Semantic Segmentation & Knowledge Distillation

ELEC 475

Lab 4

December 3rd, 2024

Ryan Zietlow 20347719

Simon John 20348233

Network Architecture

The team decided on a compact U-Net model for this semantic segmentation task. This decision was made because U-Net is specifically designed for image segmentation where pixel level accuracy is essential for the model's performance.[1] Traditional U-Net models have a very large number of total trainable parameters which is not suitable for this task, so the team made, WannabeUNet, which retains its basic architectural principles such as the encoder-decoder and skip connections, but instead uses smaller convolutional kernels, less filters in the intermediate layers, and reduces the depth and width of layers to be more lightweight and efficient but still keep the ability to learn complex features.

WannabeUNet begins with three convolutional encoder layers. The first layer, conv1, is a 2D convolutional layer that has 3 input channels for RGB image, 32 output channels (feature maps), a kernel size of 3, and padding of 1 to maintain the correct spatial dimensions. A batch normalization layer is used to normalize the activations and improve the convergence of the model. A ReLU activation function is then applied to prevent the model from being linear. The second layer, conv2, takes 32 input channels from the previous block and outputs 64. It keeps the same spatial dimensions with a 3x3 kernel and padding. Batch normalization and ReLU are applied the same way as in the previous block. The third layer, conv3, takes 64 input channels from conv2 and outputs 128 feature maps, again keeping the same spatial dimensions and applying batch normalization and ReLU. After these layers, a max pooling layer is used to reduce the spatial dimensions by half by implementing a 2x2 kernel with a stride of 2 to perform down sampling. This is used after each convolutional block to help the model capture high level features and to reduce the spatial size.

The architecture continues to the decoder section that consists of upsampling layers. The first of these is a transposed convolution block that performs upsampling (deconvolution) to double the spatial dimensions, it converts 128 input channels to 64 output channels which increases the resolution of the image. It also normalizes the activations after upsampling with a batch normalization. The next layer is a skip connection and convolution, a focal point of the WannabeUNet architecture. The skip connection allows the feature map from the previous encoder layer to be added to the upsampled tensor to maintain spatial features. It then performs a

[1] https://viso.ai/deep-learning/u-net-a-comprehensive-guide-to-its-architecture-and-applications/

3x3 convolution to the result, with 64 input and output channels respectively. Then a batch normalization and ReLU are applied to normalize and provide non-linearity to the model. This layer is followed by another transposed convolution block that upsamples the feature map from 64 channels to 32 and doubles the spatial resolution with a kernel size of 2 and a stride of 2. It performs another batch normalization after the upsampling. Next, there is a second skip connection and convolution layer that adds the feature map to the upsamples tensor and then performs another 3x3 convolution with 32 input and output channels. Again, another batch normalization and ReLU are applied to improve convergence.

The final output layer of the architecture is a convolution layer that reduces the output to the required number of classes for the dataset, which is in this case 21 for PASCAL VOC. It uses a kernel size of 1x1 to output a segmentation mask where each pixel corresponds to a class.

The model then initializes its weights and biases of the layers using Kaiming Normal Initialization. This technique adjusts the initial weights of neural network layers to make training more efficient by reducing the likelihood of the vanishing gradient problem.[2] In the initialize weight's function, the first loop goes through all the layers of the model and returns an iterator over all the submodules of the model. It then checks if the current module is a convolutional layer and then initializes the weights of the layer using Kaiming Normal based on the number of output channels, and the activation function. It checks the bias term and if it exists its set to 0 to ensure they start at a neutral unbiased value. If it's not a convolutional layer, then it’s deemed a batch normalization layer, and the weight of batch normalization is set to 1 and the bias term is set to 0. This technique is super important for the model because when layers include a ReLU, they could undergo vanishing gradient and by setting the weights to appropriate values limits the odds of it occurring.

Knowledge Distillation

CombinedLoss()

CombinedLoss() is a hybrid loss function that combines cross-entropy loss and dice loss, with adjustable weights for each type.

[2] https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/

Cross-entropy loss is used for multi-class classification problems, and it measures the difference between the predicted probability distribution (the model's output) and the actual target distribution (the ground truth). The cross-entropy loss for a single prediction is mathematically calculated as:

$$L_{CE} = -\sum_{c=1}^{C} t_c \log(p_c)$$

where C is the number of classes, t_c is the target class label for class c, and p_c is the predicted probability for class c. In this labs case, the team ignores pixels with no label during the loss computation.

