Harvard CS 242: Computing at Scale (Fall 2024) **(Due: 10/9/2024 at 11:59 PM EST)**

Programming Assignment 2

Instructor: Prof. HT Kung ***Team:*** Anmay Gupta, Carlos Luz, Elvin Lo, Jaray Liu

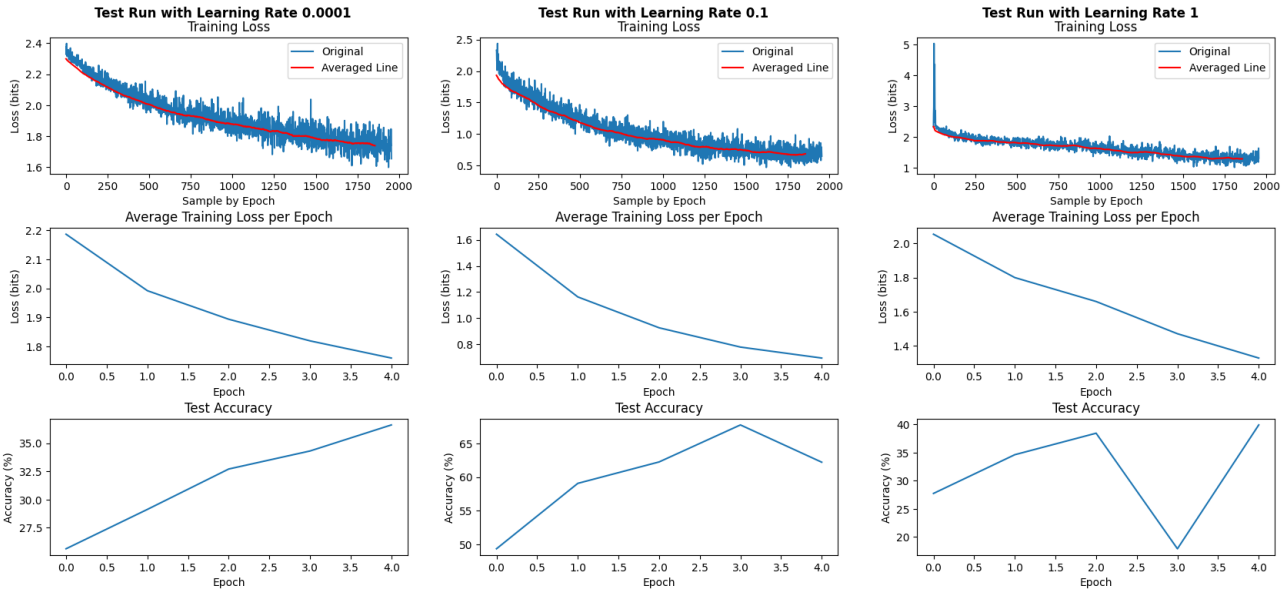
Please include the full names of all team members in your submission.

Part 1.1

(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) This gives us the following plots:



The model with a learning rate of 0.0001 finished with a final test accuracy of 36.52% in 133 seconds, the model with a learning rate of 0.1 finished with a final test accuracy of 62.21% in 135 seconds, and the model with a learning rate of 1 finished with a final test accuracy of 39.92% in 136 seconds.

