CS 2420: Computing at Scale **(Due: 10/23/2024 at 11:59PM EST)**

Programming Assignment III

Instructor: H.T. Kung

Team: Anmay Gupta, Carlos Luz, Elvin Lo, Jaray Liu

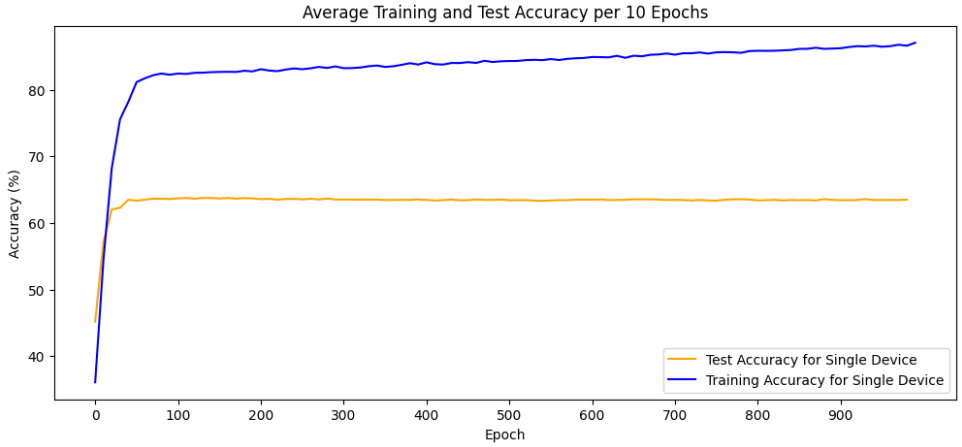
Please include the full names of all team members in your submission.

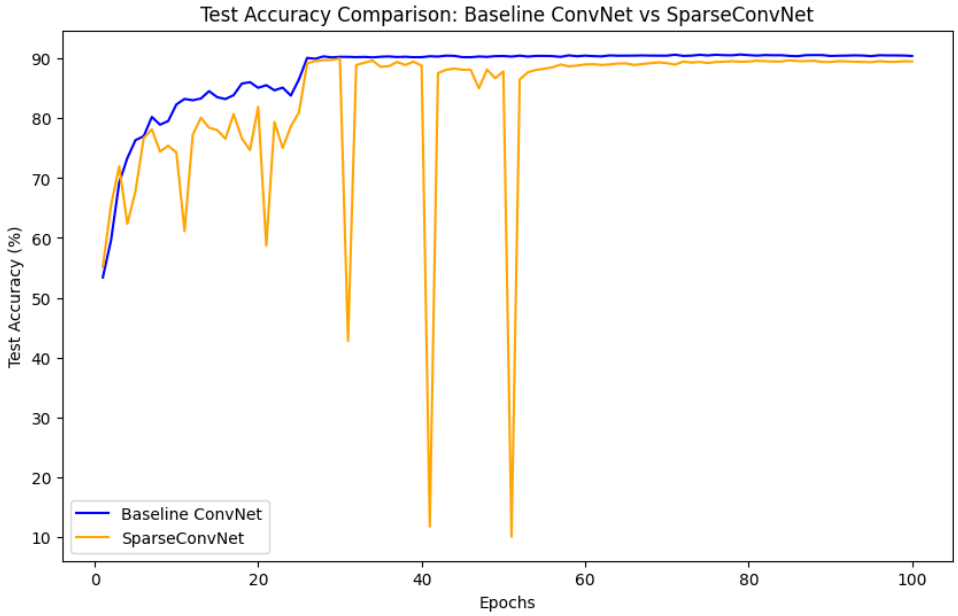
Part 1.1 (5 points)

1. In *Code Cell 1.3*, implement the function `iid_sampler` to generate **IID** (independent and identically distributed) samples from the CIFAR-10 training set. We will use this function to generate training subsets for multiple devices in **PART 1.2**.
2. In *Code Cell 1.4*, create a single device using the function `create_device`. This device should have 10% of the CIFAR-10 training set, obtained using `iid_sampler`.
3. Train the model for 1000 epochs using the specified parameters in *Code Cell 1.4* (similar to Assignment 1). The number of epochs is 10× greater due to the single device having only 10% of the data. Plot the test accuracy (`device['test_acc_tracker']`) over and comment on the classification accuracy compared to using 100% of the dataset as in Assignment 2.

