Harvard CS 2420: Computing at Scale (Fall 2025) **(Due: 10/6/2025 at 11:59 PM ET)**

Problem Set 2 (PSet 2)

Instructor: Prof. HT Kung ***Team:*** Student 1, Student 2, Student 3, Student 4

Please include the names and email addresses of all team members in your submission.
Please use Google Colab as your coding environment for Problem Set 2. (See Section 1)

Part 1.1 (10 points)

1. In Code Cell 1.4, set the number of epochs (`epochs`) to 5. Train the model 3 separate times, with learning rate (`lr`) set to 0.0001, 0.1, and 1.0.
2. For each model run over 5 epochs, plot the training loss (`train_loss_tracker`) and test accuracy (`test_acc_tracker`). For the training loss, apply the provided `moving_average` function on `train_loss_tracker` before plotting to smooth out the loss curve.
3. Plot training loss and test accuracy figure for each run with the different learning rates (0.0001, 0.1, 1.0). (x-axis is epoch)
4. Describe the difference in trends for each learning rate run. Which learning rate seemed to work best? Explain why you think it did best relative to the other learning rates. (100 words maximum)

(Solution)

Part 1.2 (10 points)

1. First, try the MultiStepLR learning rate scheduler. Set `lr_scheduler` (Line 96 in Code Cell 1.4) as 'multistep'. In addition, you also need to set the milestones (`milestones`) used in the scheduler to decrease the learning rate by a factor of 10 every 25 epochs. Train the network for 100 epochs and plot the training loss and test accuracy.
2. Then try the CosineAnnealingLR learning rate scheduler. Set `lr_scheduler` (Line 96 in Code Cell 1.4) as 'cosine_annealing'. Train the network for 100 epochs and plot the training loss and test accuracy (where the x-axis is the number of epochs).
3. Describe the trends of the learning curves of MultiStep and CosineAnnealing respectively. (50 words maximum)
4. Record and report the provided total running time.

(Solution)

Part 2.1 (20 points)

1. In Code Cell 2.2, implement the `quantize` function, which takes a 1D or 2D NumPy array and quantizes each element to be representable with 8 bits (i.e., with an integer value between 0 and 255). The function should return a NumPy array with `dtype=uint8`. Keep in mind that the quantized values will need to be dequantized (converted back to a non-quantized form), so you may also return any other value you may find useful for dequantization.
2. In Code Cell 2.2, implement the `dequantize` function which takes in a NumPy array with `dtype=uint8` (and any other parameters you think necessary), and attempts to recover the values from before quantization. Because quantizing to 8 bits loses information, you may not necessarily obtain the exact weight values from before quantization. Ideally, the quantization error is small enough to maintain good training accuracy.
3. In Code Cell 2.2, verify that both your `quantize` and `dequantize` functions work by making sure it passes the test case. The test case checks that your quantization function achieves a 4x reduction in memory and asserts that the dequantized data is within some error threshold of the original matrix. You should see "Success!" if your method passes a test case.
4. How much smaller is the quantized data than the original? (Remember to account for the extra arguments you added!) Is it 4x? If not, why might this be? (50 words maximum)

(Solution)

Part 2.2 (20 points)

1. Quantize each parameter of the model to 8 bits and dequantize (this injects simulated quantization error into the parameters). Then, create a new model from the dequantized parameters.
2. Evaluate the above model and report the test accuracy. What was the drop/increase in accuracy versus the original full-precision model? (50 words maximum)
3. Quantize to 4 bits instead of 8 and dequantize. What accuracy is achieved from these new 4-bit quantized-and-dequantized parameters? What was the drop/increase in accuracy versus the original full-precision model? (50 words maximum)
4. Quantize to 2 bits instead of 4 and dequantize. What accuracy is achieved from these new 2-bit quantized-and-dequantized parameters? What was the drop/increase in accuracy versus the original full-precision model? (50 words maximum)

(Solution)

Part 2.3 (20 points)

1. Modify the `quantize` function in Code Cell 2.2 to implement **Stochastic Rounding** instead of standard Banker's Rounding. Quantize each parameter of the model to 2 bits and dequantize. Repeat the experiment 10 times in Code Cell 2.4 and answer the following questions. What accuracy is achieved from these new Stochastic Rounded parameters? What was the drop/increase in accuracy versus the original 2-bit model that uses standard Banker's Rounding? (50 words maximum)

(Solution)

Part 3.1 (5 points)

1. In Code Cell 3.2, implement `SparseConvNet` using the `sparse_conv_block` in a similar fashion to Code Cell 1.3. Replace all the `nn.Conv2d` layers in `ConvNet` with the `SparseConv2d` layers provided in Code Cell 4.1.

