ATML: PA3 - Model Compression

Muhammad Hamza Habib [1] **Abdul Samad** [2] **Rumaan Mujtaba** [3]

Abstract

We evaluated three model optimization techniques: pruning, quantization, and knowledge distillation (KD). First, we applied structured and unstructured pruning to a VGG-11 on CIFAR-10. Structured pruning gave the best size (109MB) and latency (1.48ms), while unstructured pruning was more robust to accuracy drops but increased file size (644MB) due to masks. Second, we evaluated quantization on a VGG-11 (CIFAR-100). We found that INT4 Post-Training Quantization (PTQ) fails (35.10% accuracy), but Quantization-Aware Training (QAT) recovers it to 66.85%. An adaptive mixed-precision PTQ strategy yielded the best results, surpassing the baseline at 71.65%. Third, we used KD to distill a VGG-16 to a VGG-11 (CIFAR-100). Contrastive Representation Distillation (CRD) and Decoupled KD (DKD) were most effective, with CRD achieving 0.6565 accuracy and successfully transferring color invariance.

1. Introduction

Deep learning models are often too large, slow, and power-hungry for real-world applications. This report evaluates three main techniques to optimize and compress models, using VGG architectures on CIFAR datasets as our testbed.

First, we investigate **pruning**, comparing unstructured (removing individual weights) and structured (removing entire channels) methods to see the trade-offs in model size, speed, and accuracy.

Second, we evaluate **quantization**, testing different bit-widths (like FP16, INT8, and INT4). We compare the performance of simple Post-Training Quantization (PTQ) against the more complex Quantization-Aware Training (QAT).

Finally, we explore **knowledge distillation (KD)**, where we train a small VGG-11 "student" model to mimic a larger VGG-16 "teacher" model. We test several KD methods, like CRD and DKD, to see which one transfers knowledge most effectively.

The codebase for these experiemts conducted can be found at GitHub

2. Task 1

3. Methodology

We start the unstructured pruning task by first loading the **VGG_11** architecture and creating dataloaders for our dataset, **CIFAR-10**. Next, we first train the baseline model for *epochs=15* and *learning rate=0.001*. Using Stochastic Gradient Descent optimizer with *momentum=0.9* and *weight decay=5e-4*, the model is trained on our **CIFAR-10** trainloader and the model state with the best validation accuracy is saved. Then, we move on to the unstructured pruning implementation. We start by loading the saved baseline **VGG_11** model and then retrieve the list of parameter names suitable for unstructured pruning by iterating over the named parameters of this **VGG_11** model. Next, we initialize a binary mask tensor for each of these prunable parameters, and save a histogram visualization of each of these parameters/layer weights, comparing all weights and active weights (after applying mask) distribution. These are saved with the "before pruning" tag.

Then, to run sensitivity analysis, we first define two sparsity levels, 50% and 70%. Next, we, first, iterate over each prunable name/layer for each sparsity level, load our saved baseline model, and initialize the binary mask tensor like previously. Then, we prune each layers weights by, first, identifying active masks and pruning (setting their mask value to 0) them based on magnitude in ascending order until the sparsity level is reached. The updated mask is then applied to the model weights in-place, and this updated model is then evaluated on the **CIFAR-10** testloader. The evaluation accuracy along with sparsity, and per-layer sensitivity plot of sparsity against evaluation accuracy is saved in a json file.

Then, we apply a different approach of magnitude-based pruning and also introduce finetuning to study the impact. This time, we initialize the saved baseline model once, prune weights in all the prunable layers first and then apply the updated mask and evaluate the test accuracy. Then, for finetuning, we initialize Cross Entropy Loss function and Stochastic Gradient Descent with *learning* *rate=0.01* and similar *momentum* and *weight_decay* as for the baseline training. After finetuning for $epochs = 10$, the global sparsity of the model is computed and its state is saved.

