Homework 2

Out: Friday, Feb 24

Due: Thursday, March 23 @ 5:00 pm EST

2.1 – Describe the instance segmentation problem. What are the inputs and outputs of the task? What supervision do we need?

Instance segmentation is a computer vision technique to identify and delineate instances of a particular object within an image or video stream, which plays a crucial role in enabling robots to effectively understand and interact with their environment. It is used in a wide range of applications, such as object recognition and manipulation, autonomous navigation, human-robot interactions, inspection, and quality control. In this assignment, we aim to perform instance segmentation by partitioning five objects and one background into classes associated with numbers from 0 to 5, where 0 represents the background, 1 represents a sugar box, 2 represents a tomato soup can, 3 represents a tuna fish can, 4 represents a banana, and 5 represents a bowl.

The inputs to the model are RGB images of 320 x 240 pixels with 3 color channels per pixel. Meanwhile, the model outputs the same pixel dimensions (320 x 240) but with 6 channels, each corresponding to the probability of that pixels being associated with the classes 0 through 5. The input data to the model are stored as PNG images, while the output is exported to image and point cloud files. Within the Python implementation, the inputs and outputs to the model itself are stored in PyTorch tensor format.

For most of this assignment, the type of supervision best suited to the problem is fully supervised learning, which is a type of machine learning where the algorithm is provided a complete set of labeled training data and the input data is paired with its corresponding output or target. This is an appropriate approach because the training set is relatively small and contains full ground truth information. However, weak supervision is used for estimating pose in the final part of the assignment, since the predicted masks generated by the model are used as part of the estimation process.

2.4.1 – Why are 3 x 3 kernels used in most of the convolutional layers, but 1 x 1 kernels are used right before the output?

Generally, the primary goal of a convolutional layer is to learn and extract features from input data by applying convolution operations using multiple filters or kernels. These layers enable the network to capture local patterns, provide translation invariance, share parameters, learn hierarchical features, and reduce dimensionality, all of which can facilitate the recognition of complex patterns and objects in tasks like image classification, object detection, and segmentation.

3x3 kernels are commonly used in convolutional layers because they strike a balance between:

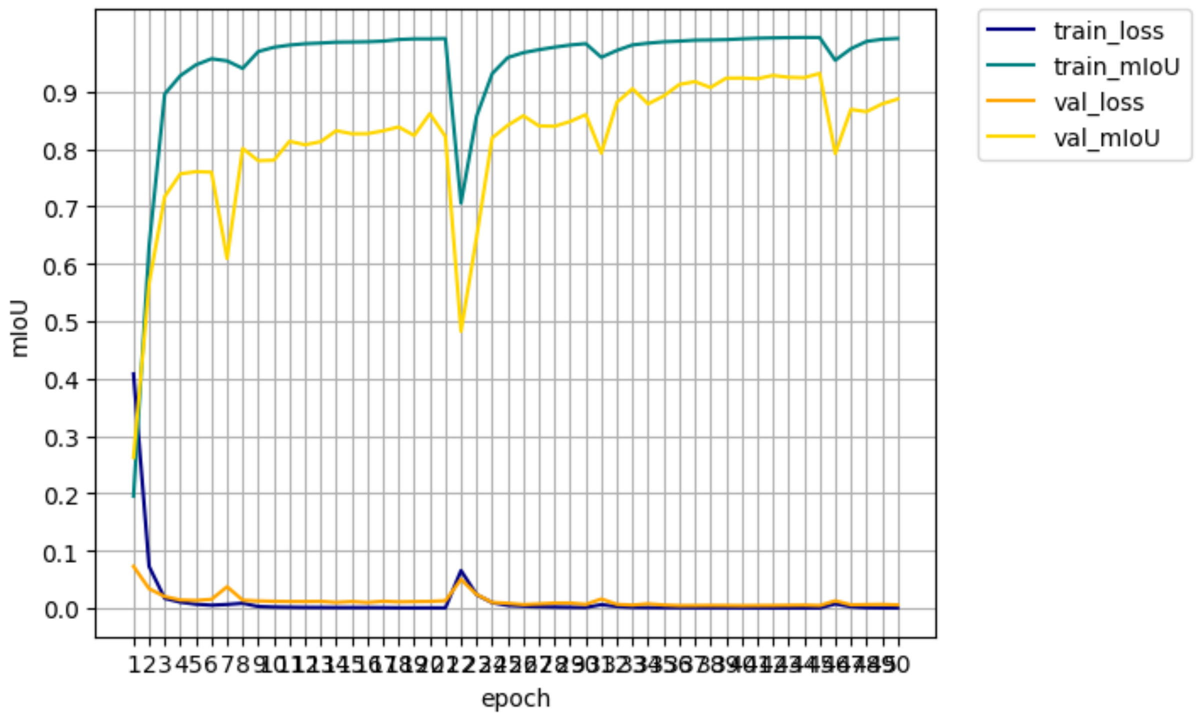
- Computational efficiency, as larger kernels require more computation;
- Parameter reduction, since larger kernels introduce more parameters and increase the risk of overfitting, while a kernel that is too small may underfit and fail to capture all useful information;
- The ability to capture local patterns in input data, which requires a kernel that is matched to the size of the pattern being captured.