Dice loss is based on the dice coefficient, and it measures the overlap between two sets. The first being the predicted mask (probabilities) and the second being the ground truth mask. The dice coefficient or a single class is mathematically calculated as:

$$Dice(A,\ B) = \frac{2\ \cdot |A\ \cap\ B|}{|A| + |B|}$$

where A and B are the two sets as defined before, $|A\ \cap\ B|$ is the intersection of the sets, and $|A|$, $|B|$ are the cardinalities. The team implemented the dice loss by letting *probs[:, i]* be the predicted probability for class i for each pixel, and *targets_one_hot[:, i]* be the one hot encoded ground truth mask for class i. For each class i, the team calculated the dice score as:

$$Dice_i = \frac{2 \cdot \sum(probs_i \cdot onehottarget_i)}{\sum(probs_i) + \sum(onehottarget_i) + \in}$$

where $\in$ is a constant of $1e - 6$ to avoid edge cases where a division by zero may occur. This results in the final dice loss being calculated as the average dice score across all classes:

$$L_{Dice} = 1 - \tfrac{1}{C}\textstyle\sum_{i=1}^{C} Dice_i.$$

The combined loss is then a weighted sum of these two losses. It is computed by:

$$L_{combined} = \lambda_{CE} \cdot L_{CE} + \lambda_{Dice} \cdot L_{Dice}$$

where $L_{CE}$is the cross-entropy loss, $L_{Dice}$is the dice loss, $\lambda_{CE}$and $\lambda_{Dice}$are the weights of the losses totalling to 1. For the purposes of this lab, they are both equally weighted at 0.5.

Knowledge Distillation Loss

The teams code employed both response-based and featured-based knowledge distillation, allowing the WannabeUNet architecture to be trained in three modes. These modes include vanilla (without distillation), knowledge distillation (response-based) and feature distillation (feature-based). These training modes can be specified through the *–mode* argument in the train and test scripts. It determines how the loss is computed.

In response mode, the student model (WannabeUNet) learns from the softened outputs (logits) of the teacher model (Pre-trained ResNet50). This distillation process focuses on aligning the probabilities of the predicted classes. It assists the student model in replicating the teacher's decision-making process.

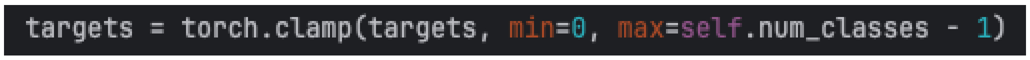
In feature mode, WannabeUNet is guided to mimic the intermediate feature representations of the teacher model. To ensure compatibility, it adapts the dimensions of the students features to match the teachers using a 2D convolutional layer when necessary. The feature similarity is then calculated using cosine similarity between normalized feature maps. This process helps the student model learn to replicate detailed patterns in the teacher's intermediate layers, ensuring the student can capture high level semantic content such as edges and textures.

The following two sections will walk the reader through the loss calculation for each mode and how knowledge distillation was employed in the code via mathematical interpretations.

Feature Based KD

Firstly, the team ensured the target labels were within the valid range of class indices. Loss functions expect targets to be in this range, so any unexpected values must be stopped before an error is produced. Mathematically this is expressed as:

$$targets = min(max(targets, 0), num\_classes - 1)$$



```
targets = torch.clamp(targets, min=0, max=self.num_classes - 1)
```

Figure 1: Code snippet that ensures the target values are within the valid range of class indices by clamping them between 0 and num_classes-1

Next step in the distillation process is to ensure the students feature map aligns with the teachers feature map. This is in regard to both channel dimensions and spatial dimensions. To adapt the number of channels to match, the code performs a 1x1 convolution. This can be presented mathematically as:

$$\overline{F_s} = W * F_s$$

where $\overline{F_s}$ is the transformed student's feature map, W is the learnable weight matrix of the 1x1 convolution, $F_s$is the feature map, and $*$ denotes the convolution operation. To ensure the spatial dimensions of the two maps match, a bilinear interpolation is performed to resize feature maps. This operation is described as:

$$\overline{F_s} = Interpolate\left(F_s, size(F_t)\right)$$

where $\overline{F_s}$ is the resized student map feature, $F_s$is the student's feature map, F_t is the teacher's feature map, and taking the size of F_t is the target spatial size that we want the student to take on.

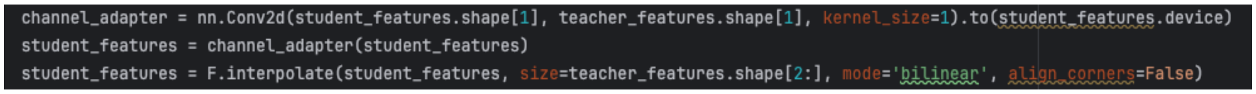

```
channel_adapter = nn.Conv2d(student_features.shape[1], teacher_features.shape[1], kernel_size=1).to(student_features.device)
student_features = channel_adapter(student_features)
student_features = F.interpolate(student_features, size=teacher_features.shape[2:], mode='bilinear', align_corners=False)
```

Figure 2: Code snippet that applies channel adapter, and bilinear interpolation

Once the feature maps align, it is key to reshape the feature maps into 1D vectors and then normalize them to have unit length. In this case, normalization is performed to ensure that the feature maps can be compared in terms of their direction and not their magnitude. The *.view* function reshapes both student and teachers' features so that the dimensions are *batch size x flattened feature map size.* The *F.normalize* function is then used to normalize the flattened feature maps across each vector. This normalization process uses L2 norm (Euclidean norm) which is computed as:

$$\overline{F_s} = \frac{F_s}{|F_s|}, \overline{F_t} = \frac{F_t}{|F_t|}$$

where $\overline{F_s}$, $\overline{F_t}$ are the normalized feature maps for student and teacher, F_s, F_t are the original feature maps, and $|F_s|$, $|F_t|$ are their magnitudes.

