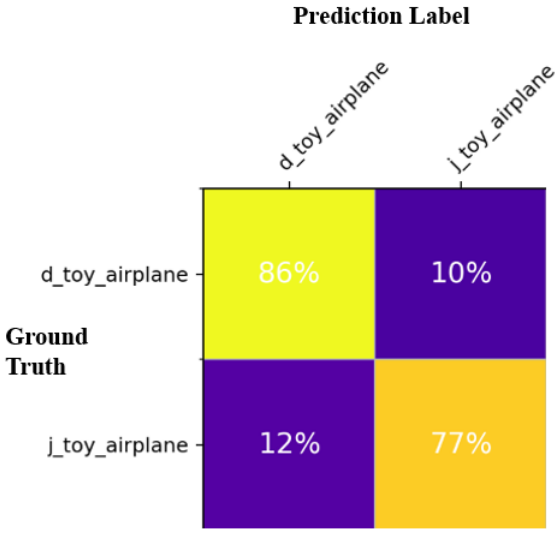

Figure 7: Larger backbone: No improvement

If the standard ResNet-50 backbone is adequate, is it possible to use a smaller backbone? We find that using a ResNet-18 backbone still has similar performance to the larger models.

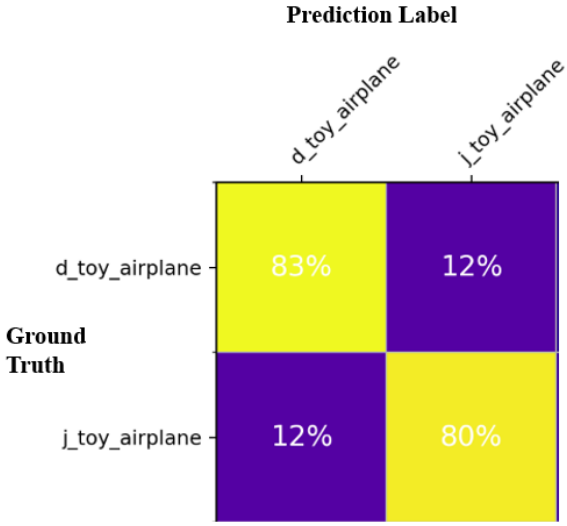

Figure 8: Smaller backbone: Similar performance to larger backbones

Table 1: Summary of Architecture Experiments

Backbone Architecture	Frozen Backbone	BBox Head	Accuracy
ResNet-50	No	Shared 2FC BBox Head	85.5%
ResNet-50	Yes	Shared 2FC BBox Head	84%
ResNet-50	Yes	Shared 4FC 1Conv BBox Head	83%
ResNet-101	No	Shared 2FC BBox Head	81.5%
ResNet-18	No	Shared 2FC BBox Head	81.5%

2.3 Difficult Cases

Since performance is similar in many of our models, it may be possible that the data samples are inherently hard to classify, meaning that perfect performance is not possible. Upon some inspection, we find that some images in the dataset are not classifiable due to obstruction. Two examples of such are shown in Figure 9

Figure 9: Examples of difficult cases

2.4 Identifying Causes for Improvement

Because the dataset has difficult cases, why do we observe improvement from the expert model? Is it simply due to random guessing for difficult cases? To determine this, we identify 3 types of cases and examinne the performance of the original and expert models on them.

2.4.1 Case 1: Easy and correct

We see that even for easy cases with no occlusion, the expert model is more confident (1.00 vs. 0.79). This means that while both models are capable of correct classification, the expert model is better at differentiating between the classes and not just randomly guessing.