In the final stages of the network, features tend to become more abstract. 1x1 kernels can be used to combine information from different feature maps and are more efficient because they require fewer parameters, which can help smooth feature maps and reduce noise, leading to more stable and reliable predictions.

2.4.2 – We know that input has 3 channels because the network takes in RGB images. But why does the output have 6 channels? What does each channel stand for? How to compare it with the 1 channel ground truth mask?

In the output, each of the 6 channels contains the probability of that pixel being associated with one of the 6 classes (0 through 5). The final step in generating a prediction is to choose the channel with the highest probability and assign the class that corresponds to that channel as the predicted label for the pixel. This can be stored either as a channel number or with one-hot encoding, whereby the channel with the highest probability is set to one and the others are set to zero. Either way, this prediction can be compared to the value in the ground truth mask, where the single channel contains the class number (0 through 5).

2.5 – Insert Learning Curve Plot



My best model reached a 93% mIoU on the validation set. This model uses a LeakyReLU with a larger dataset (training_scene = 60, num_observation = 20). Both the ReLU and LeakyReLU model consistently generates val_mIoU between 85-93% with a batch size of 8, learning rate of 0.001, and 15-50 epochs.

3.2 – Tune the parameters of the ICP method and describe what difference you observe. There are three tunable parameters:

Minimum Change in Cost: this parameter tunes the minimum change in cost for pairs of points that are rejected from consideration in the model. Lowering the threshold also lowers the Average Closest Point Distance (ACPD). I experimented with a number of thresholds (line 160 of icp.py), ranging from 10^{-4} to 10^{-10}. As mentioned in the assignment, the goal was to obtain ACPDs in the range of 10^{-4}. This was achieved for thresholds of 10^{-7} or less, though decreasing the threshold beyond 10^{-9} did not lower the ACPD significantly, likely due to diminishing returns and floating point rounding. Ultimately, a threshold of 10^{-7} was selected to meet the specification in the assignment.

Maximum Number of Iterations: increasing the number of iterations (line 160 of icp.py) can potentially lead to a decrease in the ACPD, as points within a cluster become more tightly grouped. However, increasing the number of iterations too much increases the risk of overfitting or converging to a suboptimal solution. In my tests, I found that increasing the number of iterations up to 50 yielded improved results (lower ACPDs), though improvements were relatively small beyond about 40 iterations. However, beyond 50 iterations, the output became inconsistent and showed no meaningful improvement. A value of 40 iterations was selected for this assignment.

Initial Transformation: I found that the best initial transformation (line 124 of icp.py) was the output matrix provided by the Procrustes method. I tested other initial transformations, including an identity matrix and a random matrix, but these tended to give inferior results (higher ACPDs). Thus, the initial transformation provided by the Procrustes method was selected for this assignment.

4 – Further improvements: For this assignment I experimented with a number of modifications, including adding dropout layers, adding batch normalization, adjusting hyperparameters (batch size, epochs, learning rate, etc.), changing the activation function (leaky ReLU instead of regular ReLU), using a weighted Adam optimizer (AdamW), increasing the number of training scenes, and increasing the number of observations. My generic, first run featured a regular ReLU with a batch size of 8, a learning rate of 0.001, and 50 epochs. This model typically returned greater than 80% accuracy but did not exceed 88%. When the other parts were substituted, performance improvements were marginal and the model often became unstable and started overfitting in fewer epochs. Likewise, dropout and batch normalization made the model relatively unstable with an elevated training loss. In general, the most successful combination included the leaky ReLU activation function with the generic Adam optimizer and a batch size of 8, a learning rate of 0.001, and 50 epochs, working with a larger dataset (60 training scenes, 20 number of observations). This combination produced a validation mIoU of 93%.