(Solution)

1. [See Colab.]
2. [See Colab.]
3. Training our model for 1000 epochs, we see that the maximum accuracy gained is only 64.550%, while the baseline model shown in the second figure below can reach up to 92.5% test accuracy. Moreover, the single device model also did not achieve over 87% accuracy during training. This is because not only is our model only able to access 10% of the data, and thus unable to generalize to the entire dataset, but because our data is non-IID, some of the data that the model does have access to is difficult for the model to fit without overfitting as it's highly divergent from other parts of the 10% the model can access.





Part 1.2 (10 points)

1. In *Code Cell 1.5*, implement `average_weights`. We have provided test code you can use to validate your implementation. This test code will also be useful for the full implementation of Federated Learning in **PART 1.3**.
2. In *Code Cell 1.5*, implement the `get_devices_for_round` function. Try multiple `device_pct` settings to ensure that it is working properly.

(Solution)

1. [See Colab.]
2. [See Colab.]

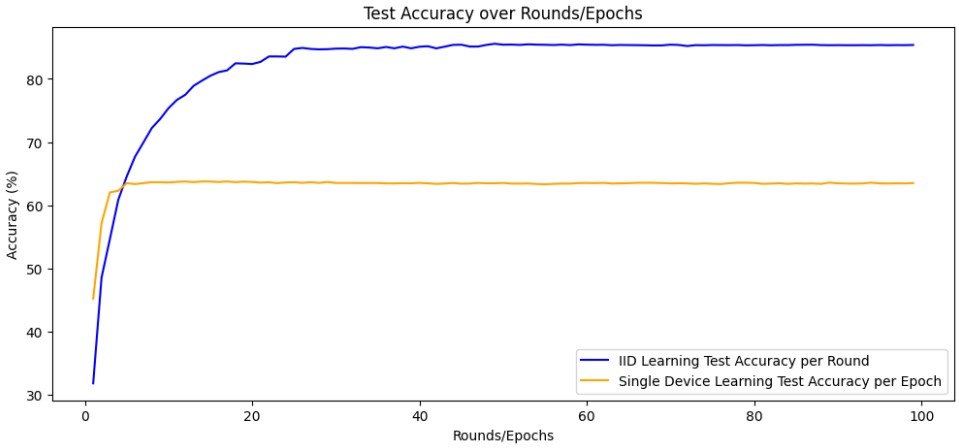
Part 1.3 (10 points)

1. In *Code Cell 1.6*, train a global model via federated learning. Much of the code has been given to you, but you will need to fill in the parts using calls to the functions you wrote above.
2. Graph the accuracy of the global model over 100 rounds. Discuss the accuracy difference between the global model trained here and the individual local model you trained in **PART 1.1**.

(Solution)

1. [See Colab.]

2. Because the global model has access to the entire dataset but is still split across multiple devices, we see a lower final test accuracy compared to the baseline ConvNet model, but there is a $\sim 20\%$ accuracy increase as the global model is able to combine the updates from all of the devices into a larger global model, allowing for individualized training on the entire dataset compared to the individual local model which only sees $\sim 10\%$ of the dataset and thus cannot generalize across the dataset like the global model can.


Test Accuracy over Rounds/Epochs
Accuracy (%)
Rounds/Epochs
IID Learning Test Accuracy per Round
Single Device Learning Test Accuracy per Epoch
30
40
50
60
70
80
0
20
40
60
80
100

Part 2.1 (10 points)

1. In *Code Cell 2.1*, implement the `noniid_group_sampler` function to generate **non-IID** samples from the CIFAR-10 training set. As input, the function should take the dataset and number of training samples each device should be assigned. As in `iid_sampler` you implemented previously, the function should return a `dict` where each key is a device ID number and each value is a `set` of the indices of training samples in the dataset assigned to that device. You may want to have the function return other data as well, depending on how you implement functions in later parts of the assignment. We have provided the mapping that indicates which classes are mapped to each group. Within a given group, you should sample the data in an IID fashion.

(Solution)

1. [See Colab.]

Part 2.2 (5 points)

1. In *Code Cell 2.2*, implement the `get_devices_for_round_GROUP` function to generate a list of devices that will participate in each round of federated learning. The function should, at minimum, take as input 1) the list of all devices, and 2) how many devices from each group should participate in a given round. It should return a list of devices that will participate in a given round. You may want to add additional parameters to the function depending on your implementation strategy.

