LUMS
A Not-for-Profit University

School of Science and Engineering

AI624 AI on Edge Devices

ASSIGNMENT 1A - Pruning

Due Date: 11:59 PM, Monday, October 13, 2025.
Format: 4 tasks worth a total of 100 marks
Instructions:

- This is the first part (out of two) of this assignment. The second part will be released in a few days.
- Collaboration with peers is allowed, but copying another student's solution is strictly prohibited. This is not a group assignment—each student must submit their own work.
- Submit all scripts/notebooks along with visible results, as well as a comprehensive markdown file summarizing the results and analyses for each task (and subtask). Grading will be based on both the markdown and the code.
- We encourage you to maintain a well-structured codebase for each assignment rather than placing all code in a single notebook. For our sanity (and yours), include a brief outline of the codebase layout at the end of your Markdown file.
- Submit your work on LMS by the deadline.
- **Start early.** This is a lengthy assignment, and we do not expect you to (nor should you expect yourself) to complete it in just the last few days. Ample time has been provided.
- **Late submission policy:** You can submit with a 10% penalty per day for 9 days after the due date. No submissions will be accepted after that.
- If you have any concerns, feel free to contact the instructor or teaching assistants. We also encourage you to be active on the Slack channel for assignments by both asking and answering questions to foster collaborative learning.

- You are among the most competent individuals in the country. Do not let plagiarism undermine your learning. Any instance of plagiarism will be dealt with in accordance with the university's rules and regulations.
- You must acknowledge any use of generative AI tools. Include the following statement where applicable:

  I have used [insert Tool Name] to [write, generate, plot, or compute; explain specific use of generative AI] [number of times].
- You can optionally share a link to your chat with the relevant chatbot for further clarity, if you deem it necessary.

---

1 Task 0: Load and Profile the Base Models

In this task, you will load baseline pretrained models and datasets, apply the correct preprocessing, and then profile the models to establish reference values. These values will serve as the baseline for the pruning tasks in the following sections.

(a) **Load Pretrained Models**
Use the VGG16-BN models pretrained on CIFAR-10 and CIFAR-100. The models are provided by chenyaofo/pytorch-cifar-models.

(b) **Load CIFAR Datasets**
Download and load the CIFAR-10 and CIFAR-100 datasets. Apply transformations consistent with the chenyaofo/image-classification-codebase to ensure fair comparisons.

(c) **Build Dataloaders**
Create training and test dataloaders for CIFAR-10 and CIFAR-100. Ensure that batch sizes are chosen to balance throughput and memory constraints.

(d) **Profile the Models**
Using the loaded VGG16-BN models and CIFAR dataloaders, perform the following:

1. **Memory Usage and Latency:** Report peak GPU memory, average GPU memory (averaged across operations), and end-to-end inference latency (ms) per batch using `torch.profiler`.
2. **Energy Footprint:** Record energy usage (CPU and GPU domains) using pyJoules.
3. **Model Size:** Compute serialized model size in MB.
4. **MACs:** Record the number of MACs using `torchprofile`.

You are expected to average your results across a few batches for (1) and (2). Choose a reasonable batch size in Part (c).

(e) **Verify Baseline Accuracy**
Evaluate Top-1 and Top-5 accuracy on the test sets and the train sets. Confirm that these are close to the values reported in the chenyaofo/pytorch-cifar-models repository.

Deliverables:

- Print profiling results, including:
  - Model size (MB)
  - Peak memory (MB)
  - Average memory (MB)
  - Latency (ms)
  - Energy consumption (mJ)
  - Top-1 / Top-5 accuracy (%)
  - MACs
- Present a table of these results in an accompanying markdown file (`README.md`) in your submission, in addition to printing them in your notebook or logging your script output to a file.

For each of the following tasks, set a 70-90% pruning ratio, whatever you can achieve while maintaining within 5-15% of accuracy drop (or better).

Note that some sub-tasks of Part 2 (Quantization) of this assignment will involve utilizing the pruned models from this part. You should, accordingly, save all your pruned models.

