Assignment 4 Report

Summary: A working image classification model that trains a neural network and also improves its performance and accuracy with unfamiliar data while avoiding overfitting. First, the InDomainDataset and OutDomainDataset components load labeled and unlabeled images into the learn function, and apply a transformation pipeline to prepare for the model. The FastCNN class is a custom model we have implemented that can take an image as input, and produce a prediction of which category the input belongs to. The learn function handles the training, by repeatedly feeding FastCNN the images to teach it how to classify them, and also update its parameters over many epochs. Finally, our compute_accuracy can evaluate how well our trained model performed, by comparing the predictions to the true labels, ensuring that FastCNN isn't just memorizing, but has learned to recognize the different categories. While our in-domain accuracy has only been able to reach an accuracy of over 50% through 40-50 epochs, this still shows a model that is learning, and hopefully has the potential to achieve significantly higher performance through additional fine tuning and augmentation.

InDomainDataset: This class is a custom PyTorch Dataset that is responsible for loading and managing the in domain training images. Originally, the dataset did not store images in RAM ahead of time, but rather, only the file paths and labels were stored, and each time __getitem__ is called, the image is read on demand from disk, loaded into RAM temporarily, and then transformed and passed to the model. This repeated process of disk access came at the cost of performance. To improve our current method, we implemented a RAM-based caching system. In the InDomainDataset, we added an if-statement in the dataset constructor to pre-load images during initialization, and store a unique version of them in RAM. By doing this, the dataset can return a clean copy of the original image for each training iteration without returning to disk. The deep copy ensures that random augmentations do not modify or corrupt the cached version, allowing the same base image to be reused safely in the following epochs. We implemented the logic for caching enabled into the training pipeline inside the learn function. Now, when we create dataset objects, we pass the argument cache=True.

FastCNN: This is a class we created that defines the neural network used in the model. It takes the number of classes for input (represented as nc), to build a series of convolutional and normalization layers to extract the features from the RGB in images before making a prediction. From line 102 - 107, we defined self.features where the initial line takes 3 input channels for each of the three colors, expand to 32 output channels using 3x3 windows, normalizes the 32 feature maps, adds non-linearity and reduces spatial resolution to make features more compact, and allow faster computation. Each following step is to increase the filter size, so that each deeper layer learns more complex visual features in an image, highlighting the difference from simple lines to more abstract shapes and textures. Originally, our model only increased the filter size up to 64, but we noticed a significant increase in our in-domain accuracy when we expanded our features to reach deeper layers, up to 256 without overfitting, granting the model more capacity to correctly distinguish among different classes by analyzing more complicated features.

After the convolutional feature extraction, we add two more lines. The first line, self.dropout = nn.Dropout(0.4) is to turn off 40% of the neurons, and this is to reduce overfitting by preventing our model from relying heavily on specific activations, and encourage it to learn from more generalized patterns from the images fed to it. We opted with roughly 40% as it provided a relatively strong regularization effect without making our model underfit, which is especially important when taking into consideration the small size of our dataset. The second line self.fc = nn.Linear(256, nc) takes 256 extracted features, and maps them to nc output values.

Lastly, the forward function defines how the data flows through the network during the prediction. First, the input image is passed through the feature extraction layers, then compressed using global average pooling, regularized with dropout, and fed into the connected layer to produce the class prediction scores.

Learn: Our learn function contains the full training process for our model by combining both in-domain and out-of-domain learning to improve generalization. We begin by selecting the computing device using PyTorch's device function, opting to use the GPU, but will switch to CPU if not available.

Next, we implemented an if statement that acts as a default transformation pipeline, by resizing each image to the same dimensions and converting it into a tensor. We chose this approach because convolutional networks require spatially consistent input sizes, and tensors are a requirement in terms of format for PyTorch operations. Without this, our model would have most likely struggled to batch all our data and be unable to generalize or learn specific patterns due to inconsistent feature scales.