Now, we move on to the structured pruning task, where we define our sparsity levels same as in unstructured pruning. For this task also, we iterate over each sparsity level for each convolution layer of our model, create a dictionary of indices of output channels to keep, and rebuild our pruned model. It does this by mapping layers using the dictionary of indices and rebuilding the network by slicing weights, rebuilding BatchNorm layers - to match the new channel count, and classifier. Then, we evaluate this pruned rebuilt network without finetuning, and store the accuracy, sparsity, and per-layer plot in a json file (similar to how we did in unstructured pruning). Then, we test how finetuning 8 least sensitive layers for **epochs=2** improve performance of a pruned network with each layer having sparsity level of 70%. The least sensitive layers represent the layers with the highest accuracy on the previous no fine-tuning structured pruning.

Then, we implement a sensitivity-aware approach for structured pruning. Based on the accuracy results of the no-finetuning structured pruning implementation, the convolution layers are sorted with the least sensitive first (descending order). For each layer the, we greedily allocate channels to be removed until the global parameter reduction of 70% is met, while ensuring that each layer cannot be pruned more than 90%. If the global reduction is still not met, we perform an incremental one-channel-at-a-time pass over the layers, still based on sensitivity score and return per-layer sparsities.

Next, we iterate over the list of our 8 least sensitive layers, and decide the channels to keep, based on our per-layer sparsities calculated previously. For other layers, all the channels are kept. Based on this, our network is rebuilt and a quick evaluation is performed. Then we finetune using the same loss function and optimizer and its parameters as in the previous finetune tasks for *epochs=10* with early stopping. The final model state along with sparsity information is saved.

Then, using the final state of our three models, i.e., baseline, structured pruning, unstructured pruning, we create a grad-cam heatmap using each model for 3 randomly chosen images in our dataset. This is overlaid on the original image and saved. Then, to calculate the average inference time for an image for each model, we use image at index 0, set the model to evaluation mode, run 20 initial passes to prime the GPU, and then run the forward pass for 200 runs. The collected run times are converted to

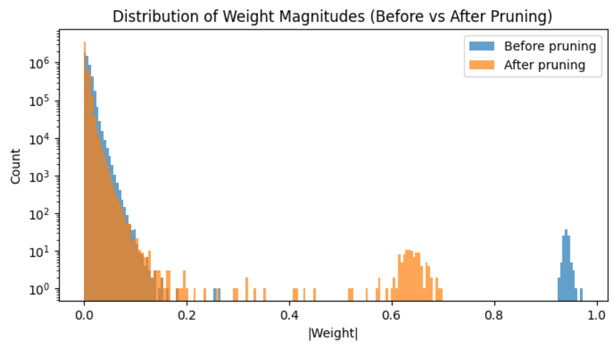

Figure 1. Weight Distribution Before and After Unstructured Pruning

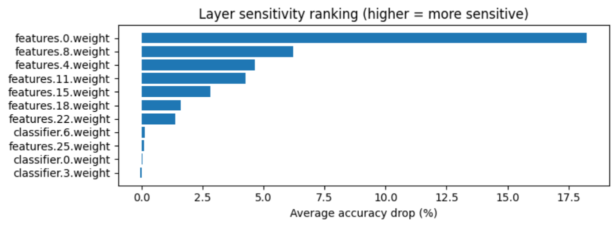

Figure 2. Accuracy Drop For Layers After Unstructured Pruning

milliseconds and mean and standard deviation are saved. At the same time, we also compute the storage on-device (in bytes) for each of our three saved models. Finally, to show severity of pruning on model performance, we first save no-finetuned model checkpoints at our sparsity list: $[0.0, 0.3, 0.5, 0.7]$, alongside the accuracy on testset for each sparsity level. We do this for both unstructured and structured pruning and then make the plots for accuracy against target sparsity for each.

4. Results

The distribution of weight tensors before and after we apply unstructured pruning is provided in **Figure 1**

The sensitivity analysis ranking the accuracy drop for the layers after unstructured pruning is shown in **Figure 2**

Similarly, the weight tensors distribution and accuracy drop for layers after structured pruning is shown in **Figure 3** and **Figure 4**

The Grad-CAM analysis of the three models (original, unstructured-pruned, structured-pruned) for a random image, to highlight their distinctiveness, is given in **Figure 5**, **Figure 6**, **Figure 7**