2 Task 1: Unstructured Pruning

(a) **Sensitivity Analysis and Layerwise Sparsity**

1. Perform sensitivity analysis across layers by setting different sparsity levels for each layer (run for {0, 10, 20, 50, 70, 80, 90}% sparsity) and measuring accuracy drops. This means sweeping fine-grained absolute magnitude-based pruning for each layer. Plot the graph showing the sensitivity curves. You can look up resources from this course from Han Lab for better clarity on doing this.
2. Decide a sparsity ratio for each layer based on your sensitivity analysis and compute the overall sparsity ratio. Remember, setting a higher sparsity ratio for layers with a larger number of parameters will lead to a higher overall sparsity ratio, but might impact accuracy.
3. Fine-tune the pruned models on the train set.
4. Save pruning masks and force the corresponding weights to zero.
5. Use `torch.sparse` to convert to the COO format.
6. Verify that the correct weights are zeroed by comparing your mask and the coordinates (indices) in the COO format.
7. Run inference and profile by replacing the dense weight tensors with the now sparse tensors. Note that you will have to modify the forward method of the model to convert the sparse tensors back to dense during inference. This is because `torch.sparse` does not support sparse convolution with unstructured sparsity. It does, however, support sparse and dense matrix multiplication. You can speed up the inference in the linear layers accordingly.

(b) **Saliency-Based Iterative Pruning**

1. Initialize the VGG16-BN model (for both CIFAR10 and CIFAR100) with random weights and train to approximately 20% accuracy.
2. Use the GrasP saliency criteria to prune to 50% of the target sparsity. A great resource on computing Hessian-Vector products can be found here. Keep the following points in mind while pruning based on the saliency criteria:
   - Keep a reasonable number of samples from the training set in your calibration dataset for computing the Hessian gradient products.
   - If your dataset spans multiple batches, you might find it useful to accumulate (a sum or average) of the gradients over the batches.
   - You will find `torch.autograd.grad` a useful function.
   - You should consult Algorithm 1 and Algorithm 2 of the paper. Note that, unlike in the paper, you will be pruning in stages, rather than a single pruning instance. This can easily be done by keeping a frozen mask.
3. At 40% accuracy, prune further to 75% of target sparsity.
4. At 60% accuracy, prune further to 100% of target sparsity.
5. Remember to train after each stage until validation accuracy improves to the required target (you may use fewer epochs or a subset of the train dataset if you face computational issues).
6. Profile again after generating sparse tensors using `torch.sparse` as in Part (a).

Note that the numbers of accuracy here on which to increase pruning are not strict; you can improvise based on the evolution you observe. We are more concerned with the correctness of the process.

Deliverables:

- Record results for Parts (a) and (b) in a table in the markdown file.

- Include comments on expected vs. observed changes in:
  - Model size
  - Performance (latency/memory/energy)
  - Accuracy compared to baseline
- Discuss why the observed results do or do not match expectations. Please note that we are not using any explicit sparse inference optimization kernels. Are they being used under the hood?
- Go through both SNIP and GrasP implementations as defined in the papers. Would you have expected better results with SNIP? Why or why not?

3 Task 2: Structured Pruning

(a) **Regression-Based Channel Pruning**

In this task, you will implement structured pruning (channels) of convolutional layers using the regression-based method introduced in He et al., 2017.

The pruning algorithm can be divided into the following steps:

1. **Collecting input–output pairs.**
   For a given convolutional layer with weights $W \in \mathbb{R}^{n\times c\times k_h\times k_w}$ and input feature maps $X \in \mathbb{R}^{N\times c\times h\times w}$ (the inputs for each layer will be collected using a small calibration dataset):
   (a) Using `torch.nn.functional.unfold`, extract all $k_h \times k_w$ sliding windows from X. After unfolding, reshape so that
   $$X_{\text{unf}} \in \mathbb{R}^{(N\cdot L)\times(c\cdot k_h\cdot k_w)},$$
   where $L = h_{\text{out}} \times w_{\text{out}}$ is the number of output positions.
   (b) Reshape the convolutional filters to
   $$W_{\text{mat}} \in \mathbb{R}^{(c\cdot k_h\cdot k_w)\times n}.$$
   (c) Compute the output activations as
   $$Y = X_{\text{unf}} W_{\text{mat}} \in \mathbb{R}^{(N\cdot L)\times n}.$$
2. **Subproblem (i): Solve for β (channel selection).**
   (a) For each input channel i, define
   $$Z_i = X_i W_i^{\top}, \quad Z_i \in \mathbb{R}^{(N\cdot L)\times n},$$
   where X_i is the slice of X_{unf} corresponding to channel i and W_i is the slice of W for that channel.
   (b) Formulating the LASSO regression problem:
   $$\hat{\beta} = \arg\min_{\beta} \frac{1}{2N} \left\| Y - \sum_{i=1}^{c} \beta_i Z_i \right\|_F^2 + \lambda \|\beta\|_1.$$
   (c) We can solve this problem using an off-the-shelf LASSO solver (e.g., `sklearn.linear_model.Lasso`). Channels with $\beta_i = 0$ are pruned (or more simply, just set the k lowest non-zero values of β to zero). Continue adjusting λ until only c_0 channels remain.
3. **Subproblem (ii): Solve for W (weight reconstruction).**
   (a) Construct the reduced design matrix
   $$X' = [\beta_1 X_1,\ \beta_2 X_2,\ \dots,\ \beta_{c_0} X_{c_0}] \in \mathbb{R}^{(N\cdot L)\times(c_0\cdot k_h\cdot k_w)}.$$
   (b) Solve the least-squares regression problem
   $$W' = \arg\min_{W'} \ \|Y - X' W'^{\top}\|_F^2.$$
   This can be solved using `torch.linalg.lstsq`.