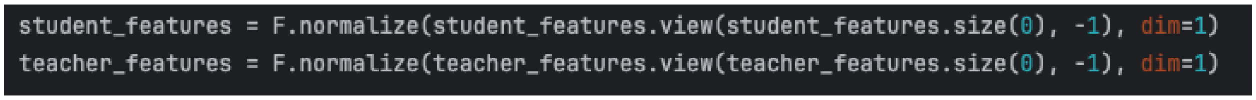

```
student_features = F.normalize(student_features.view(student_features.size(0), -1), dim=1)
teacher_features = F.normalize(teacher_features.view(teacher_features.size(0), -1), dim=1)
```

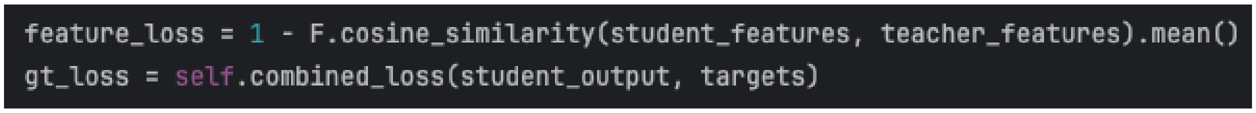
Figure 3: Code snippet that normalizes both student and teacher features

Now that the processing steps are done, the cosine similarity loss calculation can begin. Cosine similarity measures the similarity between the two vectors based on the cosine angle between them. If the vectors are pointing in the same direction, then the value will be closer to 1, if they are orthogonal then it will be 0, and if they are opposite directions then it will be –1. Cosine similarity is crucial as it quantifies the angular distance which can be used to compare feature representations. Since both vectors are normalized, the mathematical formula simplifies to:

$$CosSim(\overline{F}_s, \overline{F}_t) = \frac{\overline{F_s}}{|\overline{F_s}|} \cdot \frac{\overline{F_t}}{|\overline{F_t}|} = \overline{F_s} \cdot \overline{F_t}$$

From here, the cosine similarity loss can be calculated by taking the mean of the cosine similarity across all features in the batch and subtracting it from 1. This is important to encourage the student model to learn the same feature representations as the teacher model, and thus minimizing this loss will ensure the student produces similar intermediate features as the teacher. Cosine similarity loss can be found with:

$$L_{feature} = 1 - mean\left(CosSim(\overline{F_s}, \overline{F_t})\right)$$



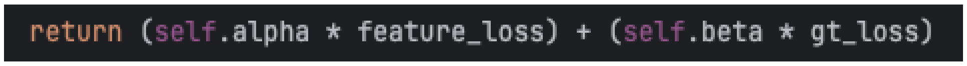
```
feature_loss = 1 - F.cosine_similarity(student_features, teacher_features).mean()
gt_loss = self.combined_loss(student_output, targets)
```

Figure 4: Code snippet that calculates feature loss and cosine similarity

After retrieving the cosine similarity loss, the final total loss can be calculated by combining the ground truth loss that was calculated in the combined_loss() function and the feature loss in the form of a weighted sum. This total loss can be calculated with:

$$L = \alpha \cdot L_{feature} + \beta \cdot L_{GT}.$$

Both of these weights are set to 0.5.



```
return (self.alpha * feature_loss) + (self.beta * gt_loss)
```

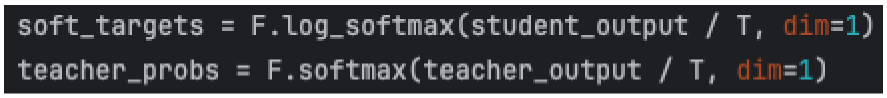
Figure 5: Code snippet that returns the total loss as a weighted sum

Response Based KD

After clamping the target labels to the necessary range as explained in the feature-based knowledge distillation. The model uses temperature scaling to scale the logits from the student and teacher models. This smooth out the probability distribution of the teacher model making it less confident, resulting in the student learning from more nuanced information. Mathematically, both logits are scaled as:

$$p_s = softmax\left(\frac{z_s}{T}\right), p_t = softmax\left(\frac{z_t}{T}\right)$$

where z_s, and z_t are the logits from the student and teacher models and T is the temperature. The division of the logits by the temperature reduces the difference between them making the SoftMax output smoother, and as said earlier this lets the student model learn richer information from the teacher's output.



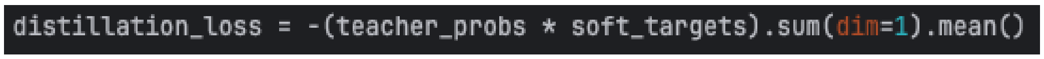
```
soft_targets = F.log_softmax(student_output / T, dim=1)
teacher_probs = F.softmax(teacher_output / T, dim=1)
```

Figure 6: Code snippet applying temperature scaling to outputs of models

Now the response-based distillation loss can be calculated. It measures the difference between the teacher's probability distribution and the students' soft targets. This is done using KL divergence, mathematically defined as:

$$L_{KD} = -\sum_i p_t(i) \log\left(p_s(i)\right)$$

where $p_t(i)$ is the teacher's probability for class i and $p_s(i)$ is the students.