In the data loading stage of the learn function, a few key factors contributed towards our overall in-domain accuracy. We enabled shuffling during each epoch so that like we explained earlier, the model would not try to memorize the sequence of images, but rather allow it to learn general characteristics within the data, to help us achieve stronger generalization and accuracy. Using the correct amount of batch size was also important, to stabilize gradient updates and allow our model to learn much cleaner features with less noise. We decided that 128 is the best size to perform on smaller sizes. It is here where we also enabled RAM caching, which preloaded all images into memory, eliminating the need for repeated disk reads and improving data delivery to the computing device. Because our training was no longer slowed down by disk access, we could run more epochs in the same amount of time, allowing the model to learn much more effectively. Finally, we defined a variable num_classes to be the length of the in-domain class index mapping, ensuring that our model's output layer exactly matched the number of categories in the dataset. This allowed the network to assign predictions correctly and prevented any misalignment between labels and model outputs, directly supporting higher in-domain classification accuracy.

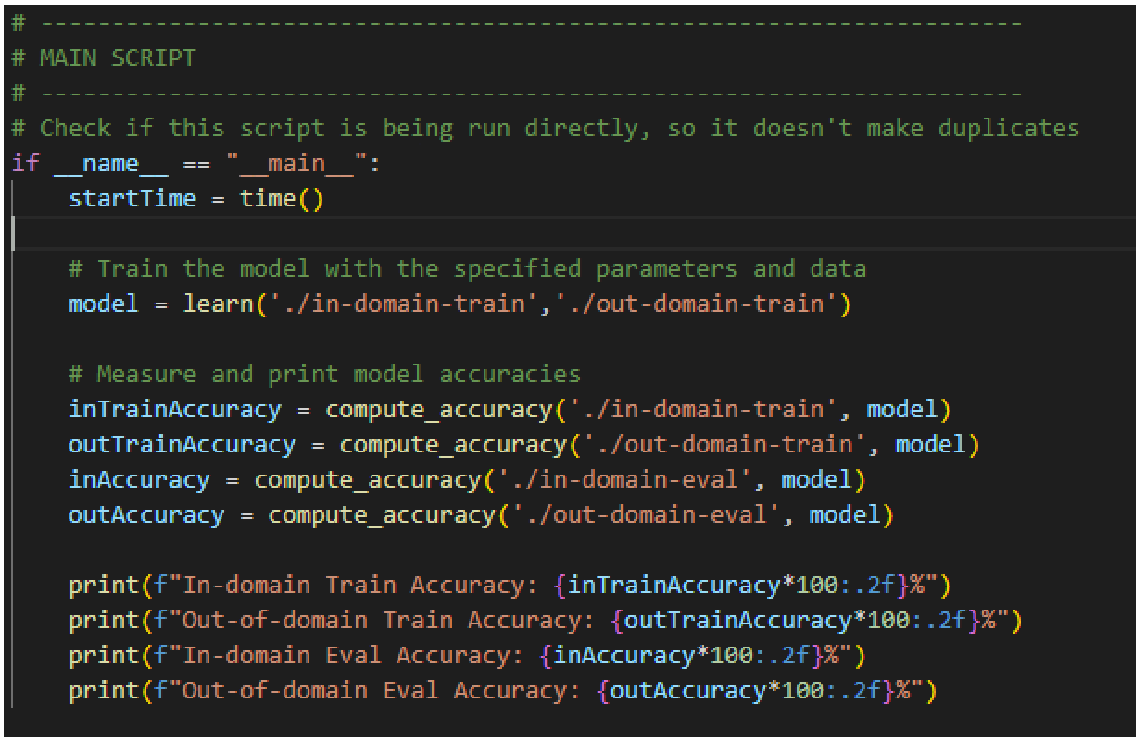
In the Stable FastCNN setup, there were several more design choices made to directly improve our in-domain performance. Initially, we chose to use SGD rather than Adam because it was simple, widely used and also known to work in many CNN architectures. We expected that its steady and controlled updates may avoid over adjusting our model, especially since our dataset was so small. We soon observed that SGD was converging much more slowly, and was also getting stuck in suboptimal points that led to lower accuracy. We decided to switch to Adam, which is capable of adapting the learning rate for each parameter automatically based on the magnitude of past gradients. Through this, we noticed that our model was able to learn much faster and efficiently. The transition to Adam allowed us to achieve higher in-domain accuracy within the same number of epochs used to loop the training. Another key factor was the Cross-Entropy loss, which we selected because it is the most effective loss function for multi-class classification tasks. It can directly measure how well our model predicts the correct class by penalizing incorrect predictions more blatantly than the ones that were somewhat close to our target. We considered alternatives such as Mean Squared Error, but found that they slowed down learning in classification problems, and may also cause our model to produce rather soft probability outputs. Through Cross-Entropy, our model was encouraged to make more confident predictions and strengthen the separation between classes and contributed to a noticeable improvement in our in-domain accuracy.