(c) Reshape W' back to convolutional form (n, c_0, k_h, k_w).

While implementing, keep the following instructions in mind:

1. Apply pruning sequentially, starting from the second convolutional block.
2. Maintain the same sparsity ratio as decided in Task 1 (via sensitivity analysis). Alternatively, you can adjust the ratio for each layer on an ad-hoc basis.
3. You will first apply step i from the paper iteratively (or in a single iteration) until the target sparsity is reached for the layer.
4. Remember: pruning input channels in one layer requires pruning corresponding output channels in the previous layer. This means you will need to achieve a lower sparsity than your target in the current layer, as some sparsity will be compensated for by pruning the next layer.
5. Remember to adjust the weights W for each layer after its pruning based on the least square solution as in the paper (step ii).
6. Continue pruning through the network, layer by layer. You can still prune the linear layers for this task using unstructured pruning as before. However, it is not necessary. You still need to meet the overall sparsity ratio, consistent with the previous tasks.
7. Finetune for a final time over a few epochs based on the task loss (Cross Entropy) after all pruning is done.
8. Profile as done in previous tasks and report results in the table. We don't need any specialized libraries for sparse convolution here, since the remaining filters are dense.

Deliverables:

- Record results for Parts (a) and (b) in a table in the markdown file.
- You may decide the number of fine-tuning epochs depending on your compute budget and required accuracy.
- Did you see an improvement in performance (memory, latency) while meeting the accuracy constraint? Why or why not?

References

1. Y. He, J. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017. arXiv:1707.06168.

2. H. Lee, J. A. Park, and W. Shin, "SNIP: Single-shot Network Pruning based on Connection Sensitivity," in *International Conference on Learning Representations (ICLR)*, 2019. arXiv:1810.02340.

3. T. Wang, C. Zhang, Y. Xu, J. Zou, and Y. Ma, "Picking Winning Tickets Before Training by Preserving Gradient Flow," in *International Conference on Learning Representations (ICLR)*, 2020. arXiv:2002.07376.

4. Han Lab, MIT, "6.5940: Hardware-Aware Efficient Deep Learning (Fall 2024)," https://hanlab.mit.edu/courses/2024-fall-65940.

5. "chenyaofo/pytorch-cifar-models," GitHub repository. https://github.com/chenyaofo/pytorch-cifar-models.

6. "chenyaofo/image-classification-codebase," GitHub repository. https://github.com/chenyaofo/image-classification-codebase.

7. PyTorch, "Profiler Recipe," Documentation. https://docs.pytorch.org/tutorials/recipes/recipes/profiler$_r$*ecipe.html*.

8. PowerAPI-NG, "pyJoules," GitHub repository. https://github.com/powerapi-ng/pyJoules.

9. Z. Liu, "torchprofile," GitHub repository. https://github.com/zhijian-liu/torchprofile.

10. PyTorch Documentation, "torch.sparse and Sparse COO Tensors," https://pytorch.org/docs/stable/generated/torch.sparse$_{c}oo_{t}ensor.html$.

11. PyTorch Documentation, "torch.sparse.mm," https://pytorch.org/docs/stable/generated/torch.sparse.mm.html.

12. ICLR Blog, "Benchmarks for Hessian-Vector Products," https://iclr-blogposts.github.io/2024/blog/bench-hvp/.