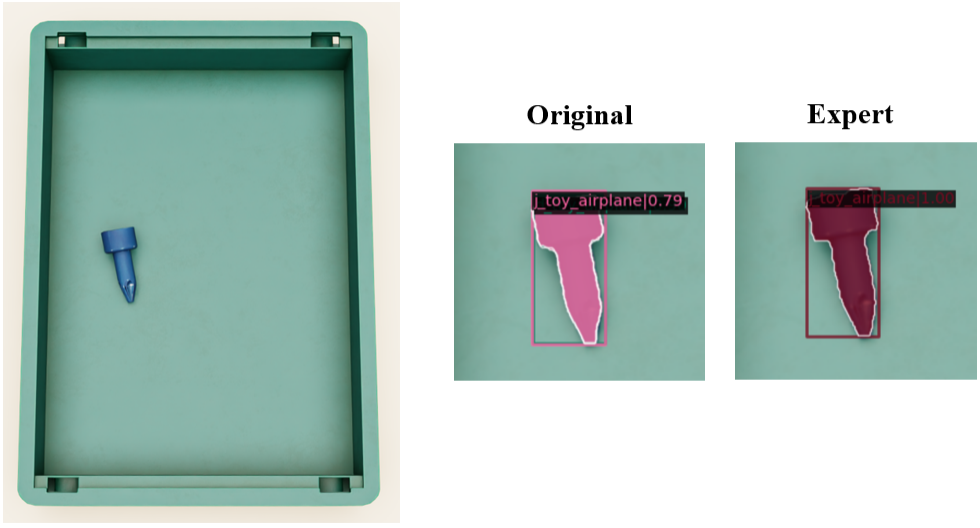

Figure 10: Easy case with expert/original model performances

2.4.2 Case 2: Misclassifications due to low confidence

The lack of confidence results in frequent misclassifications for the original model, while the expert model can identify these cases correctly.

Figure 11: Misclassification due to low confidence

2.4.3 Case 3: Difficult due to obstruction

In difficult cases with a lot of occlusion that are essentially impossible to classify, both models get similar results in terms of correctness. This is most likely due to random guessing. However, since the expert model still tends to be more confident, the original model is sometimes better at random guessing due to its lower confidence.

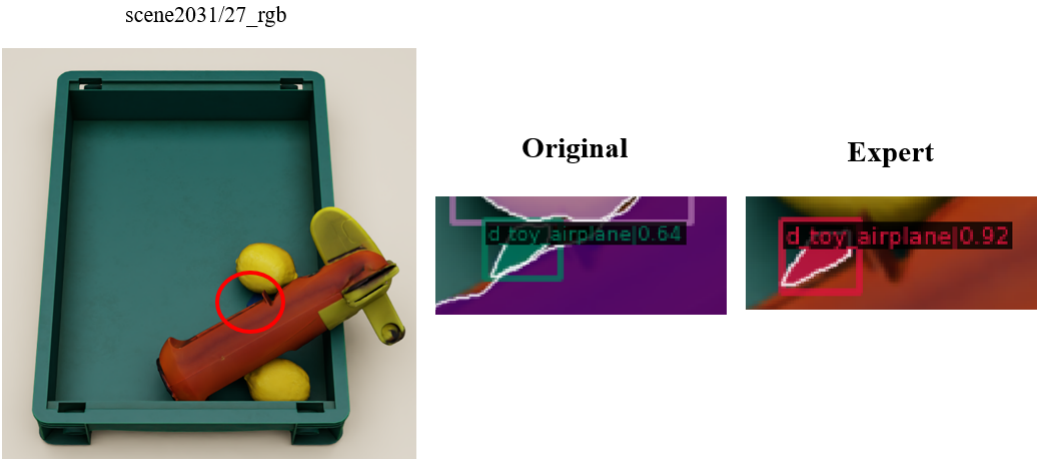

Figure 12: Easy case with expert/original model performances

2.4.4 Case Statistics

We sample a small set of 20 results output by the original model to get an idea how many of such cases are present in the dataset:

- Case 1 (Easy and correct): 9
- Case 2 (Misclassified due to low confidence): 5
- Case 3 (Obstructed, random guessing): 2 correct, 4 incorrect

We see that this distribution is approximately equivalent to the confusion matrix of predictions made by the original model in Figure 1.

2.5 Class distribution of training data

2.6 Prediction probability for all classes in images I showed

3 Implementation of MoE Proof-of-Concept

3.1 MoE Object Detector Implementation

I implemented a simple MoE proof-of-concept based on MMDetection's TwoStageDetector class. This allows us to use the utilities of the MMDetection library, which includes many useful tools. Some main changes are listed here.