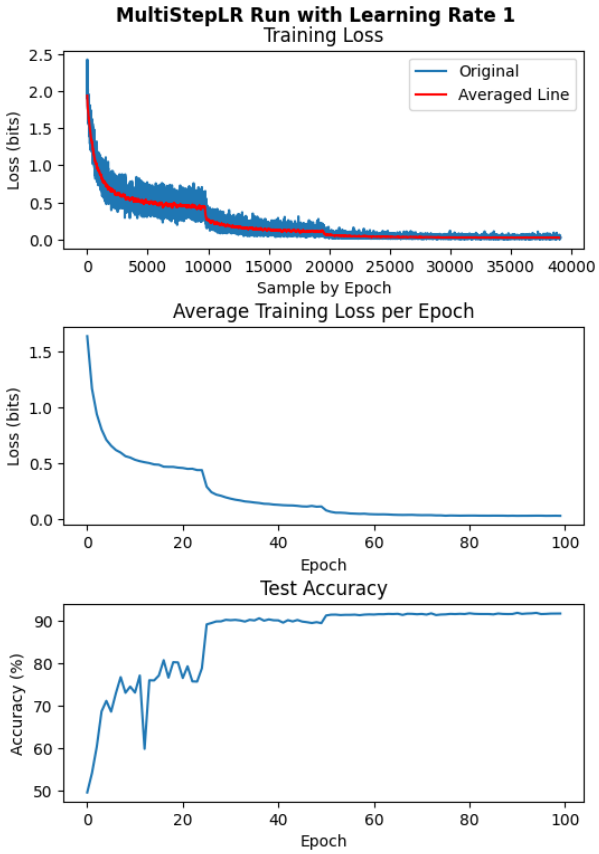
Overall, the 0.1 learning rate seemed to work best as it was able to achieve the highest test accuracy with a similar training time as the other models and being able to quickly converge on a solution. Meanwhile, the model with a learning rate of 0.0001 was much slower to converge and the model with a learning rate of 1 was able to learn rather quickly but also had far more variation in its performance, leading to lower final test accuracies for both models.

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.

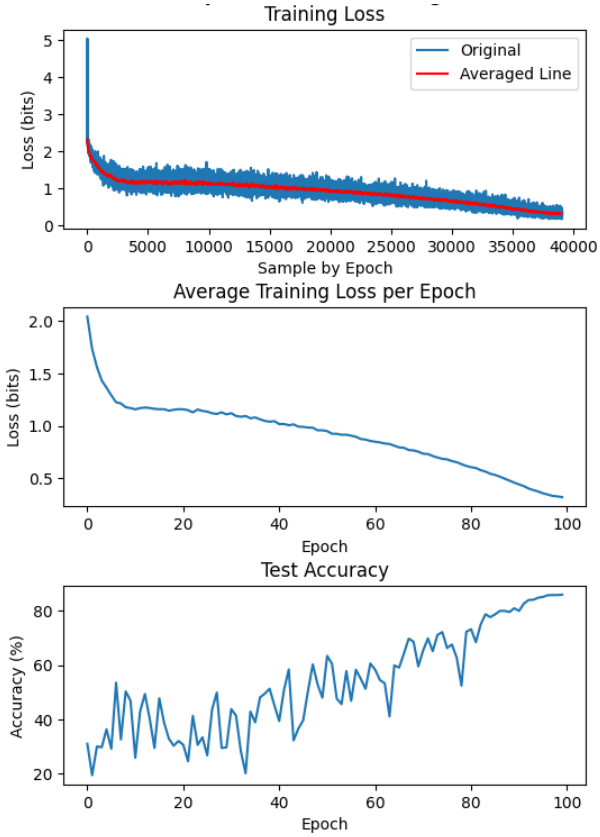
(Solution)

1. This gives us the following plot:



The final training loss was 0.028 bits, the final training accuracy was 99.490%, the final test loss was 0.288 bits, and the final test accuracy was 91.830% but peaked at 91.990%.

2. This gives us the following plot:



In our final epoch, the training loss was 0.319, training accuracy was 88.990%, test loss was 0.415 bits, and test accuracy was 85.990%.

3. The multistep scheduler yields a gradual learning curve in between milestones, but at milestones the model often sees sudden and significant improvements in training and test performance, e.g., the jumps we see at epochs 25 and 50. Meanwhile, the cosine annealing scheduler yields a smooth learning curve without abrupt changes.

4. The total running time for the MultiStepLR learning rate scheduler was 2711 seconds (about 45 minutes), while the total running time for the CosineAnnealingLR learning rate scheduler was 2534 seconds (about 42 minutes).