(Solution)

Part 3.2 (5 points)

1. Using Code Cell 3.3, train `SparseConvNet` for 5 epochs with a learning rate of 0.1. Confirm that this performance is approximately equal to what you observed in Part 1.1. (Note that this current model is not sparse, as no pruning has yet been performed. You will implement pruning in the next part. The purpose here is to validate that the performance is the same as the standard convolution.)
2. Plot the training error and test accuracy of `SparseConvNet` trained over 5 epochs. (x-axis is epoch)

(Solution)

Part 3.3 (25 points)

1. Implement the `filter_l1_pruning` function in Code Cell 4.3 and set the pruning schedule (using `prune_percentage` and `prune_epoch`) to prune an additional 10% filters every 10 epochs, starting at epoch 10, ending at epoch 50. By the end, you should achieve 50% sparsity for each convolution layer in the CNN. For simplicity, you do not need to prune the `nn.Linear` layer, which is the final layer in the model.
2. Train `SparseConvNet` for 100 epochs using the same learning rate and `MultiStep` learning rate schedule as in Part 1.2.
3. Compare the test accuracy curves against the baseline `ConvNet` model from Part 1.2 in a single plot.
4. Describe any observations in trends of test accuracy related to the pruning stages. (150 words maximum)

(Solution)

Part 3.4 (25 points)

1. Use the `SparseConv2d` in Code Cell 3.4 (the layer using *unstructured pruning*) to replace the original `SparseConv2d` in Code Cell 3.1 (the layer using *structured pruning*).
2. Implement the `unstructured_pruning` function in Code Cell 3.4 and use it to replace the `filter_l1_pruning` function in Code Cell 3.3.
3. Re-run training with `unstructured_pruning` (i.e., Code Cell 3.1, 3.2, 3.3). You may make copies of those Code Cells if that is easier for you.
4. Compare the 3 test accuracy curves among the baseline ConvNet model from Part 1.2, the structured pruning model from Part 3.3, and the unstructured pruning model from Part 3.4 in a single plot.
5. Describe any observations in the comparison of structured and unstructured prunings. (150 words maximum)

(Solution)

Part 4.1 (25 points)

1. In Code Cell 4.1, instantiate a pre-trained ResNet-18 model (`IMAGENET1K_V1` weights) using the `torchvision` library.
2. In Code Cell 4.2, implement the freezing of earlier layers in the model. As long as the final classification layer is not frozen, you are free to choose whichever layers to freeze.
3. In Code Cell 4.3, replace the final classification layer of the ResNet-18 model with a LoRA linear layer.

(Solution)

Part 4.2 (10 points)

1. You will train and evaluate the aforementioned models in Code Cell 4.4 through Code Cell 4.6. Plot the test accuracy curves (accuracy vs epoch) on the same graph for these three models: the baseline pre-trained model (`pt_net`), the partially frozen model (`net_freeze`), and the model with both frozen layers and a LoRA classification layer (`net_freeze_lora`). Note: if you make any adjustments to your models, make sure to run Code Cells 4.1—4.3 in sequential order and before Code Cells 4.4—4.6 to avoid unexpected behavior.

(Solution)

Part 5.1 (35 points)

1. In this problem, you will be implementing the transformer model architecture. Using your code, you will train a small language model in the next problem set. Follow the instructions on the notebook and complete all the "TODO"'s in the problem. Here, briefly detail your process, noting any hyperparameters within the architecture that is important. Using a table can be helpful.
2. How many matrix multiplications do we have for a single inference in this transformer model? Where are these multiplications happening? Remember that there are 'h' heads. You can ignore the matrix multiplication in the 'WordEmbedding' layer.

(Solution)

Part 6.1 (25 points)

1. Implement the `DepthwiseSeparableConvolution` module in PyTorch. (Do not use the "groups" parameter of Pytorch `nn.Conv2d` to implement this – we want you to understand the details and mechanics of the operation!)
2. Verify the correctness of `DepthwiseSeparableConvolution`. (Code Cell 6.1 should print "Success...!" after you run it.)
3. Verify that `DepthwiseSeparableConvolution` reduces the number of parameters by a factor of the number of output channels by printing out the number of parameters for both a standard convolution and the depthwise separable convolution. By what factor did depthwise separable convolutions reduce the parameters? Does this match what should theoretically happen? (50 words maximum)

(Solution)

Part 6.2 (15 points)

1. Implement the `dw_conv_block` function which performs in sequence: depthwise separable convolution, batch normalization, ReLU.
2. Train the neural network for several epochs with your implementation of `DepthwiseSeparableConvolution`; compare the runtime with `DepthwiseSeparableConvolutionSolution`. How much faster is the solution versus your implementation? Why might this be? (50 words maximum)
3. Train the neural network for 100 epochs and plot the training error and test accuracy (x-axis is epoch).

(Solution)

1 What to Submit

Your submission should be a `.zip` archive with a `CS2420_PSet2_` prefix followed by all your team members' names. The archive should contain:

- PDF write-up
- A **single** Colab Notebook with all the outputs in place
- Text files or PDFs containing the complete outputs (e.g., ChatGPT logs) of all generative AI tools used.

Example filename: `CS2420_PSet2_Name1_Name2_Name3_Name4.zip`

Write-up

Written responses should be contained within a single PDF document. (LaTeX is highly recommended!) Each response or figure should clearly indicate which problem is being answered. Please include all required figures in the write-up.

Colab Notebook

All outputs from the Colab Notebook **must** be saved. Do not delete code, comment it out if needed. TFs must be able to reproduce your results. You will not receive credit for implementations that are not reproducible.