For each method you implement, write a brief description of how it works, parameters you use and special features in your design (if any).

---

camera.py > compute_camera_matrix()

This method computes the intrinsic and projection matrices. The formula for the intrinsic matrix is

$$\begin{bmatrix} f & 0 & w/2 \\ 0 & d & h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

where f is the focal length and x and y are respectively the width and height of the image. Meanwhile, the projection matrix is computed using the computeProjectionMatrixFOV() method in PyBullet.

dataset.py > class RGBDataset()

- Implemented the transform in the constructor as a composition of two operations:
  - First, convert to tensor;
  - Second, normalize for RGB.
- Implemented code to get the length of the dataset by counting the number of PNG files in the directory.
- Implemented code in the getitem dunder method to retrieve the requested RGB image as well as the associated ground truth mask if one is present. The result is returned as a dict. The RGB image is read with the read_rgb() method from the image.py file and then transformed, while the mask is read with the read_mask() method from image.py and converted to a tensor.

model.py > class MiniUNet()

Implemented the MiniUNet neural network as described in the diagram illustrating its architecture (page 7 of assignment handout). The network includes 10 2-D convolutional layers (Conv2d in PyTorch) as well as ReLUs between layers. There are also interpolation and concatenation layers present in the second half of the network.

segmentation.py > train()

The train method follows a standard workflow for training the neural network over one epoch:

1. Load a batch of training data.
2. Predict the output on the training set.
3. Compute the loss.
4. Compute the mIoU.
5. Zero the gradient and back-propagate to adjust the weights.
6. Load more training data and repeat from step 2 until all training data has been used.

A record of the loss history and mIoU is kept for each batch, and the average of each is returned by the method.

segmentation.py > val()

This method is almost the same as train(), except that it omits step 5. Hence, there is no back-propagation and no adjustment of weights (to this end, the code for the other steps in enclosed in a "with torch.no_grad()" block).

segmentation.py > main()

- The train, validation, and test datasets are created using the init method of the RGBDataset class (see dataset.py). The check_dataset() method was used to verify functionality, though it is not included in the final submission.
- PyTorch dataloaders are created for the three datasets. The check_dataloader() method was used to verify functionality, though it is not included in the final submission.
- The model is prepared by creating an instance of the MiniUNet class (see model.py).
- The model is trained and validated using the train() and val() methods. The maximum number of epochs was chosen to be 50, as this was found to provide a sufficient amount of training for the loss and mIoU curves to plateau and for the network to achieve a satisfactory performance on the test set.

icp.py > gen_obj_depth()

This method filters out the depth of the background (in all cases) as well as that of all other objects if a specific object ID is provided. This is achieved using the where() method in numpy.

icp.py > obj_depth2pts()

This method returns the point cloud for a given object by performing the following steps:

1. Get the filtered depth map just for the requested object using the gen_obj_depth() method.
2. Convert to a depth map to a point cloud using the depth_to_point_cloud() method.
3. Convert to world coordinated using the transform_point3s() method, with the transformation matrix obtained from the cam_view2pose() method.

icp.py > align_pts()

This method aligns the point clouds, first by attempting the Procrustes method in the trimesh package (the function returns None if the method encounters an error performing the linear algebra), and second using the icp registration method in the same package.

icp.py > estimate_pose()

Returns a list of object poses by doing the following for each object:

1. Retrieve the projected points using the obj_depth2pts() method.
2. Retrieve the sample points using the obj_mesh2pts() method/
3. Align the points using the align_pts() method and append the resulting pose to the list.

icp.py > main()

Scene estimation code was implemented in the for loop at the end of the method. For each scene, the depth and view matrix are loaded. The following steps are then performed for the predicted mask, as well as ground truth mask in the case of validation data only:

- The mask is loaded.
- The pose is estimated using the estimate_pose() method and saved.
- A ply of the predicted pose is exported. For validation data only, a ply of the ground truth is also exported.