Part 2.1 (5 points)

1. In Code Cell 3.1, 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 3.1, 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 3.1, 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)

1. See code. For 8-bit quantization, we return the scale and shift quantization parameters in addition to the quantized values.
2. See code.
3. See code. We pass all test cases.
4. The quantized data does not quite achieve the ideal 4x size reduction. In particular, it is 84 bytes larger for 1D arrays and 96 bytes larger for 2D arrays. This is due to the fixed overhead of the numpy array objects—since quantization only scales the size of the data buffer, not any metadata or structural information.

   Note also that an additional 8 bytes are used for the two quantization parameters which we store as full precision floats.

Part 2.2 (10 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 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 in accuracy versus the original full-precision model? (50 words maximum)

(Solution)

1. See the code implementation.
2. Our 8-bit quantization shows only a 0.07% accuracy drop compared to our full precision baseline. In particular, our baseline model yields average loss 0.1179 and accuracy 97.22%, while our quantized model yields average loss 0.1170 and accuracy 97.15%.
3. Even at 4-bit quantization, our quantized model shows only a 0.35% accuracy drop compared to our full precision baseline. In particular, our baseline model yields average loss 0.1179 and accuracy 97.22%, while our quantized model yields average loss 0.1231 and accuracy 96.87%.

Part 3.1 (5 points)

1. In Code Cell 4.2, 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) See notebook Code Cell 3.2

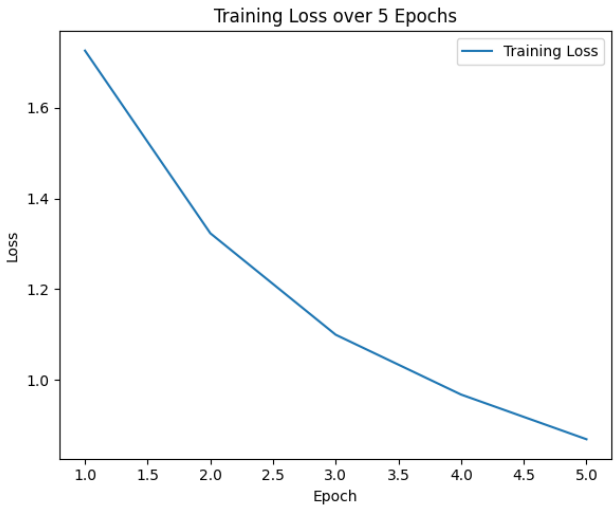
Part 3.2 (5 points)

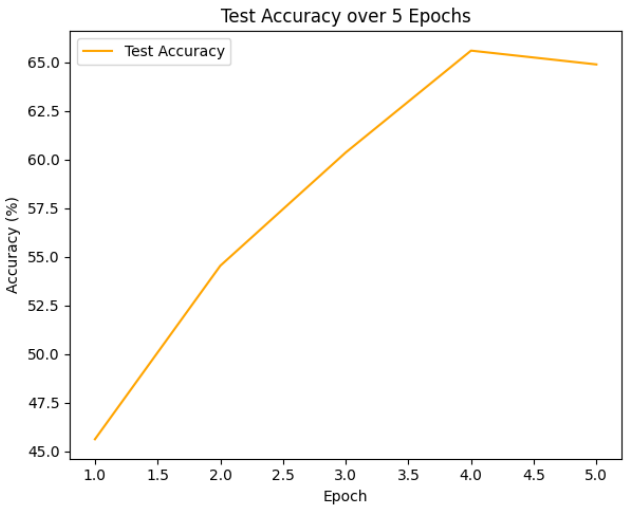
1. Using Code Cell 4.3, train `SparseConvNet` for 5 epochs with a learning rate of 0.1. Confirm that this performance is approximately equal to what you observed in Part 1.1. (Note that this current model is not sparse, as no pruning has yet been performed. You will implement pruning in the next part. The purpose here is to validate that the performance is the same as the standard convolution.)

   (Solution) See notebook Code Cell 3.3 for implementation (no pruning).

2. Plot the training error and test accuracy of `SparseConvNet` trained over 5 epochs. (x-axis is epoch)

   (Solution) For 5 epochs with learning rate $\eta = 0.1$, we find the training error and test accuracy of `SparseConvNet` to be given over 5 epochs as follows:





   For $\eta = 0.1$, this is largely consistent with our result from Part 1.1 where we saw test accuracy peak at 65% and loss drop off to below 1 by the 4th/5th epoch.

