Group 12: CSCE 614 Term Project Report: Accelerating DNN Training By Adaptive Gradient Prediction (ADA-GP)

Eshaan Mandal (UIN:335006499), Keshav Kishore (UIN:335005932),
Rebecca Robin (UIN:635008133), Neha Daoo (UIN:936002036)

***Abstract*—Gradient prediction techniques have shown significant potential in accelerating the training of deep neural networks (DNNs) by addressing the computational overhead associated with backpropagation. In this work, we aim to implement ADA-GP, a method that leverages adaptive gradient prediction to alternate efficiently between backpropagated and predicted gradients during training. Using PyTorch, we map the fixed-size output of our gradient predictor to the ground truth gradient tensors by repeating the predicted tensor values to match the dimensions of the ground truth. To evaluate the credibility and effectiveness of the approach, we analyze the layer-wise predictor loss, model accuracy, and overall training speedup on standard benchmark datasets. Our implementation not only validates the core principles of ADA-GP but also provides a reproducible framework for further exploration of gradient prediction in efficient deep learning. With our implementation of ADA-GP, we achieved an average speedup of 1.68 over baseline models.**

I. Introduction

Deep Neural Networks (DNNs) have emerged as a revolutionary breakthrough, driving transformations across various domains. DNNs have been employed in diverse applications such as image classification [8], text generation [5], robotics [2] , and disease diagnosis. They are trained on large sets of input data, which is why they perform accurately. Traditional DNN training uses the backpropagation algorithm. In backpropagation, the input data is processed by passing through it, beginning from the first layer to the last. The final layer calculates a predetermined loss function. Then, gradients are obtained based on the computed loss function. The gradients are propagated backward from the last layer to the first while revising the weights for each layer. The weights of a layer can only be updated after all the layers have completed the forward pass and have the gradients propagated back to them. Thus, it is evident that backpropagation is inherently sequential. Being of a sequential nature means that training DNN models using backpropagation is tedious. There have been attempts to reduce the time taken by sequential processes. Most of the work in this area has focused on predicting memory access patterns, branches, or dependencies. The authors of [6] proposed a solution that uses gradient prediction to reduce the training time for sequential DNN models. The primary obstacles while adopting this approach are scalability and the tradeoff between accuracy and performance. Scalability in gradient prediction is hindered by the large number of layers in modern DNNs, often in the hundreds, making it infeasible to assign a predictor to each layer. Consistently using gradient prediction can speed up training by three times by eliminating backpropagation but risks significantly reducing prediction accuracy. To overcome these hindrances, the authors of [6] propose ADA-GP, an approach that speeds up DNN training without compromising accuracy. It employs a predictor model which predicts the gradients for the layers. This reduces dependence on backpropagation which has significant computational overhead. For the initial few epochs, the DNN is trained using traditional backpropagation which helps produce accurate gradient data. The gradient data is in turn used to train the predictor model. The training is alternated between the two approaches, traditional backpropagation (Phase BP) and the gradient calculated based on the predictor model (Phase GP). The balance between the two approaches is modified in a manner that improves performance while maintaining accuracy. Gradient prediction is managed on a large scale by employing tensor reorganization. Hardware extensions were also proposed to optimize implementation on DNN accelerators. This work aims to reproduce a section of the results obtained in [6] .

II. Related Works

Backpropagation [9] is the standard algorithm for computing gradients in deep neural network models. It has been extensively researched and is implemented as an optimized default in most deep learning libraries. However, backpropagation inherently depends on the completion of the forward pass, which prevents its parallelization with forward computation. Additionally, for large models, the gradient tensors can become substantial in size, making the computation and storage of exact gradients time-consuming and resource-intensive. These challenges have been recognized by researchers, and several approaches have been proposed to address these limitations. Some of these methods are discussed below.

Wang et al. [12] proposed methods to approximate gradients during CNN training, significantly reducing computation time while maintaining high accuracy. Their work highlighted the potential of gradient approximation in accelerating training for large-scale datasets and complex architectures. Building on similar objectives, Guan et al. [3] introduced XGrad, a framework designed to predict future weights before each mini-batch training. By incorporating these predictions into both the forward and backward passes, XGrad not only improved

convergence rates but also enhanced generalization, resulting in consistently higher accuracy across diverse neural network models.

In the domain of distributed training, He et al. [4] addressed the communication bottlenecks inherent in large-scale deep learning by proposing gradient compression techniques. Their method involved selecting the most significant gradient elements from layer residuals to minimize communication overhead, enabling faster training while maintaining model accuracy. This approach proved particularly effective in distributed environments, where communication costs often dominate the training process.