The last part of our training function was the training loop, which played the most direct role in terms of improving our in-domain accuracy. Our loop was designed such that each iteration begins by retrieving a new batch of labeled in-domain samples. Consistent exposure to supervised data was what allowed our model to continuously refine its understanding of which key shapes and features determined each class. Within the loop, we compute the predictions using a standard forward pass (logits_in = model(imgs_in)) so that gradients could flow cleanly through each layer of the network. For loss, we chose Cross-Entropy (loss = ce_loss(logits_in, labels_in)) rather than our previous method Mean Squared Error, simply because Cross-Entropy directly optimized the class probability separation and results in faster and reliable convergence for classification tasks. For our parameter updates, we used mixed-precision gradient scaling. We chose this design choice as without GradScaler, our model would be forced to compute each and every value in full precision, resulting in much slower training and even limiting the number of epochs we could run within our desired time constraints. Through the use of scaled gradients, we were able to allow our model to learn more efficiently and complete more full training cycles, leading to improved final in-domain accuracy.

Compute_Accuracy: This function was important for determining how we got our final in-domain accuracy. In our first if statement in the function, we only resize and convert our images into a tensor, with no other augmentations. We deliberately made this choice as we wanted a measure of how well the model could perform on unmodified in-domain data, not on distorted samples. Setting shuffle to false made the evaluation deterministic, and model.eval() disabled dropout and fixed the batch-norm statistics so that the predictions were stable.

Executable: Finally, this section includes the series of preprocessing and augmentation steps (under transform) applied to every training image before it is passed into the neural network. These transformations, such as the resizing, conversion to tensor and variations such as horizontal flips (recognize objects regardless of which direction they were facing), color changes (controlled changes in brightness, contrast, saturation and hue helped the model become less sensitive to lighting effects, and made less of an impact on the model than shapes and textures) and Gaussian blur helped simulate realistic visual diversity while preserving the correct class labels. We did a lot of experimenting here, trying out different augmentations that could help improve our accuracy, and were ultimately able to improve our initial accuracy of 35% up to 50%. That being said, there were a few we thought could be beneficial, but ultimately didn’t make much of an impact. Just to name a few, RandomResizedCrop and RandomPerspective would sometimes distort or crop out parts of the object which may have been important for generalization, this made training harder and slightly reduced our final in-domain accuracy. RandomVerticalFlip was too unrealistic for our data, as many objects don’t appear upside down in practice, flipping them vertically could potentially confuse the model. We also had a second GaussianBlur at one point, but realized it was unnecessary as adding more blur would degrade visual details without significant benefit.

Main Script: We have a main script beginning with **(if __name__ == "__main__":)** that serves as the main entry point of our script. The purpose is to ensure that the training and evaluation routines only run when the file is executed directly, rather than when it is imported as a module in another script. We have commented it out so the graders can run and test our code in their own environment without triggering a full training run.