I changed the initialization of the class to also take arguments for "experts" (a list of trained models) and "gating network" (described below).

```
# define the MixtureOfExpertsTwoStageDetector class
class MixtureOfExpertsTwoStageDetector(TwoStageDetectorMoE):
    def __init__(self, experts, gating_network, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.experts = experts
        self.gating_network = gating_network
```

Figure 13: MoE class initialization

Then, I changed the forward pass for the training function ("forward_train") to:

- Calculate gate values using the gating network
- Select an expert based on the output of the gating network
- Extract features from the expert using the selected expert

```
#Gate
# Calculate gate values using the gating network
gate_probabilities = self.gating_network(img)

# Find highest gate value (most confident value)
selected_expert = self.experts[torch.argmax(gate_probabilities)]

# Extract features using the expert with highest confidence
x = selected_expert.extract_feat(img)
```

Figure 14: Gate and feature extraction

I chose to design this in this manner according to our experiments in Section 2.2 (Architecture Experiments), where we determined that the backbone feature extraction is the most crucial stage. A similar gate-based feature extraction is also included in the forward pass of the testing function ("simple_test").

3.2 Gating Network Implementation

I implemented a simple gating network that takes an input image and outputs a set of weights that allow us to choose an expert to use based on their confidence. I experimented with several different architectures; the one shown below is one of the relatively simpler ones. Here is the way I defined the gating network:

- Defined the input and output layers of the gating network. The input layer takes the input image, and the output layer output the model confidences for each of the expert models.
- Defined hidden layers of the gating network, as well as activation functions (ReLU).

```python
class GatingNetwork(nn.Module):
  def __init__(self, input_size, num_experts):
    super(GatingNetwork, self).__init__()

    # Define the input and output layers
    self.input_layer = nn.Linear(input_size, input_size)
    self.output_layer = nn.Linear(input_size, num_experts)

    # Define the hidden layers
    self.hidden_layers = nn.Sequential(
        nn.Linear(input_size, input_size),
        nn.ReLU(),
        nn.Linear(input_size, input_size),
        nn.ReLU()
    )

  def forward(self, x):
    x = self.input_layer(x)
    x = self.hidden_layers(x)
    x = self.output_layer(x)

    # Normalize the output to sum to 1
    return nn.functional.softmax(x, dim=-1)
```

Figure 15: Simple gating network

The "train_step()" function of the detector, which defines an iteration step during training, is modified to include the gating network, allowing it to be trained simultaneously.

In order to allow the gating network to be trainable during the whole training process, I added a custom hook, which trains the gating network after every training iteration. This hook is optional.

```python
from mmcv.runner.hooks import HOOKS, Hook

@HOOKS.register_module()
class GateTrainingHook(Hook):
    """Check invalid loss hook.
    This hook will train the gating network after .
    Args:
        interval (int): Checking interval (every k iterations).
            Default: 50.
    """

    def __init__(self, data, optimizer, gating_network, gating_loss):
        self.data = data
        self.optimizer = optimizer
        self.gating_network = gating_network
        self.gating_loss = gating_loss

    def after_train_iter(self, runner):
        # Compute the output of the gating network and the expert networks
        gate_output = self.gating_network(self.data)
        expert_outputs = self.expert_networks(self.data)

        # Compute the loss for the gating network
        gating_loss = self.gating_loss(gate_output)

        # Compute the loss for the expert networks
        expert_losses = self.expert_losses(expert_outputs)

        # Weight the expert losses by the gate output and sum them to get the total loss
        total_loss = torch.sum(gate_output * expert_losses)

        # Add the gating loss to the total loss
        total_loss += gating_loss

        # Backpropagate the total loss and update the parameters
        self.optimizer.zero_grad()
        total_loss.backward()
        self.optimizer.step()

        # Return the total loss and any other log variables
        return dict(loss=total_loss, log_vars={'gating_loss': gating_loss})
```

Figure 16: Custom hook

4 Other Work

- During the first few weeks of this term, I worked on a **COCO Label Generator** for the MetaGraspNet dataset, which allows users to customize subsets of the dataset based on object classes and novelty inclusion.
- I volunteered to be a part of the **CVIS 2022** organizational committee, and was mainly in charge of banner design and printing. I also presented a poster at the conference as part of the COVID-Net project.
- Half of my working time this term (20 hours/week) were assigned to work on the **NRC-IRAP** project under Dr. Shimon Schwartz and Prof. Linlin Xu, where I worked on a graphical user interface (GUI) to allow users to train financial growth prediction models.