The mean and standard deviation of the inference times for

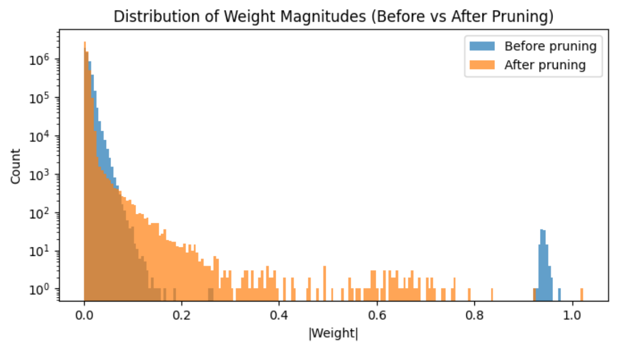

Figure 3. Weight Distribution Before and After Structured Pruning

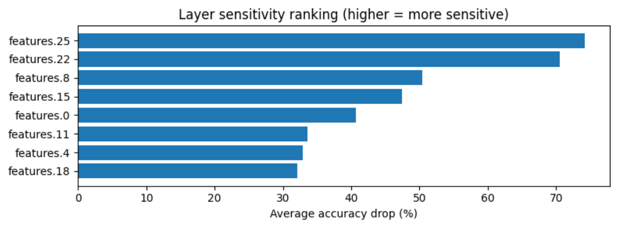

Figure 4. Accuracy Drop For Layers After Unstructured Pruning

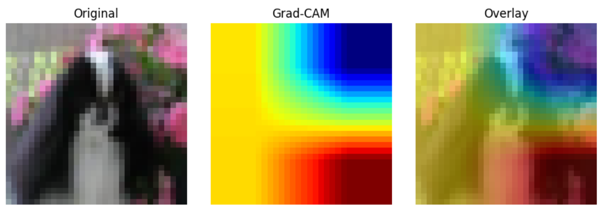

Figure 5. Grad-CAM Heatmap of the Original Model

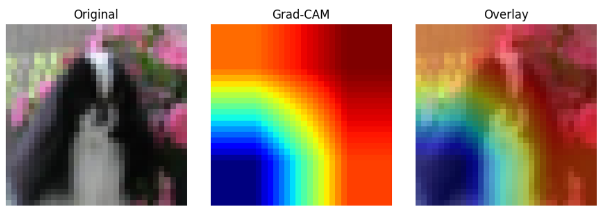

Figure 6. Grad-CAM Heatmap of the Structured-Pruned Model

Model	Mean	Standard Deviation
Original	4.13	4.81
Unstructured-Pruned	3.78	2.90
Structured-Pruned	1.48	0.06

Table 1. Mean and Standard Deviation of Inference Time of Each Model

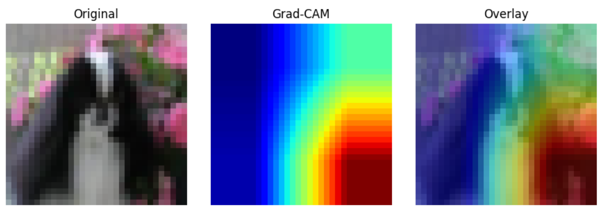

Figure 7. Grad-CAM Heatmap of the Unstructured-Pruned Model

a random image for each of the three models is summarized in **Table 1**. In addition to this, the storage on device for each model is shown in **Table 2**

Model	Storage on-device
Original	515
Unstructured-Pruned	644
Structured-Pruned	109

Table 2. Storage On-Device in Mega Bytes For Each Model

To show how the severity of pruning affects performance, the plot of test accuracy against target sparsity for unstructured pruning is shown in **Figure 8**, while for structured pruning is shown in **Figure 9**

5. Discussion

The sparsity ratios we set for each layer for both structured and unstructured pruning is determined dynamically. That is we calculate the sparsity ratio that should be applied to each layer through a greedy method. For unstructured pruning, we prune layer's weights using their magnitude through the binary mask until the global sparsity level is reached. This ensures that the important weights remain active while the less important ones are pruned until the global sparsity of 70% is reached.

