Memory-Efficient Training of Large-Scale Deep Learning Models Using K-Means Compression

Optimizing Parameter Storage for Enhanced Training Efficiency

Xinhao Ying

December 5, 2024

https://github.com/Luyooooom/CSCI-6962-G1-Project

Agenda

Introduction

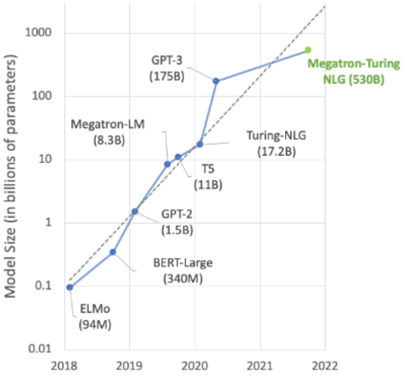
- **Optimization Problem**
- **K-Means for Model Compression**
- **Applying Compression to Models**
- **Experimental Validation**

Expected Contributions

Expected Contributions:

- Introduce a K-Means-based compression technique for deep learning models.
- Demonstrate the effectiveness of K-Means optimizations in compression performance.
- Provide empirical results showing memory savings with minimal accuracy loss.
- Offer insights into integrating compression methods with existing training frameworks.

Background: Challenges in Training Large-Scale Models



Memory Calculation for GPT-3 using float32 (175B Parameters)

$$\text{Memory} = 175 \times 10^9 \times 4 \text{ bytes} = 700 \text{ GB}$$

Current GPU Memory Capacity:
High-end GPUs, like the NVIDIA A100, typically offer 40GB or 80GB of memory.

Optimization Problem

Can we store parameters with less memory during training?

Let W_{ij} be the weight matrix of a layer. Given the number of centroids s, the objective is to minimize the sum of squared distances between each parameter W_{ij} and its closest centroid y_s:

$$\min_{y} \sum_{i,j} \min_{s}(W_{ij} - y_s)^2$$

After finding the centroids, we replace each parameter with its closest centroid:

$$W'_{ij} = y^*_s \quad \text{where} \quad s = \arg\min_{s'}(W_{ij} - y_{s'})^2$$

This results in compressed parameter storage.

3.6	3.9	3.9
6.2	11.0	10.0
1.1	1.4	1.7

⇒

3.8	3.8	3.8
6.2	10.5	10.5
1.4	1.4	1.4

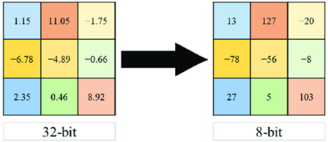
Compression Methods in Inference

Quantization:

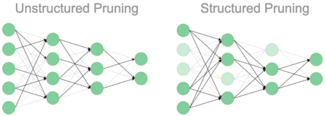
- Reduces model size and latency (up to 4x) by using lower precision formats.
- Potential trade-off: May lead to accuracy loss.

Pruning:

- Sets some weights to zero to save space and computation.
- Potential trade-off: Requires sparse execution, may lose accuracy.



Quantization

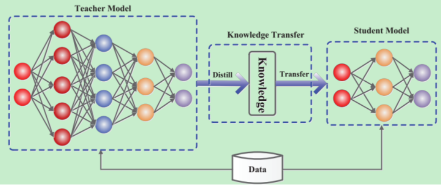


Pruning

Compression Methods in Inference

Knowledge Distillation:

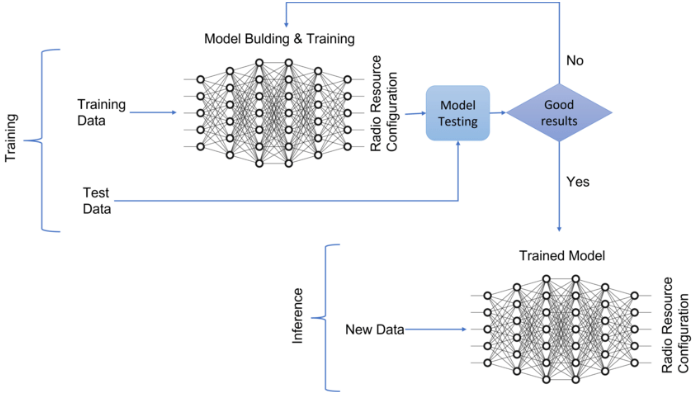
- Trains a smaller model by transferring knowledge from a larger one.
- Potential trade-off: Training is computationally expensive.



Knowledge Distillation

Additional Notes: These optimization methods significantly reduce model size and improve efficiency during inference, but they often incur additional memory and computational costs during training.

- These are methods for compressing models during inference.
- Some methods, such as low-rank matrix approximation, have successfully reduced memory usage during training by modifying the model structure.
- Can we store parameters with less memory during training?



K-means Algorithm

This problem has been proven to be NP-hard.