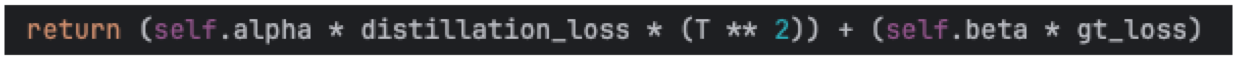
```
distillation_loss = -(teacher_probs * soft_targets).sum(dim=1).mean()
```

Figure 7: Code snippet that calculates the distillation loss

Once the distillation loss is calculated, the total final loss can be found via a weighted combination of the distillation loss and the ground truth loss. This is given by:

$$L = \alpha \cdot L_{KD} \cdot T^2 + \beta \cdot L_{GT}$$

where α and β are scalars to control the weights of each loss, and T^2 is scaling the distillation loss with respect to the temperature.



```
return (self.alpha * distillation_loss * (T ** 2)) + (self.beta * gt_loss)
```

Figure 8: Code snippet that calculates final loss in knowledge distillation using a weighted sum

Hyperparameters

The team trained the model in three different modes, vanilla (no distillation), response based knowledge distillation, and feature based knowledge distillation. Vanilla is a regular training mode without any distillation, it simply trains WannabeUNet on the PASCAL VOC dataset.. Response based uses ResNet50 as a teacher model to the WannabeUNet student model for soft label distillation. Feature based uses ResNet50 for intermediate feature representations to guide the student model. All models were trained for 25 epochs, with a batch size of 16, and with a learning rate of 0.0001. The mode, epochs, batch size and learning rate can be modified via the training and testing scrips by with their corresponding parser arguments.

The team chose 25 epochs as it demonstrated no overfitting, and the model showed signs of convergence in a reasonable amount of training time. A batch size of 16 was selected as that was the maximum amount before the GPU ran out of dedicated memory for the task, and did not cause any overfitting. The standard learning rate was chosen as it offered a good trade-off between learning speed and a stable gradient descent. This resulted in training durations listed below.

Mode Type	**Training Duration**	**Epochs**	**Batch Size**
Vanilla (No Distillation)	37 minutes and 46 seconds	25	16
Response Based Knowledge Distillation	48 minutes and 54 seconds	25	14
Feature Based Knowledge Distillation	1 hour, 12 minutes, and 35 seconds	25	16

An Adam optimizer was used as it is well suited for segmentation tasks where there are large datasets and noisy gradients. It is also helps converge faster by adapting to the dynamics of the problem which the team thought was very important for a knowledge distillation model.

Cosine annealing with warm restarts scheduler was used to adjust the learning rate during training. Cosine annealing refers to gradually reducing the learning rate in a cosine shaped curve, and warm restarts means resetting the learning rate back to a greater value at certain intervals. After a restart occurs it would be annealed again following the cosine schedule. This was chosen as the team believed in traditional learning rate schedulers where there is constant decay, there may be a risk of getting trapped in a local minima or saddle points, but by periodically resetting this we can escape from them and continue to generalize and improve performance.

As explained earlier, the loss functions used was a combination of dice loss and cross entropy loss, both weighted at 0.5.

The training also used gradient clipping to mitigate any issues that could arise during backpropagation when a gradient gets very large. If the gradient surpassed the threshold of 1.0 it would get scaled down.

As seen below, there are three loss plots, one for each of the modes. All three loss plots have the training loss below the validation loss which means it barely generalizes to unseen data, and it did not learn much more than the basic trends of the training data. This means the models are underfit and could possibly use a more complex architecture to pick up more features from the data.