Then, for structured pruning, we also use a greedy, parameter-count method while ensuring that we don't prune a layer completely and destabilize the architecture - hence the 90% prune cap on each layer. By using validation accuracy as the indicator of a layer's sensitivity for this greedy approach, it allows us to ensure that sensitive

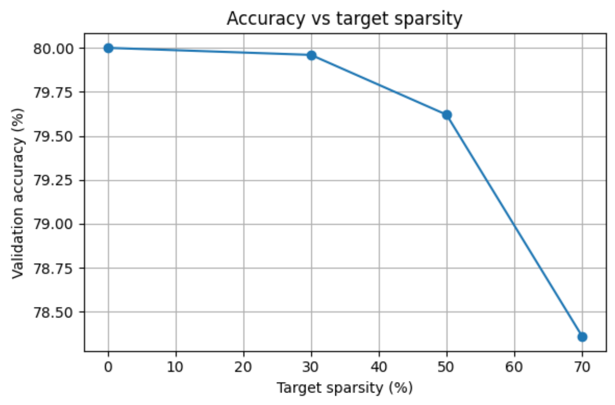

Figure 8. Test Accuracy vs. Target Sparsity For Unstructured Pruning

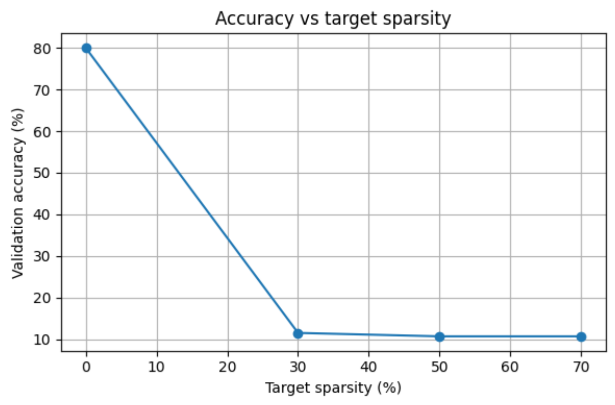

Figure 9. Test Accuracy vs. Target Sparsity For Structured Pruning

channels contributing to model's performance remain active. The goal of 70% global sparsity is mostly ensured by this, but still, we perform an incremental one-channel-at-a-time pass over the layers at the end to reach our 70% global sparsity ratio. For storage on-device comparison between the three models, structured pruning achieved the greatest reduction because it permanently removes channels by rebuilding the architecture. On the other hand, the increase in parameters for unstructured pruning is explained by how it only introduces zero-values weights which are often saved as full-precision numbers. This increases file size due to storing mask.

For comparison of Grad-CAM heatmap of the three models, the structured-pruned model focuses more on body of the animal in the image, highlighting that it's focus is on the important features, since it removes entire neurons. On the other hand, the unstructured-pruned model's focus is less on important features and its attention is spread more overall because it zeroes out random individual weights. When comparing test accuracy of structured pruning against different sparsity levels, the sudden drop in accuracy indicates that removing entire channels significantly damages the model's core ability to compute feature maps. Since there is no fine-tuning as well, the model cannot compensate. Whereas, for unstructured pruning, removing the redundant individual weights preserves the core ability, since the network is already over-parameterized, and doesn't cause much drastic drop in accuracy even at 70% global sparsity.

Finally, we move to the qualitative analysis of unstructured and structured pruning. Structured pruning removes entire filters or channels and results in smaller dense models that can execute quickly on standard hardware. Unstructured pruning removes individual weights, offering higher compression but limited speedups unless there is specific hardware for this. While unstructured pruning maintains accuracy better at high sparsity, its irregularity keeps it from real-time efficiency. Structured pruning is more hardware-friendly but can harm accuracy if overused. Thus, structured pruning is suitable when latency and efficiency matter most, e.g., deployment, while unstructured pruning suits research and memory reduction.

6. Task 2

7. Methodology

The experiment was carried out using a pretrained VGG-11 backbone from PyTorch library which was trained on ImageNet weights, classifying images from CIFAR-100 dataset. The dataset images were resized to 224x224 size, and normalized using ImageNet mean and std. First, the VGG-11 backbone was fine tuned for 10 epochs on the CIFAR-100 dataset achieving a training accuracy of 73%. This sets the baseline for model accuracy on float32 (default one) data type. While performing all these experiments, 2xT4 GPUs were used in parallel. Since, T4 GPUs don't support int4 calculations, a few tricks were employed to go perform int4 experiment.