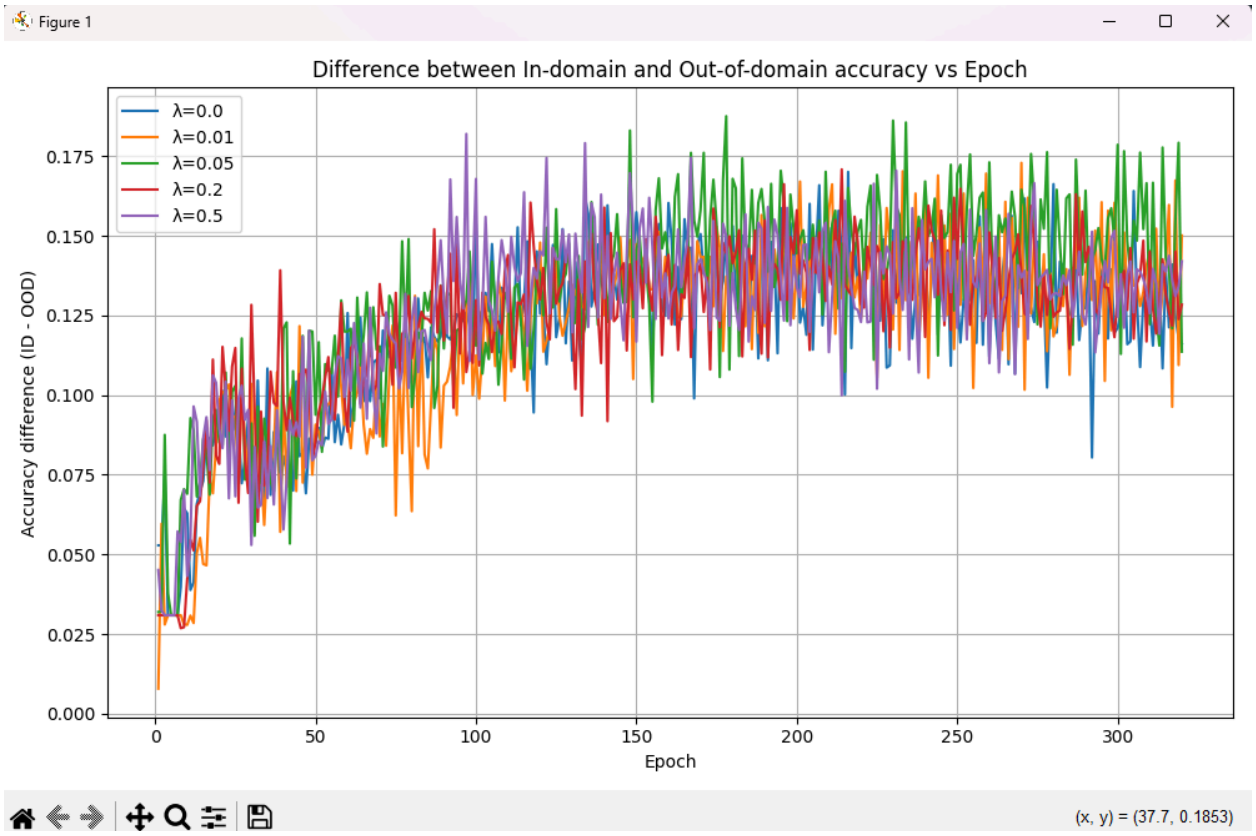


```python
# ---------------------------------------------------------------------
# MAIN SCRIPT
# ---------------------------------------------------------------------
# Check if this script is being run directly, so it doesn't make duplicates
if __name__ == "__main__":
    startTime = time()

    # Train the model with the specified parameters and data
    model = learn('./in-domain-train','./out-domain-train')

    # Measure and print model accuracies
    inTrainAccuracy = compute_accuracy('./in-domain-train', model)
    outTrainAccuracy = compute_accuracy('./out-domain-train', model)
    inAccuracy = compute_accuracy('./in-domain-eval', model)
    outAccuracy = compute_accuracy('./out-domain-eval', model)

    print(f"In-domain Train Accuracy: {inTrainAccuracy*100:.2f}%")
    print(f"Out-of-domain Train Accuracy: {outTrainAccuracy*100:.2f}%")
    print(f"In-domain Eval Accuracy: {inAccuracy*100:.2f}%")
    print(f"Out-of-domain Eval Accuracy: {outAccuracy*100:.2f}%")
```



To determine the best value for the out-of-distribution regularization weight, we trained the model for 300 epochs under five different λ settings:

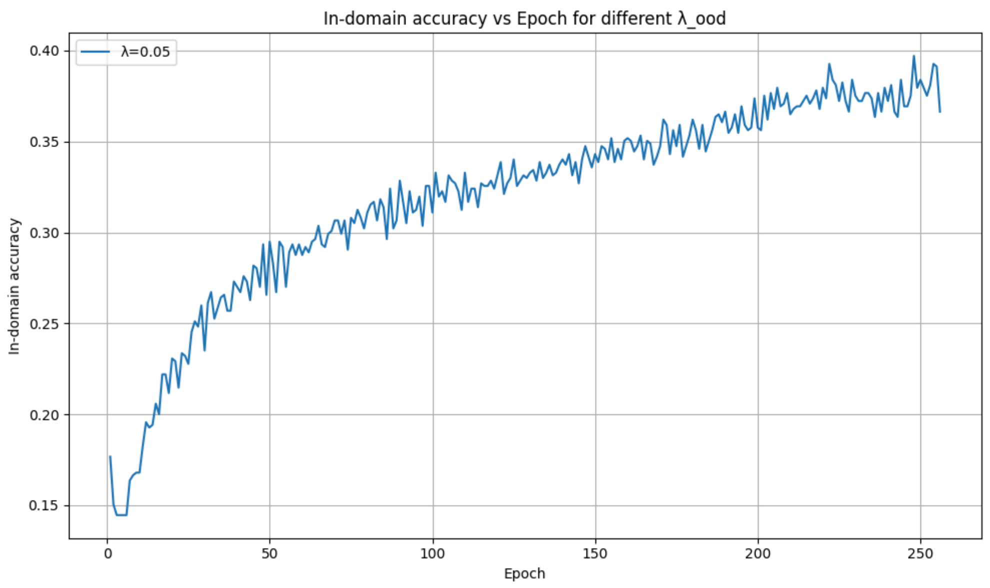
λ ∈ {0.0, 0.01, 0.05, 0.2, 0.5}

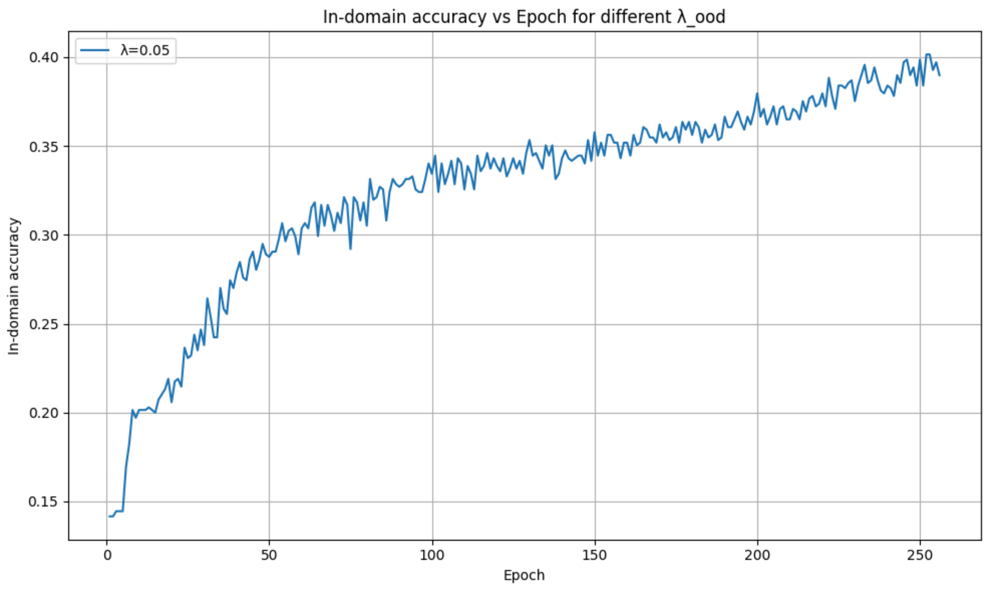
For each configuration, we recorded both in-domain and out-of-domain accuracies at every epoch. We then plotted the accuracy gap between in-domain and out-domain performance:

Gap = AccuracyID - AccuracyOOD

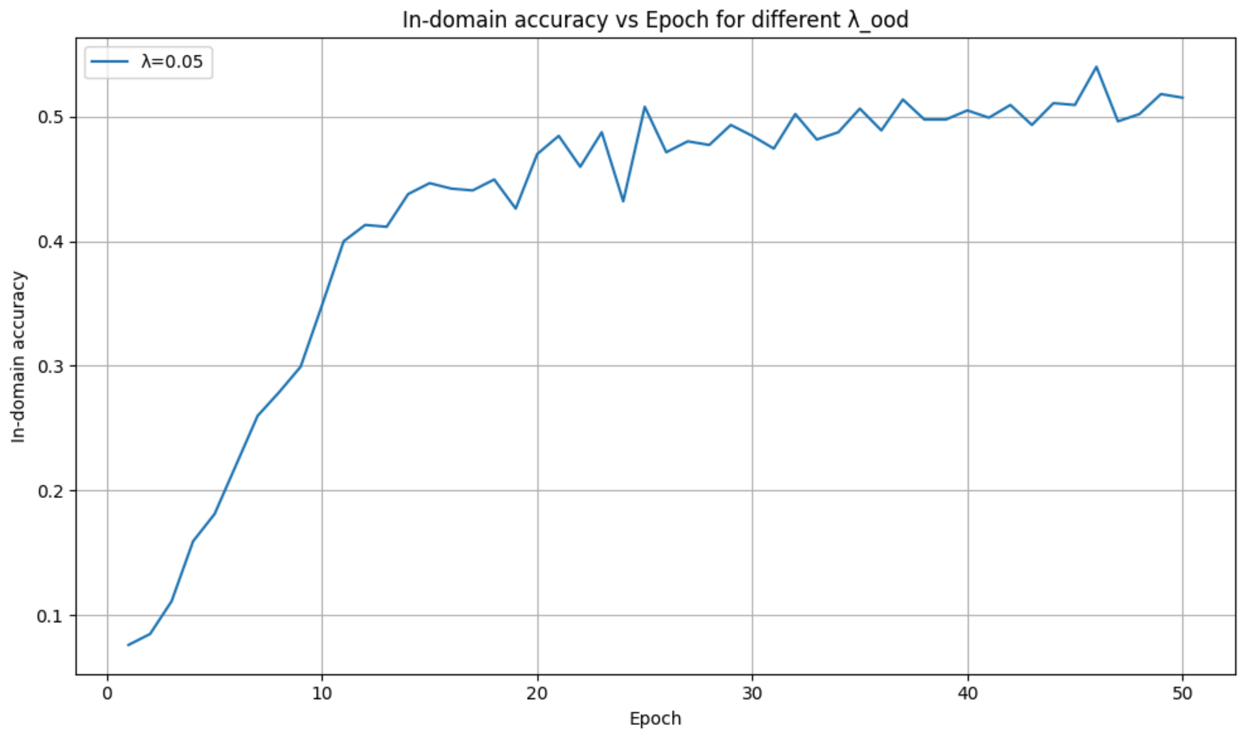
A smaller gap would indicate a better generalization and reduced sensitivity to domain shift.

As seen in the graph above, λ = 0.05 consistently produced the lowest and most stable ID–OOD accuracy difference after convergence, compared to other regularization strengths. Very small λ values showed weaker robustness, while large λ values (e.g., 0.5) introduced noise and slight performance drops.