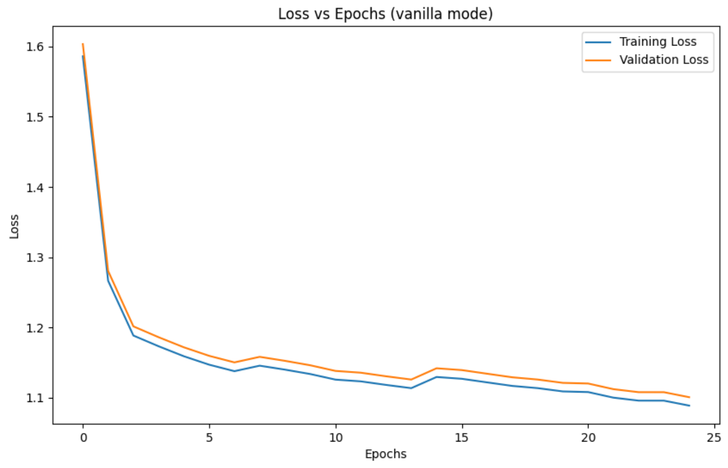


Figure 9: Loss Plot of Vanilla Model

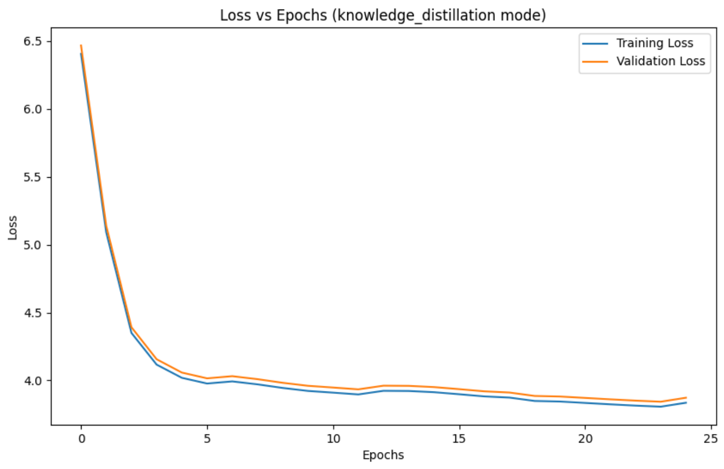


Figure 10: Loss Plot of Knowledge Based Model

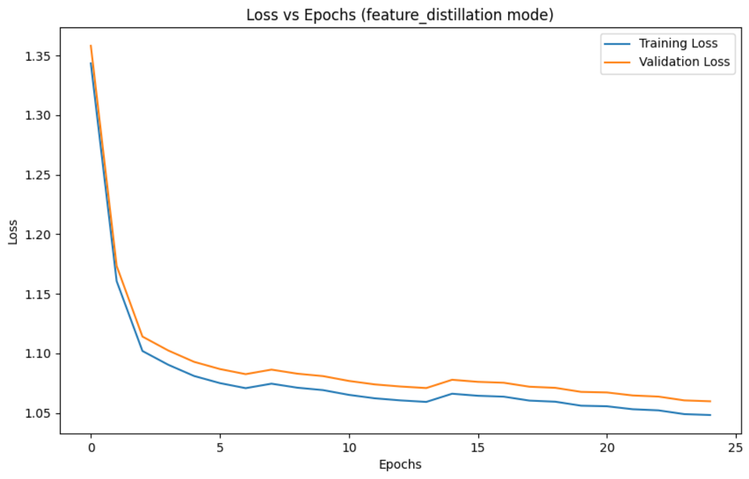


Figure 11: Loss Plot of Feature Based Model

Experiments

The model was trained on an NVIDIA 4070 Super GPU in a system with 32GB of DDR5 RAM at a speed of 6000MHz, and an AMD 7800x3D CPU with 8 cores and 16 threads. The GPU contains 7168 CUDA (Compute Unified Device Architecture) cores which are the basic processing units in NVIDIA GPUs that are optimized for parallel processing which enables the GPU to execute multiple threads at the same time. Since CNNs involve many matrix and tensor operations, these CUDA cores allow them to operate in parallel which speeds up the model training process. The GPU also contains 568 4th generation AI TOPS Tensor Cores which are specialized hardware units that are designed specifically for deep learning tasks. These cores allow models to compute calculations with reduced precision without sacrificing accuracy leading to faster computations and lower memory usage. This can be done because the tensor cores support mixed precision training. Furthermore, tensor cores are designed to perform matrix multiplications more efficiently than CUDA cores, leading to faster training times. The GPU uses both types of cores in parallel, CUDA for general purpose computations and tensor for matrix multiplications.

Below is a table that shows the time performance for both training and testing in each mode. It is measured in msec/image, so it shows the amount of time it takes for the model to process a single image during both training and testing.

Mode Type	**Training Time Performance**	**Testing Time Performance**
Vanilla (No Distillation)	31.12 msec/image	0.23 msec/image
Response Based Knowledge Distillation	40.29 msec/image	0.32 msec/image
Feature Based Knowledge Distillation	59.81 msec/image	0.24 msec/image

Below is a table that shows the mIoU, # of parameters, and inference speed for each mode. The total number of parameters seen in the table below refers to the total count of learnable weights and biases that the model will adjust during training. The inference speed is the reciprocal of the time performance in msec/image. It displays how many images the model can process per second. MIOU was found by calculating the mean intersection over union by calculating the intersection which is the common pixels between the predicted and ground truth segmentation maps and the union which is all unique pixels in either the predicted or ground truth segmentation maps. Mathematically this is defined as:

$$IoU = \frac{Intersection}{Union} = \frac{|A \cap B|}{|A \cup B|}$$

where A is the predicted set of pixels and B is the ground truth set of pixels. For multiple classes this can be averaged resulting in the mIoU defined as:

$$mIoU = \frac{1}{C}\sum_{i=1}^{C} IoU_i$$

where C is the number of classes and IoU is the IoU for class i.

Mode Type	**mIoU**	**# Parameters**	**Inference Speed**
Vanilla (No Distillation)	0.05381	182,005	4347.8 images per second
Response Based Knowledge Distillation	0.05729	182,005	3125 images per second
Feature Based Knowledge Distillation	0.05434	182,005	4166.7 images per second

As seen in this table above, there is a small but distinct difference in the mIoU between each mode. In the next section, the results will be thoroughly explained but for now note that without distillation got the lowest mIoU but performed the fastest.