Algorithm 1 K-means Clustering

Require: Data matrix X, number of clusters s, maximum iterations T

Initialize k centroids $\{y_1, y_2, \ldots, y_s\}$ randomly or using K-means++

for each iteration $t = 1, \ldots, T$ **do**

  for each data point $W_{ij} \in W$ **do**

    Assign W_{ij} to the nearest centroid:

$$s_{ij} = \arg\min_{s'} |W_{ij} - y_{s'}|_2^2$$

  end for

  for each centroid $y_{s'}$ **do**

    Update the centroid $y_{s'}$ based on the mean of assigned points:

$$y_{s'} \leftarrow \frac{1}{|C_{s'}|} \sum_{W_{ij} \in C_{s'}} W_{ij}$$

  end for

end for

Ensure: Final centroids $\{c_1, c_2, \ldots, c_k\}$

Visualizing K-means Clustering

K-means++ Initialization

Reduce the problem of local optima in K-means.

Algorithm 2 K-means++ Initialization

Require: Weight matrix W, number of clusters s

1: Select the first centroid y_1 randomly from W
2: **for** each remaining centroid $y_2, \dots, y_s$ **do**
3:     For each element W_{ij}, compute $D(W_{ij})$ as the distance to the nearest chosen centroid
4:     Select W_{ij} as the next centroid y_s with probability proportional to $D(W_{ij})^2$
5: **end for**

Ensure: Initial centroids $\{y_1, y_2, \dots, y_s\}$

Mini-Batch K-means

By approximating updates using smaller batches, significantly improving computational efficiency.

Algorithm 3 Mini-batch K-means Clustering

Require: Weight matrix W, number of clusters s, batch size b, maximum iterations T

Initialize s centroids $\{y_1, y_2, \ldots, y_s\}$ using K-means++ or randomly

for each iteration $t = 1, \ldots, T$ **do**

  Randomly sample a batch of b elements from W

  for each sampled element W_{ij} **do**

    Assign W_{ij} to the nearest centroid:

$$s_{ij} = \arg\min_{s'} |W_{ij} - y_{s'}|_2^2$$

    Update the assigned centroid $y_{s_{ij}}$ based on W_{ij}

$$y_{s_{ij}} \leftarrow y_{s_{ij}} + \eta \left(W_{ij} - y_{s_{ij}}\right)$$

  end for

end for

Ensure: Final centroids $\{y_1, y_2, \ldots, y_s\}$

Compression Process

We assume that W is the weight matrix of a linear layer in a model. Let's now see when it is used during the training process.

Algorithm 4 Model Process with Compression of W

Initialize: Initialize the weight matrix W
Apply K-means on W to obtain cluster centers and store the centers using a specific storage structure to get W'
for each training step **do**
  Forward Pass:
    Compute the output $y = \mathrm{ReLU}(W'x + b)$
  Backward Pass:
    Compute the error term $\delta_{\text{current}} = \frac{\partial L}{\partial y} \cdot \mathrm{ReLU}'(W'x + b)$
    Compute the error term for the input $\delta_{\text{previous}} = (W')^T \delta_{\text{current}}$
  Model Update:
    Compute the gradient $\frac{\partial L}{\partial W'}$
    Update $W \leftarrow W' - \eta \frac{\partial L}{\partial W'}$ or use other optimizer
    Apply compression: Compress W using the cluster centers of W' as initialization to get new W'
end for

Compression Process

Algorithm 4 Model Process with Compression of W

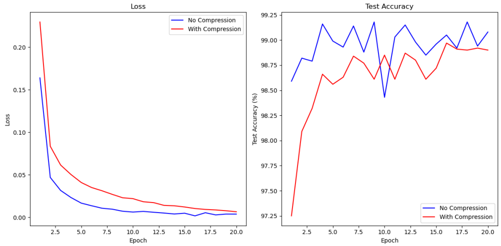
1: **Initialize:** Initialize the weight matrix W
2: Apply K-means on W to obtain cluster centers and store the centers using a specific storage structure to get W'
3: **for** each training step **do**
4: **Forward Pass:**
5: Compute the output $y = \text{ReLU}(W'x + b)$
6: **Backward Pass:**
7: Compute the error term $\delta_{\text{current}} = \frac{\partial L}{\partial y} \cdot \text{ReLU}'(W'x + b)$
8: Compute the error term for the input $\delta_{\text{previous}} = (W')^T \delta_{\text{current}}$
9: **Model Update:**
10: Compute the gradient $\frac{\partial L}{\partial W'}$
11: Update $W \leftarrow W' - \eta \frac{\partial L}{\partial W'}$ or use other optimizer
12: Apply compression: Compress W using the cluster centers of W' as initialization to get new W'
13: **end for**

- Use the previous cluster centers as the initial values for the next K-means iteration.
- Do not retain the uncompressed W to achieve the theoretical memory savings.
- Accessing W' incurs additional costs due to its special storage format, requiring special handling for each access.