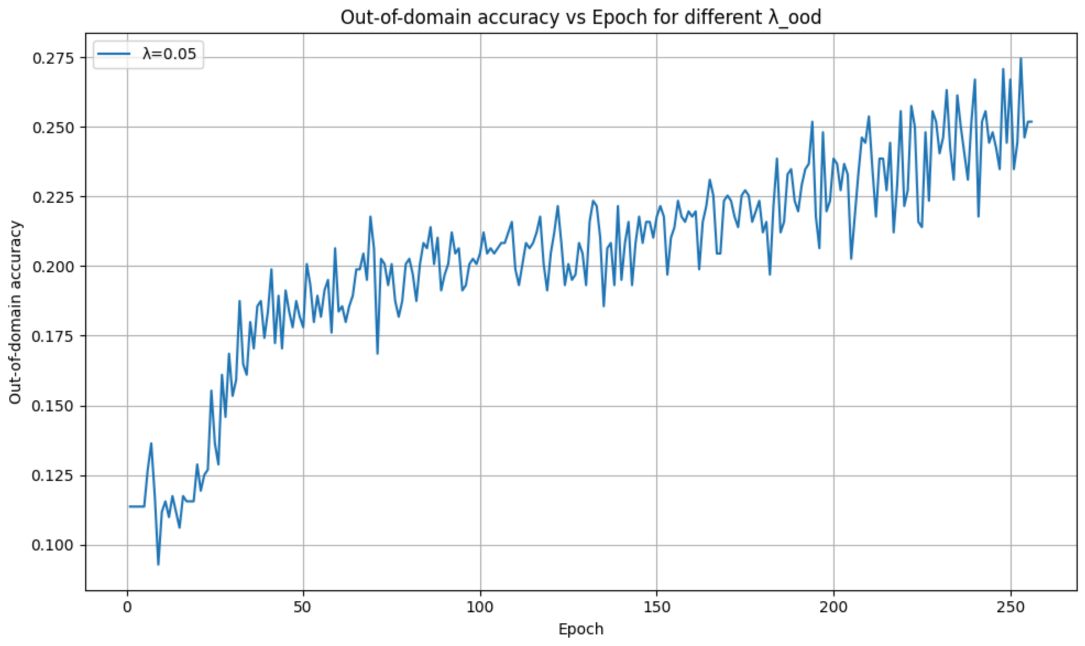


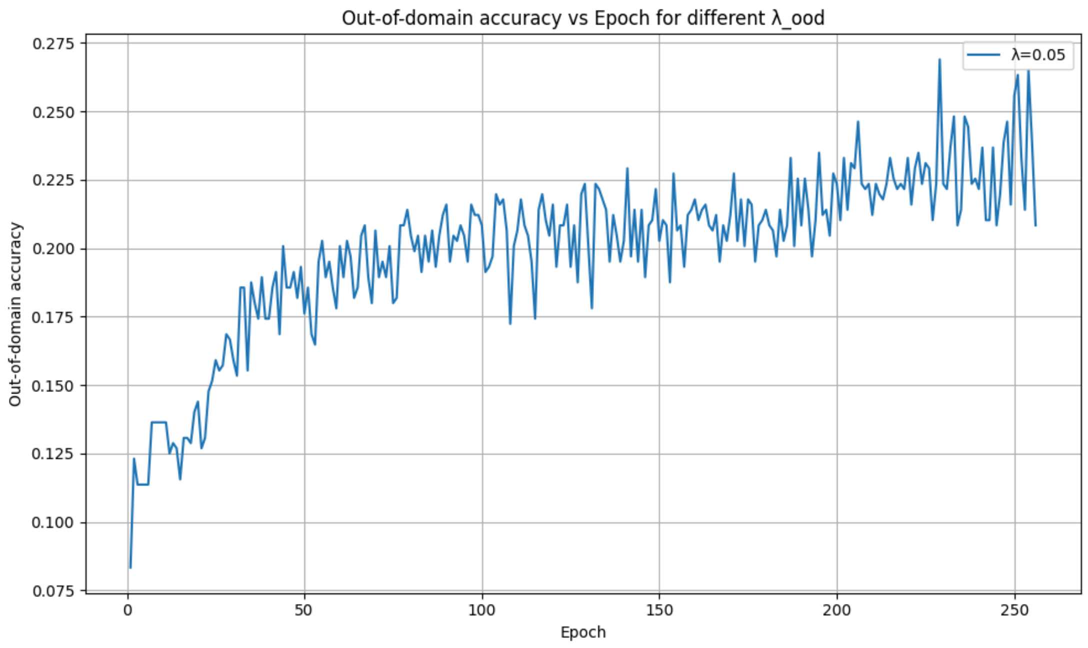


In the two graphs shown above, we recorded the in-domain accuracy over 300 epochs. The graph on the top was the accuracy when the model was trained with images resized to 256x256 resolution, while the bottom one was with 128x128 resolution. While both models show a similar trend in accuracy, the model trained with 128x128 images consistently achieved slightly higher accuracies through the same number of epochs, leading us to believe that the smaller image size had better generalization, on top of smaller computational cost. From this performance, we decided to continue training with 128x128 images.