(Solution)

1. [See Colab.]

Part 2.3 (5 points)

1. In *Code Cell 2.3*, implement the `cifar_noniid_group_test` function to create a test dataset for each group. It should take the full CIFAR-10 test dataset as input, and return a dictionary where each key is a group ID, and each value is a `set` of the indexes for all test samples for that group.
2. In *Code Cell 2.3*, implement the `test_group` function to output the per-group classification accuracy of the global model.

(Solution)

1. [See Colab.]
2. [See Colab.]

Part 2.4 (10 points)

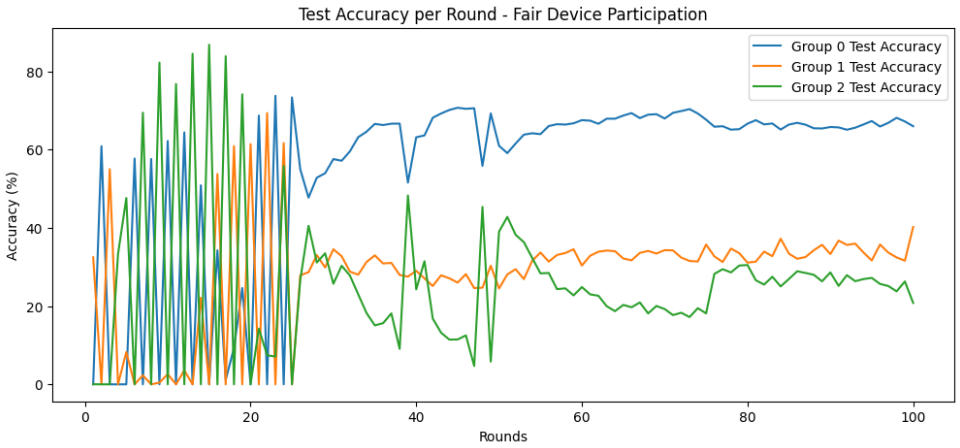
1. In *Code Cell 2.4*, train a global model via federated learning for the group-based non-IID setting. Much of the code has been given to you, but you will need to fill in the parts using calls to the group-based, non-IID functions you wrote above. You will likely be able to re-use parts of the code you wrote in **Part 1.3**.
2. Graph the per-group test accuracy over 100 rounds in the **Fair Device Participation** scenario. Each group should have its own line in the graph.
3. Graph the per-group test accuracy over 100 rounds in the **Unfair Device Participation** scenario. Each group should have its own line in the graph.

4. Describe the differences you see between the two scenarios. How can you explain what you are seeing?

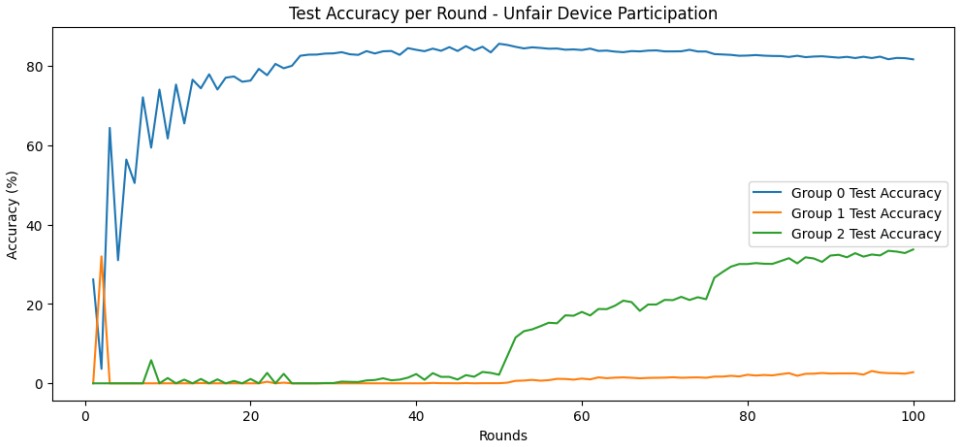
(Solution)

1. [See Colab.]