Some of the results from each model can be seen below. All models struggles heavily as seen in the two poor mIoU cases, where it could barely highlight if at all the required targets. For the good mIoU cases the model chose not to predict any and since most of the image was background and classless it got a good result, but it doesn’t mean the model performed well.

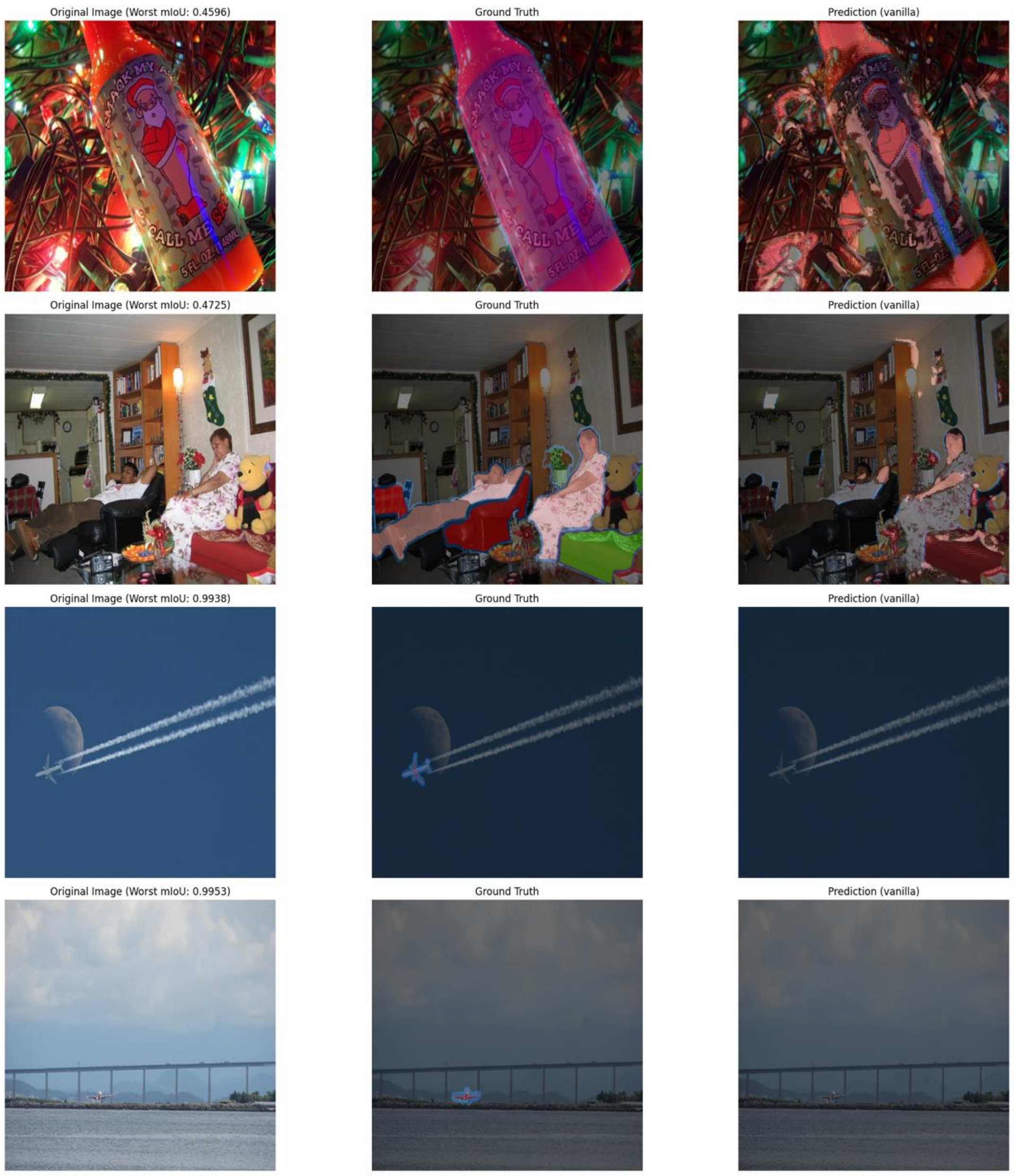

Original Image (Worst mIoU: 0.4596)
Ground Truth
Prediction (vanilla)
Original Image (Worst mIoU: 0.4725)
Ground Truth
Prediction (vanilla)
Original Image (Worst mIoU: 0.9938)
Ground Truth
Prediction (vanilla)
Original Image (Worst mIoU: 0.9953)
Ground Truth
Prediction (vanilla)