Float16 and Bfloat16: The baseline model was deep copyed and quantized to lower bit width, float16 and bfloat16, significantly reducing its size. Torchao version 0.13.0 was used to carry out the quantization of VGG11.

INT4 and INT8: The same baseline fine-tuned on CIFAR-100 was used for this experiment. For INT4, and INT8 quantizaton, INT8 weights only and INT8 dynamic activation with INT4 weights only quantization was carried out. The model was then evaluated on the same test dataset as before.

QAT: This experiemtn aimed to evaluate the effectiveness of QAT for minimizing model performance loss. We conducted QAT experiment on four different bit-widths, FP16, BF16, INT8, and INT4. The experiment was performed by replacing all the Conv2d and Linear layers with custom quantized Conv2d and Quantized Linear modules. There modules were initialized with the weights of original baseline layers after the quantization. Each Quantized model went through 2 epochs of fine tuning after that. The learning rate for fine tuning INT quantized models was 0.0001 and 0.001 for FP quantized models. The custom modules' forward pass defaults to FP32 operations in eval() modee, so it measures baseline accuracy again.

Mixed Precision: We also evaluated PTQ strategies. Uniform (all layers to same precision), Simple (first and last layer at FP16 and others INT8), and adaptive (FP16 for high sensitivity layer, INT8 for medium sensitive layers, and INT4 for low sensitivity layers).

8. Results

PTQ: A graph comparing sizes of all the models after post training quantization is given in Figure 10

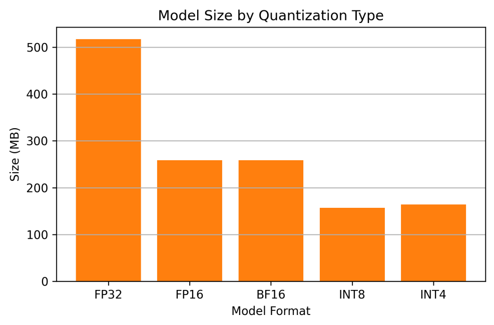

Figure 10. Size of VGG-11 on different bit width

We can see that the size of the model falls as we decrease bit width. Morover, in case of INT4, the model was quantized using dynamic int8 activation with int4 weights, therefore, the model size didn't fall to half and is comparable. This was necessary as T4 GPUs can't work on INT4 calculations, and time constraints of using CPU.

Moreover, the accuracy graph of these models is also given in Figure 11

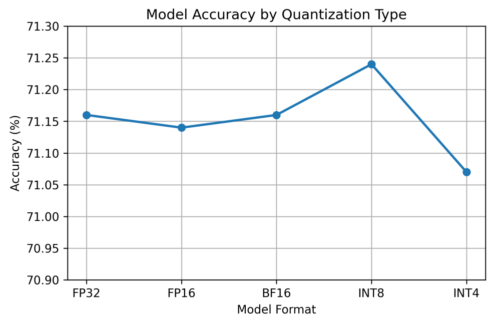

Figure 11. Accuracy of VGG-11 on different bit widths

We can see that that accuracy drops by a little margin going from FP32 to FP16, and falls drastically when going from INT8 to INT4.

QAT: The results from QAT experiments give insights into the performance-comparison trade-off, and also how QAT performs better than PQT when it comes to model performance. During training, it was observed that the models recovered their accuracy in the two epochs they were fine-tuned in. The accuracy comparison of these models is given in the Table 3. Moreover, you can also observe that the size for INT4 and INT8 is the same as the base table, that's because we only simulated the INT4 and INT8 quantization given the hardware limitations.

We can see that the accuracy drops more significantly to 68.15% for INT4 compared to INT 8 which has a good compression ratio. Moreover, it can also be seen that bf16

Table 3. QAT results after a 2-epoch fine-tuning. The `int8` and `int4` sizes reflect the simulation artifact of measuring the in-memory FP32 master weights, not the theoretical compressed size.