2. For fair device participation, we obtain the following graph. Note that because we're using the naive averaging method to update the weights on every device, we initially see a large variation in accuracies for the first ~ 25 rounds as the model struggles to find a solution that fits all label classes, but after this training time the model has converged on a set of solutions for each of the label classes and is able to somewhat accurately fit them.



3. For unfair device participation, we obtain the following graph. Note that again as a result of the averaging policy that the Group 0 test accuracy is high while Groups 1 and 2 have lower accuracy, since Group 0, as a result of having more devices, has more of an impact on how the weights are updated than Groups 1 and 2. This results in Groups 1 and 2 having far lower accuracy that Group 0.



4. We see that the unfair device participation policy is able to achieve much higher accuracies for groups 0 and 1 ($\sim 80\%$ and $\sim 40\%$ respectively), but is almost unable to fit group 2, while the fair device participation policy has a large amount of variation for the first 25 epochs but later converges on a solution that retains $\sim 25 - 40\%$ accuracy for groups 1 and 2 while group 0 only has a $\sim 60\%$ accuracy. This is because unfair participation while using averaging to calculate weight updates allows for fast optimization of the dominant group at the expense of minority groups, while fair participation treats each of the groups equally so any differences in accuracy are purely due to the models' ability to converge on solutions for each of the labels.

Part 3.1 (5 points)

1. In *Code Cell 3.1*, implement a function that converts full-precision numbers in the range $[0, 1]$ into n-bit fixed-point numbers. If your implementation is correct, it should return *'Output of Quantization Matches!'*.

   (Solution) See notebook cell 3.1. Test case passed.

Part 3.2 (10 points)

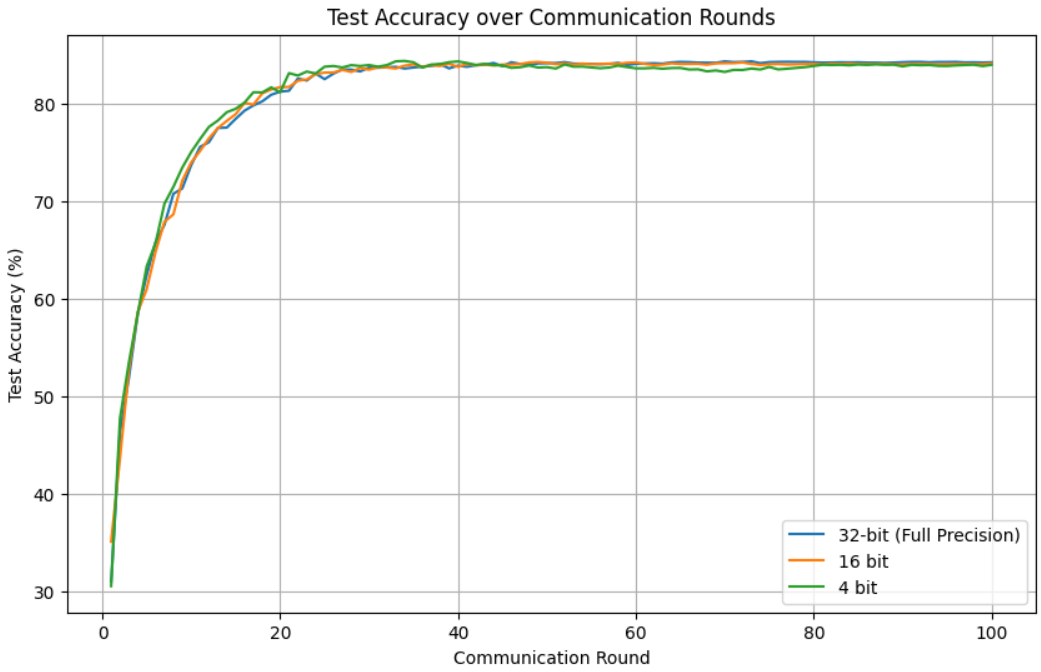
1. In *Code Cell 3.2*, implement `dorefa_g(w, nbit, adaptive_scale=None)` using the **stochastic quantization function** shown above. Again, if your implementation is correct, it should return *'Output of Quantization Matches!'*.