Figure 12: Results for Vanilla Model

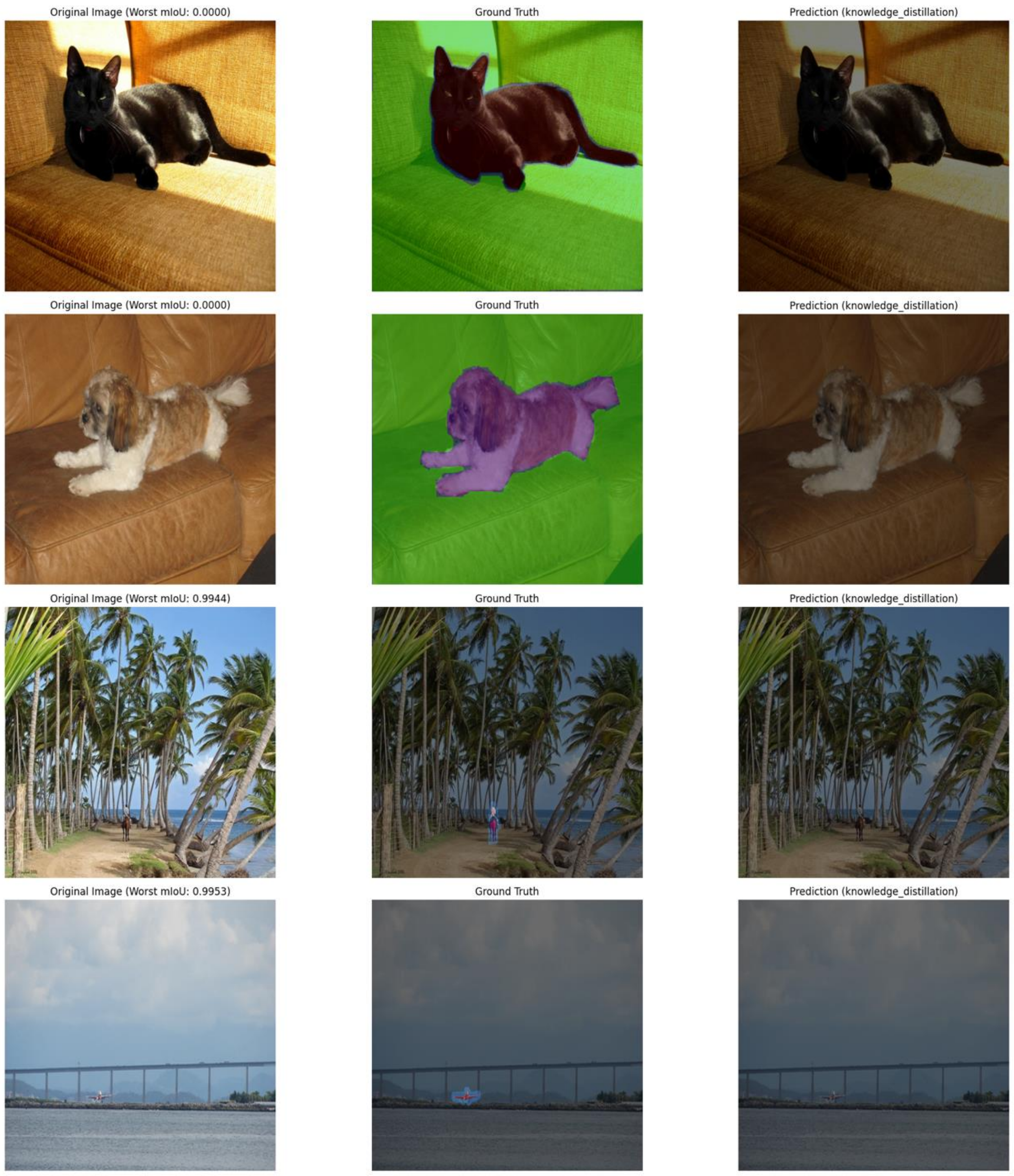

Original Image (Worst mIoU: 0.0000)
Ground Truth
Prediction (knowledge_distillation)
Original Image (Worst mIoU: 0.0000)
Ground Truth
Prediction (knowledge_distillation)
Original Image (Worst mIoU: 0.9944)
Ground Truth
Prediction (knowledge_distillation)
Original Image (Worst mIoU: 0.9953)
Ground Truth
Prediction (knowledge_distillation)