The article [1] suggests a technique that combines Nesterov's accelerated gradient descent with conventional gradient boosting to produce faster and more accurate model training for a range of applications. In order to potentially converge more quickly and with fewer trees than traditional gradient boosting, Accelerated Gradient Boosting iteratively constructs an ensemble of decision trees while including the acceleration mechanism. Two model sequences are maintained by the algorithm: one from the acceleration stages and one from the gradient steps.Together, these works demonstrate the growing focus on improving the efficiency of gradient computation and utilization in deep learning.

III. Problem Description

DNN training is a computationally intensive process with substantial challenges that increase as models and datasets grow in size. At the core of this challenge is the inherently sequential nature of the backward propagation algorithm, which is traditionally used to update a neural network's weights. Training involves passing data through each layer in a forward pass. The forward pass is followed by a backward pass, where the calculated gradients are propagated through the network. The model's weights are updated using the gradients. There is a dependency created due to the sequential nature of calculations due to which a layer's weights cannot be updated till all the previous layers have gone through forward and backward passes. This restricts the scope of parallelization and slows down training. This problem is exacerbated for deeper and more complex networks.

Furthermore, the increasing size of datasets makes this issue more problematic, since more time and resources are required to process and update the model. DNN training is hence not only time-consuming but also energy-intensive. High-performance hardware is required to meet the demands of DNN training, which leads to greater costs, especially for large amounts of data or cloud-based environments.

Hardware acceleration techniques, (such as GPUs, TPUs, or specialized DNN accelerators), have been deployed for faster DNN training. However, these do not address the issue of sequential backpropagation completely. While accelerating training is the primary focus, reducing time or resource overhead beyond a point is a detriment to model performance. The aim is to speed up training while maintaining the accuracy and stability of the model. Striking this balance is particularly critical for larger datasets since even a minor reduction in accuracy can have significant consequences. Overall, accelerating DNN training without reducing accuracy remains a major problem in machine learning. The sequential nature of backpropagation, combined with the growing complexity of models and datasets, creates a bottleneck that limits the scalability of training. Existing solutions have made strides in addressing this issue but often fall short in balancing training speed, model performance, and energy consumption.

IV. Proposed Solution

A. Overview

ADA-GP consists of three phases. To begin, there is a warmup phase, during which both the DNN and Predictor models are initialized. In this phase, the DNN model is trained using the input dataset, while the Predictor model is trained using the actual gradients from the DNN model. Once this phase concludes, ADA-GP alternates between Phase-BP and Phase-GP.

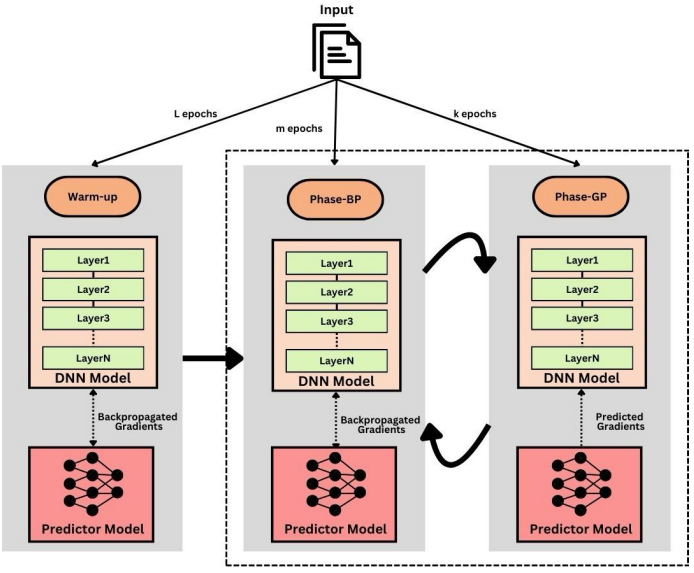

Fig. 1. ADA-GP Overview.

In Phase-BP, similar to the warmup phase, the DNN model is trained using the input dataset, and the Predictor model is trained using the actual gradients from the DNN model. In Phase-GP, however, the DNN model skips backpropagation and instead uses the gradients predicted by the Predictor model for its updates. Thus, ADA-GP alternates between learning the actual gradients and predicting them.