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.

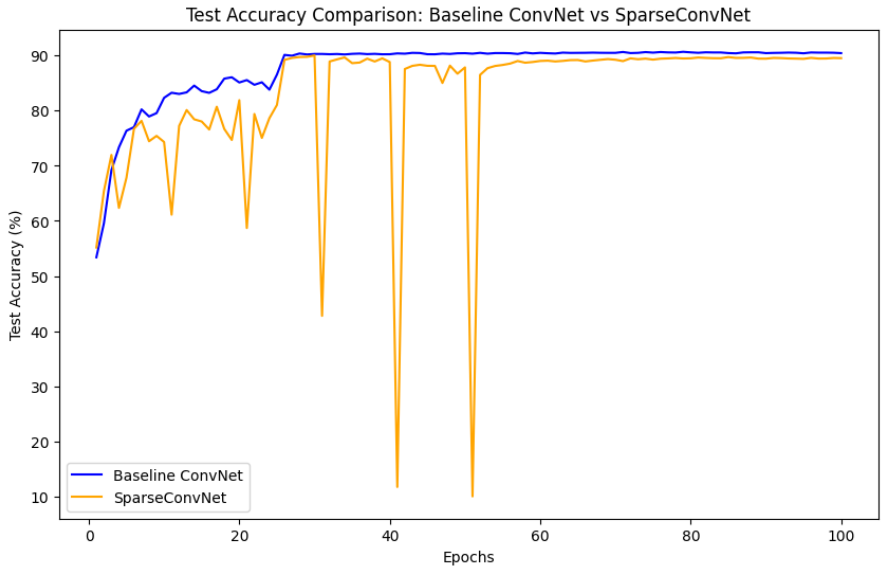
   (Solution) See Code Cell 3.3 for implementation.

2. Train `SparseConvNet` for 100 epochs using the same learning rate and `MultiStep` learning rate schedule as in Part 1.2.

   (Solution) See Code Cell 3.3 for implementation and training feedback.

3. Compare the test accuracy curves against the baseline `ConvNet` model from Part 1.2 in a single plot.

   (Solution) We find the following test accuracy curve (over 100 epochs) for both the baseline (as described in Part 1.2) and the trained `SpareConvNet`. For consistency, both use $\eta = 0.1$ with η degradation by a factor of 10 every 25 epochs. That is $\gamma = 0.1$:



4. Describe any observations in trends of test accuracy related to the pruning stages. (150 words maximum)

   (Solution) There are several notable observations pertaining to the pruning stages.

   The test accuracy shows a noticeable fluctuation during the pruning stages. Initially, the `SparseConvNet` achieves competitive accuracy, even slightly outperforming the baseline `ConvNet` in early epochs. However, as pruning is applied (starting at epoch 10 and repeated every 10 epochs), the test accuracy temporarily drops or stagnates, indicating that removing filters affects the network's ability to generalize. Despite this, the model recovers in subsequent epochs, suggesting it can adapt to the reduced network capacity after pruning.

   Notably, pruning seems to introduce instability in the accuracy, but it does not result in a catastrophic accuracy loss. Thus, the `SparseConvNet` maintains comparable performance to the baseline while benefiting from a more efficient, compact model structure. Such demonstrates the trade-off between accuracy and network efficiency when structured pruning is applied.

Part 3.4 (25 points)

1. Use the `SparseConv2d` in Code Cell 4.4 (the layer using *unstructured pruning*) to replace the original `SparseConv2d` in Code Cell 4.1 (the layer using *structured pruning*).

   (Solution) See Code Cell 3.1 (CLONE) for implementation.

2. Implement the `unstructured_pruning` function in Code Cell 4.4 and use it to replace the `filter_l1_pruning` function in Code Cell 4.3.