Bit-Width	Initial (%)	Final (%)	Recovery (%)	Size (MB)	Comp.
fp16	71.25	71.65	+0.40	253.60	2.00x
bf16	71.55	71.72	+0.17	253.60	2.00x
int8	68.42	71.15	+2.73	**492.77**	**1.00x**
int4	35.10	66.85	+31.75	**492.77**	**1.00x**

performs better than the baseline. The justification for this would be that bf16 recovered more accuracy compared to its competitors because of large range of values it can have, and that baseline model wasn't converged so precision wasn't the bottleneck for model performance.

Mixed Percision: We next evaluated mixed precision PTQ strategies to compress models while maintaining the maximum accuracy. The results are show in table 4. The most significant observation is that adaptive strategy maintained the best accuracy given that it took the importance of layers into given while quantizing them.

Table 4. Mixed-precision PTQ results. The `uniform` (all-`int8`) strategy is the baseline. The FP32 model size is 492.76 MB. The `adaptive` strategy achieves the highest compression with no accuracy loss.

Strategy	Accuracy (%)	Size (MB)	Latency (ms)
uniform	70.25	123.19	1.51
simple	70.3	123.58	1.52
adaptive	**71.65**	**113.29**	**1.53**

9. Discussion

PQT: PQT works by applying quantizion after the training of the model, so the stored weights lose some of their precision. This helps to reduce the model size without comprising too much on the model performance. In our results, we can see that accuracy falls by a margin when we quantize to FP16 from FP32, BF16 has more accuracy compared to FP16, that might be because BF16 uses more range while compromising on precision in its decimal part. One worth mentioning thing is that the model didn't converge properly so its weights wouldn't be precise enough, otherwise the accuracy fluctuation would have been more significant. This is clearly visible in 11 where the accuracy drop from INT8 to INT4 is drastic. This confirms that for aggressive, low-bit quantization, a simple post-training approach is not enough.

QAT: The downside of PQT is that the models parameters. lose their precision, QAT aims to fix that by performing quantization during training so the model weights are updated in the presence of quantization and imprecision. In our fine tuning, QAT brought back INT4 model from 31.75% accuracy to 66.85% which shows a massive improvement. One really important result is that BF16 performed better than the FP32 baseline (71.72% vs 71.15%). The justification for this, as we mentioned, is that our baseline model wasn't fully converged. The extra 2 epochs of fine-tuning combined with BF16's large range, actually pushed the model to a better performance. Moreover, the 1.00x compression ratio for INT8 and INT4 models is because we only simulated the quantization, given the hardware limitations. As mentioned in the methodology, the custom quantized layers keep the FP32 weights for the optimizer to use during the backward pass, so the in size memory doesn't change. The key takeaway is not the size, but that the accuracy can be recovered in a quantized model.

Mixed Precision: The mixed precision experiment shows the best path for optimization. The uniform and simple one both saw a drop in performacne, but the adaptive model achieved the highest accuracy of all the rest. It means that hte data driven approach seccessfully identified the most sensitive layers that could be quantized while saving the maximum performacne.

10. Task 3

11. Methodology

The CIFAR- 10 dataset is used for this task, it is preprocessed and normalized to be used as validation dataset also.Both the models are VGG based, with student using VGG-11 and teacher using VGG -16. After adaptive average pooling, the final classification layer of these models is adapted to output 100 CIFAR - 10 classes. The independent student VGG-11 model is trained from scratch on CIFAR-100, the Adam optimizer is used, and cross-entropy loss for 10 epochs. For logit-matching, the student VGG-11 is trained using a custom logit matching loss which uses KL divergence loss with temperature along with cross-entropy loss, the student tries to mimic the softened output logits of the teeacher. For label smoothing , the target labels are softened during training to prevent the model from becoming overconfident using the label smoothing cross entropy loss without directly using teacher for loss calculation. Grad-Cam heatmaps are generated to visualise teacher and all student model's feature localization. The original teacher is fine-tuned to create a colour-invariant teacher, a new student is then trained using Contrastive Representation Distillation and the new teacher. The VGGWithExtractorForCRD architecture with NCE loss, is used for both models and the student is trained without colour jitter to observe colour invariance transfer. To see the importance of size of teacher, a VGG-19 model pretrained on Image - Net is fine tuned on CIFAR-100 to be a large teacher, a VGG-11 student is then trained using this larger teacher and its performance is compare to the VGG-11 student trained on VGG-16 teacher.