   (Solution) See notebook cell 3.2. Test case passed

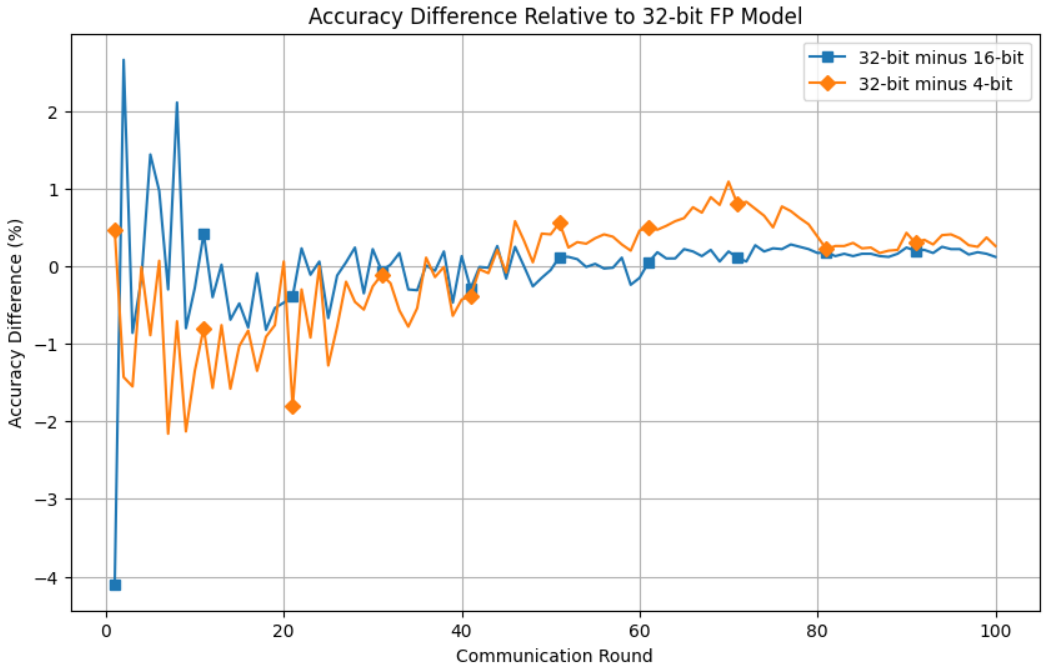
Part 3.3 (10 points)

1. In *Code Cell 3.3*, run federated learning with the following two quantization settings (bit widths): `nbit=16` and `nbit=4`. Graph the accuracy of the global models over 100 rounds for the different bit widths: 32-bit (the full-precision baseline you ran previously), 16-bit, and 4-bit.

   (Solution) We graph the accuracy of the global models over 100 rounds of communication for the different bit widths: FP32, 16-bit, and 4-bit.



For clarity, we also include the accuracy difference relative to the non-quantized, FP32 model:



2. Discuss the accuracy difference between the global models across the three different bit width settings: 32-bit, 16-bit, and 4-bit. (100 words maximum)

(Solution) The global models trained with 32-bit, 16-bit, and 4-bit quantization seem to exhibit very similar accuracy over 100 communication rounds. In the later communication rounds, the 16-bit model's performance is nearly indistinguishable from the full-precision 32-bit model, indicating minimal impact from quantization at this bit width. The decline in accuracy is more noticeable in the 4-bit model between rounds 40-80 where the accuracy difference is positive but still marginal between 0 and 1% (see second graph).

Intuition: our intuition for the results is that quantization was applied only during communication, not during local training. Local updates were performed in full precision, allowing the models to recover from any quantization-induced errors. The averaging process in federated learning further mitigates quantization errors across devices. Therefore, quantizing model updates effectively reduces communication overhead without signficantly compromising model performance as suggested by our results.

Part 4.1 (15 points)

1. Implement three defense strategies (i.e., noise injection, pruning, and quantization) defense_method in *Code Cell 4.1.*
2. Run the 4 experiments (i.e., Baseline, Noise, Pruning, and Quantization). For each experiment, you should report: (1) the final inversion result measured by $\|\text{image} - \text{recovered\_image}\|_2$ and (2) the optimization process of inversion using *Code Cell 4.2.*
3. Compare and discuss the effectiveness of the 3 different defense strategies. (100 words maximum)

(Solution)