Simple CNN on MNIST

- **MNIST Dataset:** 28x28 grayscale images of handwritten digits (0-9). It contains 60,000 training images and 10,000 test images.
- **Compression Applied to FC1:** In this experiment, K-means compression was applied to the fully connected layer (fc1) of the model.

```python
class SimpleCNN(nn.Module):
    def __init__(self, compression=False, s=32):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
```



Parameter Calculation and Memory Reduction

Metric	Non-Compressed	Compressed
Epoch Time	8.5–10.5 seconds	43–51 seconds
Final Accuracy	99.08%	98.90%
Training Loss at Epoch 20	0.0038	0.0065
Memory Reduction in Parameters	-	71.42%

Parameter Count for Each Layer:

- `conv1`: $1 \times 32 \times 3 \times 3 + 32 = 320$ parameters
- `conv2`: $32 \times 64 \times 3 \times 3 + 64 = 18496$ parameters
- `fc1`: $64 \times 7 \times 7 \times 128 + 128 = 401536$ parameters
- `fc2`: $128 \times 10 + 10 = 1290$ parameters

Parameter Calculation and Memory Reduction

Original Data Size:

$$421642 \text{ parameters} \times 32 \text{ bits} = 13492544 \text{ bits}$$

Compressed Data Size:

- `fc1` layer: 401408×8 bits $= 3211264$ bits
- Cluster centers: 32×32 bits $= 1024$ bits
- Uncompressed parameters: 20362×32 bits $= 651584$ bits
- Total Compressed Size: $3211264 + 1024 + 651584 = 3863872$ bits

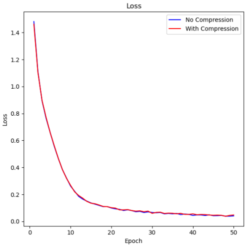
Memory Reduction: $\frac{13492544-3863872}{13492544} \approx 71.4\%$
Compression reduces the data size to about 28.6% of the original size.

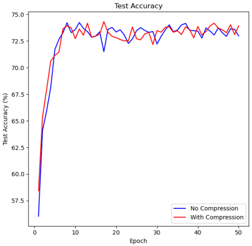
ResNet18 on CIFAR-10

- **Dataset:** CIFAR-10, consisting of 60,000 32x32 RGB images across 10 classes.
- **Model:** ResNet18 architecture.
- **Compression Applied:** The second convolutional layer in the fourth residual block (`layer4[1].conv2.weight`) was compressed using Mini-Batch K-means with 64 centroids.

```
class ResNet18WithCompression(nn.Module):
    def __init__(self, compression=False, s=64):
        super(ResNet18WithCompression, self).__init__()

        self.model = models.resnet18(weights=None)
```





Parameter Count and Compression

Metric	Non-Compressed	Compressed
Epoch Time	20-22 seconds	140-170 seconds
Final Accuracy	72.97%	73.91%
Training Loss at Epoch 50	0.0410	0.0479
Memory Reduction in Compressed Layer	-	15.1%

Parameter Count: By checking the `model.named_parameters()`:

- `layer4[1].conv2.weight`: 2359296 parameters
- **Total Parameters in Model**: 11689512 parameters

Original Data Size:

$$11689512 \text{ parameters} \times 32 \text{ bits} = 374064384 \text{ bits}$$

Parameter Calculation and Memory Reduction

Compressed Data Size:

- `conv2` layer: 2359296×8 bits $= 18874368$ bits
- Cluster centers: 64×32 bits $= 2048$ bits
- Uncompressed parameters: 9330216×32 bits $= 298566912$ bits
- **Total Compressed Size:** $18874368 + 2048 + 298566912 = 317443328$ bits

Memory Reduction:

$$\text{Memory Reduction} = \frac{374064384 - 317443328}{374064384} \approx 15.1\%$$

Compression reduces the data size to about 84.9% of the original size.

Challenges and Conclusion

- **Model Optimality:**
  - The compressed model is, at best, a suboptimal version of the original.
  - Increasing the number of cluster centers can improve the approximation of original parameters.
- **Computational Overhead of K-means:**
  - Despite optimizations, significant time is consumed during compression.(CPU and GPU)
  - Further integration with the model's architecture may be necessary to improve efficiency.
- **Lack of Native Support for Memory Savings:**
  - Existing machine learning frameworks do not natively support this compression method.
  - Custom implementations are required for handling hashed matrices during the forward pass.
- **Conclusion:**
  - Despite challenges, compression does not significantly degrade model accuracy and convergence.
  - Beneficial in scenarios requiring smaller parameter matrices for transmission.

Q&A

Questions?

$$y_{s_{ij}} \leftarrow \frac{N_{s_{ij}} \cdot y_{s_{ij}} + W_{ij}}{N_{s_{ij}} + 1}$$