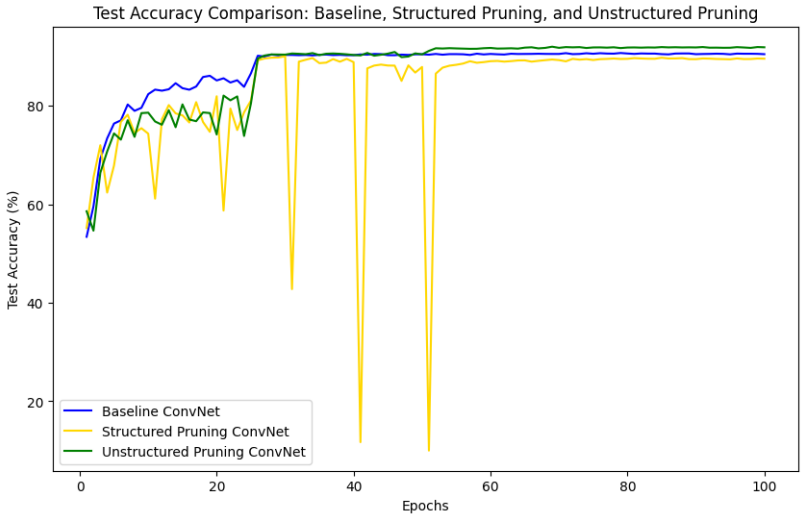
   (Solution) See Code Cell 3.4 and Code Cell 3.3 (CLONE); the latter involved a 1-line change.

3. Re-run training with `unstructured_pruning` (i.e., Code Cell 4.1, 4.2, 4.3). You may make copies of those Code Cells if that is easier for you.

   (Solution) We make copies of Code Cells 3.1, 3.2, 3.3 and name them Code Cell 3.X (CLONE). See Code Cell 3.1 (CLONE), Code Cell 3.2 (CLONE), and Code Cell 3.3 (CLONE) for feedback by epoch (summarized in graph in next question).

4. Compare the 3 test accuracy curves among the baseline ConvNet model from Part 1.2, the structured pruning model from Part 4.3, and the unstructured pruning model from Part 4.4 in a single plot.

   (Solution) We summarize our test accuracy results over 100 epochs in the graph below:



   For clarity, the training setup for the unstructured pruning ConvNet uses the same learning rate of $\eta = 0.1$ and $\gamma = 0.1$. Further, the same number of epochs, pruning schedule, and milestones are used for consistency across models.

5. Describe any observations in the comparison of structured and unstructured prunings. (150 words maximum)

   (Solution) In comparing structured and unstructured pruning, we observe that structured pruning results in sharp drops in accuracy after pruning steps, particularly around epochs 30 and 40, but the model is able to recover over time. This is to be expected since structured pruning removes entire filters, which significantly reduces the network's capacity, leading to more immediate performance degradation.

   On the other hand, unstructured pruning shows a smoother accuracy curve with minimal dips, and seemingly outperforms the baseline model toward the end. This is likely due to its fine-grained approach, where

individual weights are pruned, allowing the model to retain more of its expressive power and gradually adjust to the reduced weight count without large accuracy losses. The obvious tradeoff however is that unstructured pruning introduces irregular sparsity in the network, which makes it difficult to optimize for hardware efficiency, as sparse weight matrices are not as easy to accelerate on normal hardware. While unstructured pruning may maintain higher accuracy, it can be less efficient in terms of computational speed and memory use compared to structured pruning, which removes entire filters and is generally more hardware-friendly.

Part 4.1 (25 points)

1. In Code Cell 5.1, instantiate a pre-trained ResNet-18 model (`IMAGENET1K_V` weights) using the `torchvision` library.
2. In Code Cell 5.2, 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 5.3, replace the final classification layer of the ResNet-18 model with a LoRA linear layer.