1. See Colab.
2. **Baseline:** Final inversion result (L2 norm) is: 0.0694517195224762

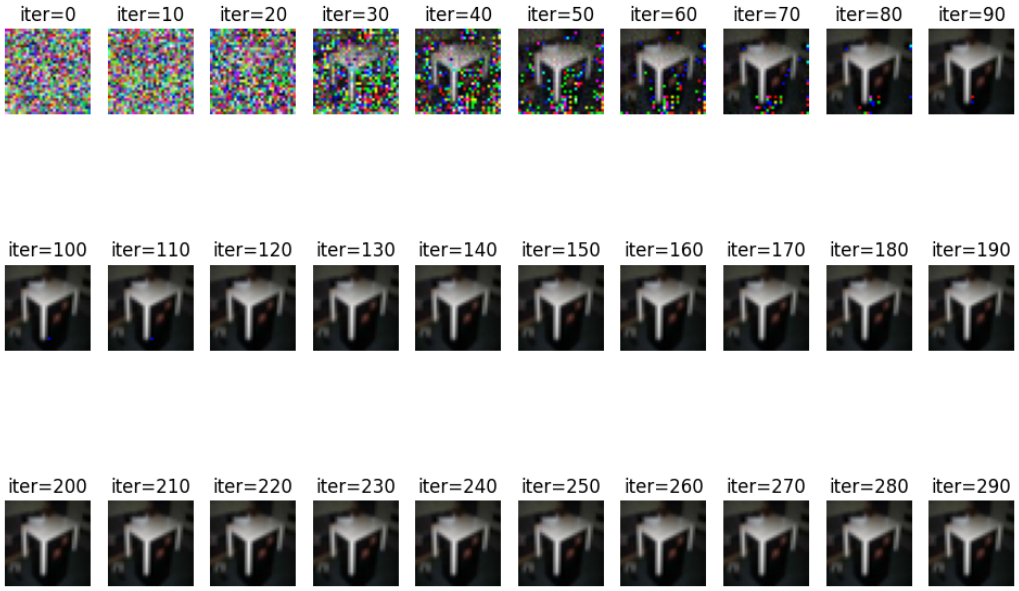


Figure 1: No Defense

Pruning: Final inversion result (L2 norm) is: 9.196956634521484

Figure 2: Pruning Defense

Noise: Final inversion result (L2 norm) is: 326.1081237792969

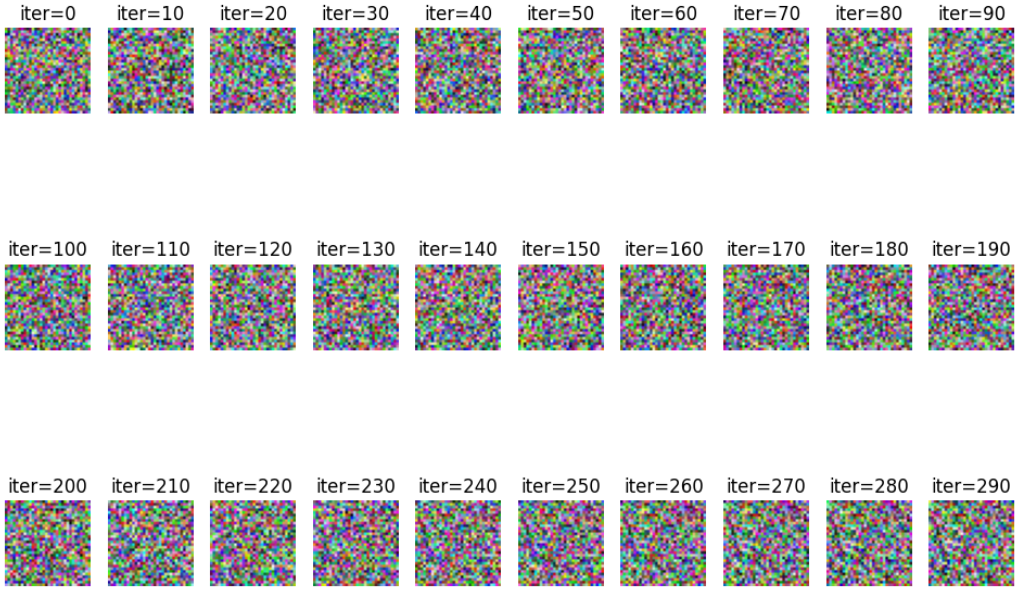


Figure 3: Noise Defense

Quantization: Final inversion result (L2 norm) is: 236.5286102294922

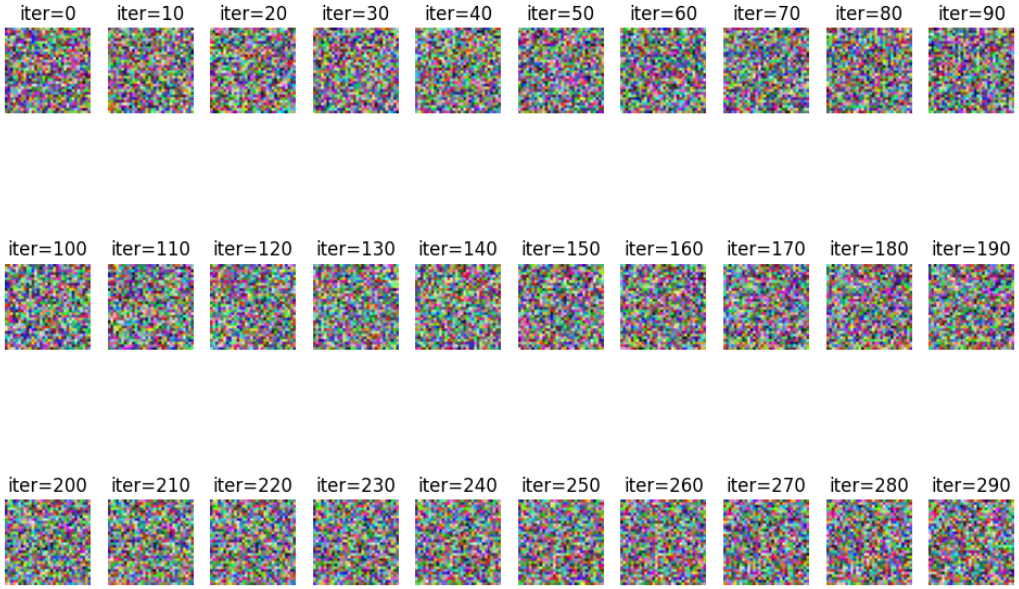


Figure 4: Quantization Defense

3. Based on the L2 norm final inversion results, noise is the most effective defense, with quantization being the second most effective, and pruning being the least effective. This is verified by looking at the optimization process of each– the noise defense never converges to a recognizable final inversion image, the quantization defense also never converges to a recognizable final inversion image, and the pruning defense converges to a final image that is bears some resemblance to the data. This makes sense intuitively – with quantization and noise, we are altering *all* the data in the gradient (drastically reducing precision by sorting all values into 16 bins, or by injecting random noise). On the other hand, pruning simply zeroes out the least significant values in the gradient which intuitively hold the least information, resulting in inversion still being effective.

Part 5.1 (15 points)

1. Using a regularization set, with `with_prior_preservation` set to True, helps prevent overfitting. How does this help the fine-tuned Stable Diffusion model generate both generic and personalized images?