B. Warmup Phase

The warmup phase is used to allow the predictor model to "learn" to predict meaningful data. Initially, both the DNN and the predictor model have random weights and biases, which, if used for prediction, would result in high loss. Thus, during the warmup phase, the DNN model is trained using the input dataset, and the predictor model is trained using the activations and actual gradients captured from the DNN model.

C. Phase-BP

In Phase BP of ADA-GP, both the original and predictor models are trained using true gradients. While predicted gradients are calculated during forward propagation, they are not used to update the original model's weights. Instead, true gradients are used to calculate the predictor model's loss. During backward propagation, the true gradients update the original model's weights and train the predictor model concurrently. Figure 2 depicts this process, where $d_1, d_2, d_3, \ldots, d_N$ are the backpropagated gradients and $d'_1, d'_2, \ldots, d'_N$ are the predicted gradients.

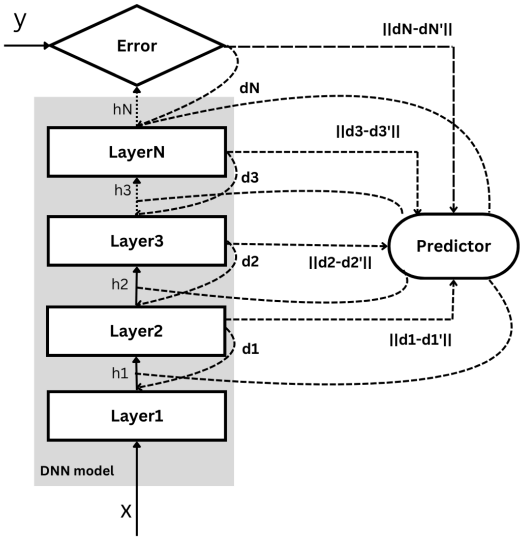

Fig. 2. Phase-BP

D. Phase-GP

In Phase-GP, during the DNN model training, the standard backpropagation step is skipped, and the original model is trained using the predicted gradients. To map the number of outputs predicted by the predictor model to the gradients of each layer in the DNN model, a *Tensor Reorganization* technique is utilized. Figure 3 depicts Phase-GP, where the DNN model uses the predicted gradients $d'_1, d'_2, \ldots, d'_N$ generated by the Predictor model.

E. Tensor Reorganization

ADA-GP addresses the challenge of predicting a large number of gradients with a compact predictor model using a novel tensor reorganization technique. This method rearranges the output activations of a DNN layer before forwarding them to the predictor model, achieving two goals: maintaining the predictor's compact size and ensuring higher-quality gradient predictions.

The issue arises because layers with a large number of weights demand significant memory and computational resources if a small predictor attempts to predict all gradients. The tensor reorganization technique solves this by averaging output activations across the batch, thereby reducing the tensor

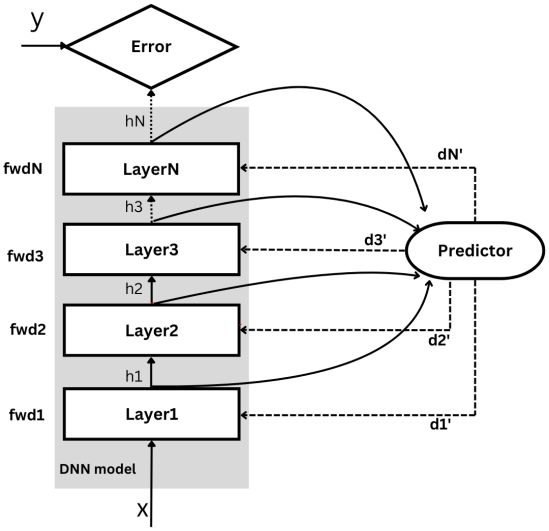

Fig. 3. Phase-GP

size and representing the combined effect of all samples. Each output channel is treated as a distinct training sample for the predictor. The reshaped tensor significantly reduces computational overhead, with the input size adjusted to match the predictor's compact design.

To generalize across layers in a DNN, the method incorporates pooling layers, a small convolutional layer, and a fully connected layer tailored to the largest DNN layer. Smaller layers use masking to skip unnecessary output operations. This approach ensures the predictor remains efficient while maintaining gradient prediction quality.

F. Tensor Re-Mapping

To address the inefficiencies in gradient prediction observed in the original ADA-GP paper, we propose a novel optimization termed *tensor re-mapping*. The original method predicted the maximum possible gradients across all layers and dynamically selected subsets as needed. While effective, this process is computationally expensive and does not scale well to larger networks.