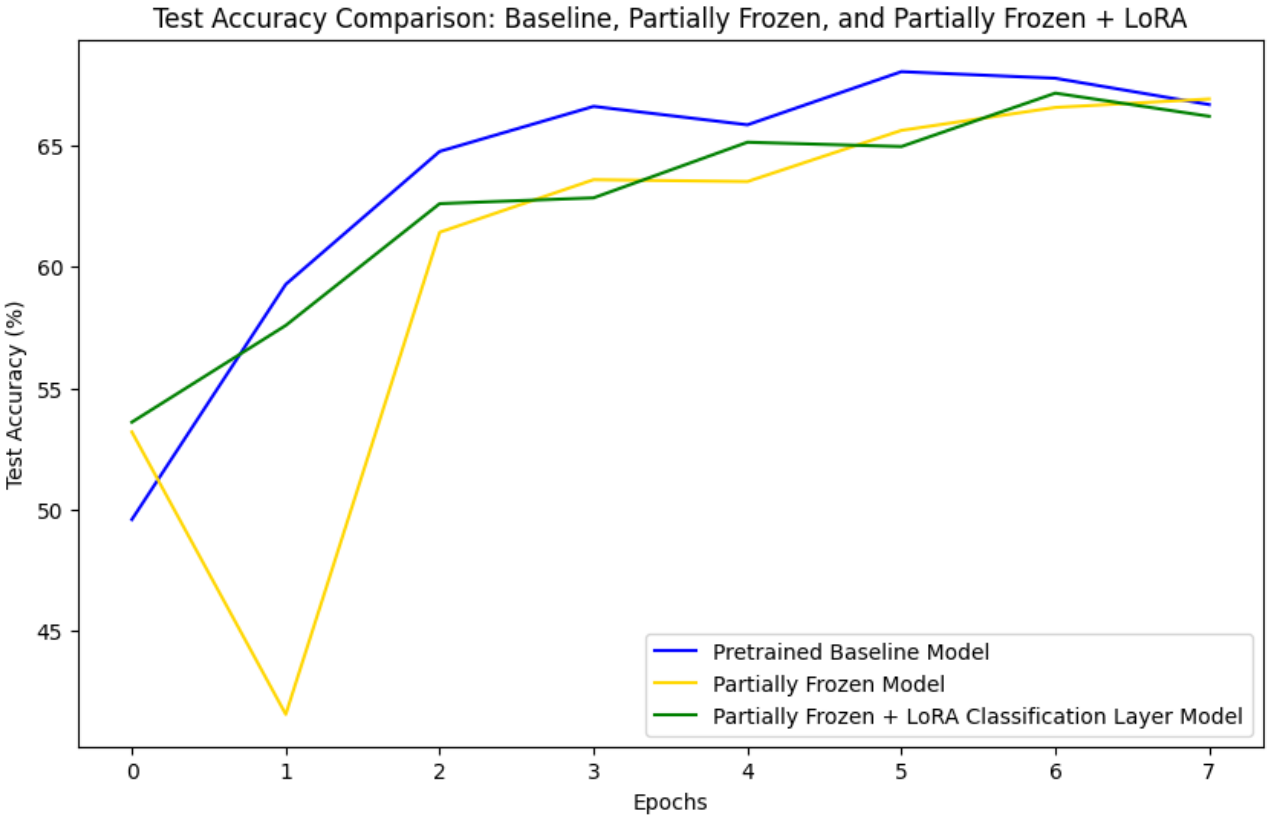
(Solution)

1. See code.
2. See code.
3. See code.

Part 4.2 (10 points)

1. You will train and evaluate the aforementioned models in Code Cell 5.4 through Code Cell 5.6. 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 5.1—5.3 in sequential order and before Code Cells 5.4—5.6 to avoid unexpected behavior.

(Solution) We trained each of the three models for 8 epochs each and plotted their respective test accuracy in relation to epoch on the same graph.