(Solution)

1. By introducing a prior preservation loss into our fine-tuning via a regularization set (containing the model's own generated images of our target class), we help the model retain what it has already learned in pretraining and discourage overfitting (i.e., memorizing the fine-tuning set too well). Hence, prior preservation helps ensure the model retrains its ability to generate generic images, without being overly biased towards our fine-tuning dataset.

Part 5.2 (15 points)

Please attach 4-5 generated images using another set of personal assets in Overleaf. You will need to fine-tune the Stable Diffusion model on this new set of personal assets (not the personal dog from the provided URLs). For instructions on downloading personal images/assets, please refer to the "Where Can I Find and Download Public Personal Images?" section.

(Solution) We train the model on a set of shiny sneaker images from the DreamBooth database. We obtain the following images without our special token:

And the following images with our special token:

While the generic prompt is still able to generate some appropriate images of shiny sneakers (and, in particular, more diverse images), our prompt with the special token is more consistent in generating shiny sneakers matching the style of the dataset used for fine-tuning.

Part 6.1 (15 points)

1. Implement the hooks in *Code Cell 6.9* to modify the behavior of the cross-attention layers.

(Solution)
1. [See Colab.]

Part 6.2 (15 points)

1. Implement the hooks in *Code Cell 6.15* to modify the behavior of the cross-attention layers.

(Solution)
1. [See Colab.]