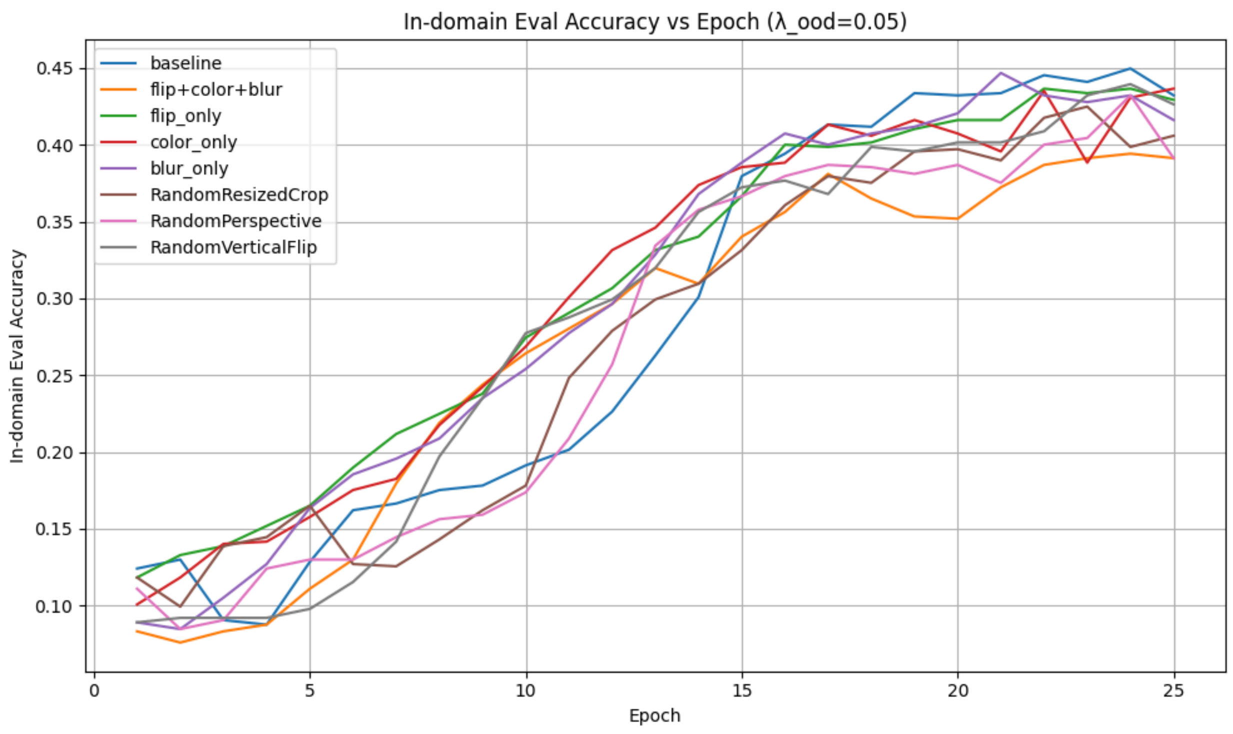


The figure above now represents the final in-domain accuracy over 50 epochs, using 128x128 image resolution. The difference here is that we have achieved much higher in-domain accuracy, using much less epoch cycles as compared to our graphs shown earlier. This significant change was primarily due to our improved caching, by optimizing how our data was loaded into memory, allowing our GPU to remain more fully utilized by minimizing I/O overhead.





The two graphs above represent the out-of-domain accuracy over 300 epochs for different input image resolutions. The top graph represents the training with 256x256 images, and the bottom shows the same experiment but with 128x128 images. In both graphs, we notice that the changes in OOD accuracy is rather minimal, on top of 128x128 being already faster and more efficient. The comparison allowed us to acknowledge that reducing the image res did not negatively impact out of domain generalization, and made us more confident to continue experimenting with this size.



Here is a graph regarding how well specific transformation choices we made impacted our models in-domain training accuracy. Both baseline and blur_only achieved the highest final training accuracies near the end of the epoch, indicating that slight degradations like blurring can improve robustness without altering key features. Other transformations like flip_only and color_only also show decent performance, while RandomPerspective and RandomVerticalFlip fell a little off behind the others, suggesting these transformations may have also introduced distortions not found in the original data.