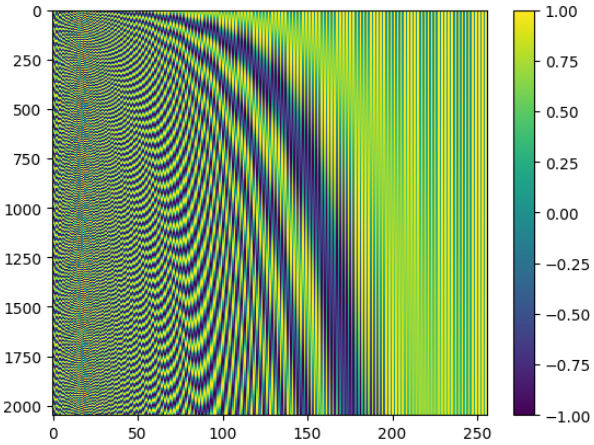


Part 5.1 (35 points)

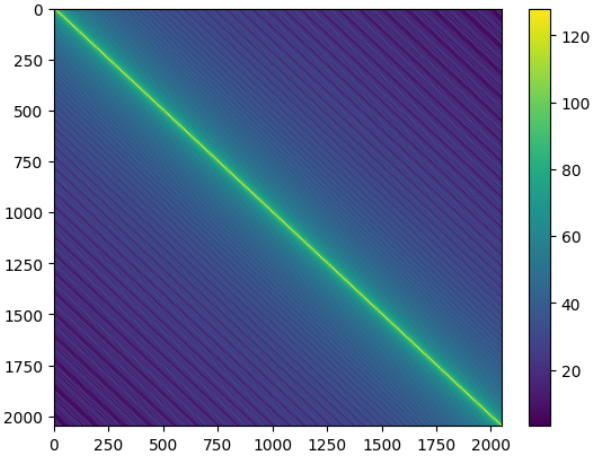
1. In this problem, you will be implementing the transformer model architecture. Using your code, you will train a small language model in the next problem set. Follow the instructions on the notebook and complete all the "TODO"'s in the problem. Here, briefly detail your process, noting any hyperparameters 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)

1. **Part 5.1.1(b):** Plotting the positional encoding matrix, we obtain the following plot:



where the y-axis is the length of the input sequence while the x-axis is the dimension of the output encoding space (`d_model`). Note that for each word's position in the sequence, a different d-dimensional output vector (a row in the above image) is generated. To enable this, each of these d-dimensional output vectors has a different frequency, but there remains a relative position difference since the d-dimensional output vector for a position $p+k$ is a linear function of the vector for a position p. We can demonstrate this relative position difference by multiplying our positional encoding with its transpose, effectively taking the dot product of all the row vectors with one another.



There's a center line with a high magnitude (as expected), but there also exist parallel lines that correspond to the linear transformations between relative positions. These lines allow for both absolute and relative positional encodings.

2. The transformer model architecture has been implemented in the notebook as described, giving us the following hyperparameter table:

Hyperparameter	Impact
Number of Layers	Increases accuracy in exchange for training + inference time
Number of Heads	Allows references to different spaces and positions
Model Embedding Dimension	Output for residual connections + embedding layers
Feed Forward Depth	Allows long-range modeling via learnable mappings
Input Vocabulary Size	Application dependent; Increases accuracy and training time
Target Vocabulary Size	Application dependent; Increases accuracy and training time
Dropout Rate	Percent of network randomly set to 0; prevents overfitting
Batch Size	Number of training examples used per training step

3. **Part 5.1.2** For a single inference, we have 3 matrix multiplications per head in each multiheaded attention layer to multiply the input X with the query weights, key weights, and value weights. These matrices are then put through 2 more matrix multiplications to compute the attention for each head, giving us $5h$ total matrix multiplications. Then, in each multiheaded attention layer, the attention are concatenated and again multiplied by W^O, so there are $5h + 1$ matrix multiplications per multiheaded attention layer (note that this also applies to the masked multiheaded attention layer as the masked multiheaded attention layer doesn't require more matrix multiplications). Adding in the 2 additional matrix multiplications from the 2-layer feed forward neural network, we have that there are $5h + 3$ multiplications in the encoder and $10h + 4$ multiplications in the decoder. Adding in the single matrix multiplication from the final linear layer in the transformer, there are $\boxed{N \times (15h + 7) + 1}$ matrix multiplications in a transformer with N layers and h heads per multiheaded attention layer.