12. Results

The Independent Student (SI) had a validation accuracy of 0.4660 with a loss of 1.9581. The fine-tuned Teacher (T) model(a VGG-16 model), had a significantly higher validation accuracy of 0.6869 and a lower loss of 1.1672.The Logit Matching Student (LM-SD) reached an accuracy of 0.5887 and a loss of 1.5559. The Label Smoothing Student (LS-SD) had an accuracy of 0.5399 and a loss of 2.2344. The Decoupled KD Student (DKD-SD) performed the best among these, achieving an accuracy of 0.6080 and a loss of 2.8137.

We further investigated KD methods,the Hint-based Student achieved an accuracy of 0.4669 and a loss of 1.9601, showing only a minimal improvement over the Independent Student.The CRD Student (CRD-SD) demonstrated strong performance with an accuracy of 0.6565 and a loss of 1.2904, closely approaching the Teacher's accuracy.

Table 5. Average KL Divergences ($D_{KL}(Teacher \mid\mid Student)$)

Student Model	KL Divergence
SI vs T	0.3243
LM-SD vs T	0.1086
Hints-SD vs T	0.2546
CRD-SD vs T	0.1938

We used GradCAM to visualize how effectively localization knowledge was transferred from the teacher to the student models.

Table 6. Average GradCAM Cosine Similarities (Teacher vs Student)

Student Model	Cosine Similarity
SI vs T	0.1271
LM-SD vs T	0.0916
Hints-SD vs T	0.0385
CRD-SD vs T	0.1731

The teacher model was fine-tuned by augmenting color jitter to achieve color invariance. The Color-invariant Teacher (VGG-16) achieved a validation accuracy of 0.6655 on a color-jittered dataset. The CRD distillation was then done on the independent student using this color-invariant teacher and regular augmentations. The final evaluation of the CRD Student on a color-jittered validation set showed an accuracy of 0.6630.Suggesting that CRD effectively transferred the color invariance property from the teacher to the student.

13. Discussion

The independent student model set the baseline with its lowest accuracy, the logit matching student performed much better as it was able to better transfer the complex interclass relationship of teacher model, the label smoothing model was able to improve accuracy as a regulizer but performed poorly as compared to direct knowledge transfer methods. The DKD model performed the best and achieved highest accuracy due to its decoupled approach to target and non target class learning.

The CRD student achieved much higher validation accuracy as compared to hint base student , suggesting that CRD which focuses on aligh=ningbfeature representations is more effective in improving student's overall classification, as compared to Hint based which uses intermediated feature transfer.

All KD based methods performed better than the independent student, with the logit matching demonstrating strongest alignment and lowest KL divergence.

The CRD-SD achieved highest cosine similarity with the teacher, showing its effectiveness in aligning student model focus on important features.

14. Conclusion

We evaluated three different ways to optimize models: pruning, quantization, and knowledge distillation.

For **pruning**, we found a clear trade-off. Structured pruning is best for real-world deployment. It gave the fastest inference (1.48ms) and the smallest file size (109MB). Unstructured pruning was better at keeping accuracy at high sparsity levels, but it gave no real speedup and actually made the file size bigger (644MB) because the masks had to be saved.

For **quantization**, we showed that simple Post-Training Quantization (PTQ) is not safe for aggressive, low-bit (INT4) quantization. It caused accuracy to collapse. Quantization-Aware Training (QAT) is required to fix this, and it worked well, recovering the INT4 model's performance. The adaptive mixed-precision strategy was the clear winner, giving the highest accuracy (71.65%) with good compression.

For **knowledge distillation**, we confirmed that advanced methods like CRD and DKD are much better than training a student model by itself. CRD was especially good, not just for accuracy, but also for transferring the teacher's "focus" (shown by GradCAM) and special properties like color invariance.

In summary, the best method depends on the goal. Use structured pruning for speed, adaptive quantization for the best accuracy/size balance, and advanced KD like CRD to transfer complex knowledge.