Figure 13: Results for Response Based Model

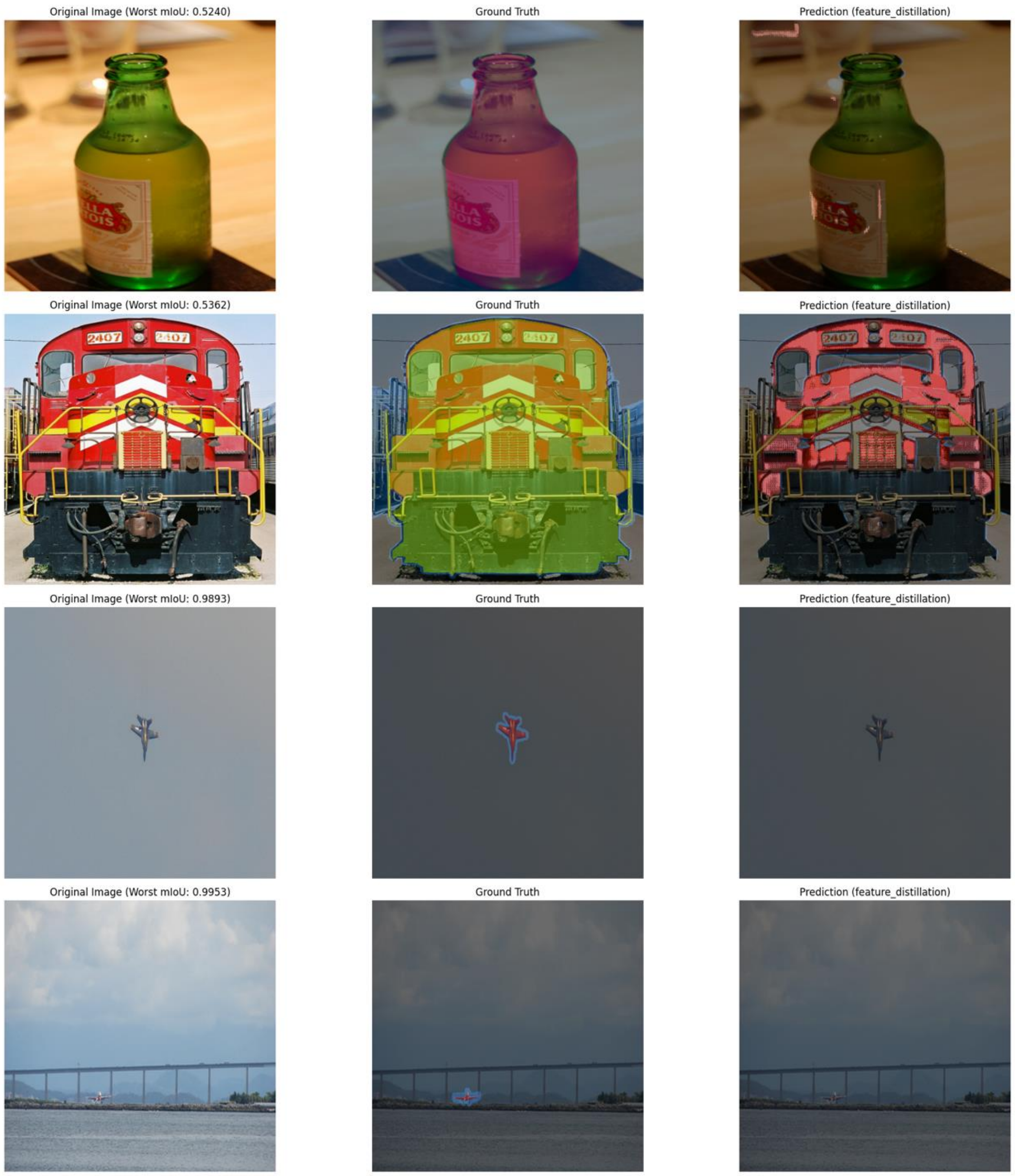

Original Image (Worst mIoU: 0.5240)
Ground Truth
Prediction (feature_distillation)
Original Image (Worst mIoU: 0.5362)
Ground Truth
Prediction (feature_distillation)
Original Image (Worst mIoU: 0.9893)
Ground Truth
Prediction (feature_distillation)
Original Image (Worst mIoU: 0.9953)
Ground Truth
Prediction (feature_distillation)

Figure 14: Results for Feature Based Model

Performance

The models performed as expected, in the sense that the different knowledge distillation techniques showed modest improvements in the accuracy (mIoU) and resulted in a slower inference speed. The Vanilla model had the fastest inference speed as it was the simplest and didn’t undergo any distillation which allowed it to perform faster. The trade-off was it had the lowest mIoU. The knowledge based model barely outperformed the feature based model but they both did better than the Vanilla version and performed slower. The knowledge based model performed ~1000 images per second slower than feature based and ~1200 images per second slower than the Vanilla. Both distillation models can be deemed beneficial and with a bit more tuning to the model architecture, it is believed they would be greatly beneficial. The unexpected portion of its performance was how inaccurate it was at image segmentation. As mentioned earlier the best cases were when no guesses occurred, and it was only deemed accurate since the background was classified as right. The bad guesses it picked up a tiny piece of the object or was completely off and picked up a fully wrong object for example classifying a box as a chair.

The main challenge the team experienced was shape mismatches between student and teacher models. As mentioned earlier the channel and spatial dimensions of two layers are not always going to match and this caused the team a great deal of trouble. To fix this problem the team had to change the shape of the features from the layers that were extracted so that they matched the teachers’ layers. This was done by first using print statements to print the tensor shapes after a specific layer was extracted. This told the team whether they were matching or not and then allowed the team to perform the resizing when they knew the error occurred. The second major challenge was balancing a small model with less parameters versus using a larger model but taking a longer time to train and model. In the end this was solved unsuccessfully as the model ended up being underfit which is a challenge with smaller models as they have limited capacity and lack the ability to capture complex data. In the future the architecture will be changed to a bigger but still compact version of UNet with more layers to ensure more complex relationships are captured.