Our Approach: We simplify this process by predicting a fixed number of gradients (10,000 in our implementation) and mapping these predictions to the required gradient tensor using repetition and padding:

- **Fixed Gradient Prediction:** During training and the GP phase, the predictor outputs a gradient vector of size 10,000, independent of the target tensor's size.
- **Mapping via Repetition:** For target tensors larger than 10,000 (e.g., 51,000), the predicted gradients are repeated cyclically to fill as much of the target tensor as possible. For instance:
  - The first 50,000 elements are filled with five repetitions of the predicted vector.

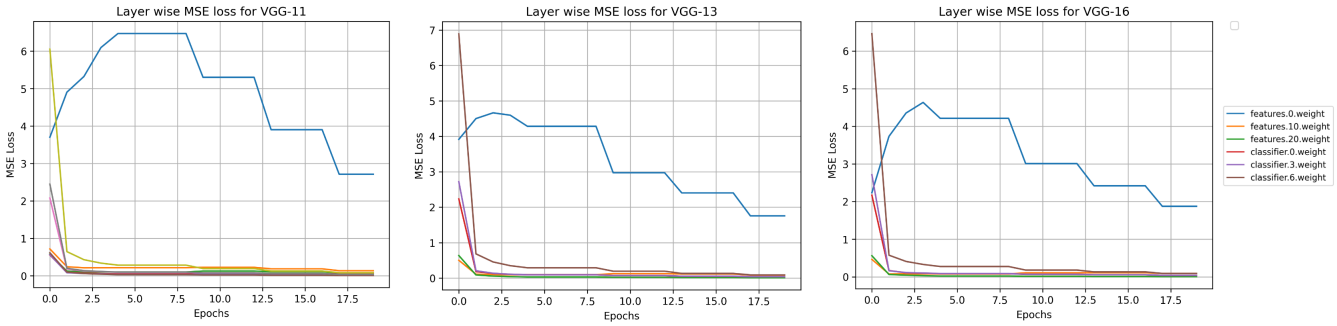

Fig. 4. Layer-wise Mean Squared Error (MSE) loss of the VGG models

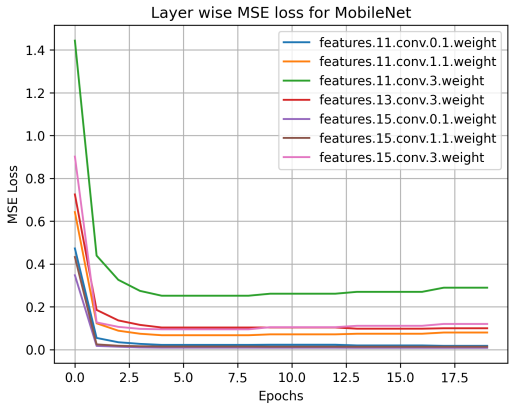

Fig. 5. Layer-wise MSE loss of MobileNet

- **Padding with Final Value:** Any remaining elements (e.g., the last 1,000 in a 51,000-long tensor) are filled with the final value of the predicted gradient vector.

Benefits of Tensor Re-Mapping:

- *Efficiency:* Eliminates the need for dynamic subset selection, significantly speeding up the process.
- *Scalability:* The method adapts to any target tensor size using repetition and padding.
- *Simplicity:* Simplifies the implementation pipeline without sacrificing performance.

This method ensures a practical balance between computational efficiency and gradient representation quality. Our experimental results show that tensor re-mapping achieves comparable accuracy to the original method while reducing runtime significantly.

G. Predictor Model Definition

The predictor model in this project is designed to predict gradient values efficiently for training deep neural networks (DNNs). It begins with an adaptive pooling layer (`nn.AdaptiveMaxPool2d`), which standardizes the spatial dimensions of the input tensor to a fixed size (`pool_output_size`), allowing the model to handle inputs of varying dimensions.

The core of the model consists of two convolutional layers (`nn.Conv2d`) followed by batch normalization (`nn.BatchNorm2d`) and ReLU activations. The first convolutional layer processes the input, mapping it to a set number of `hidden_channels`. The second convolutional layer further refines these features. Both layers are followed by batch normalization to stabilize training and ReLU activations to introduce non-linearity.

After feature extraction, the output is flattened and passed through a fully connected layer (`nn.Linear`) to map the features to a fixed output size. The model then computes the mean of the predicted gradients across the batch (`torch.mean`), providing a compact and stable gradient prediction for the target DNN while utilizing the *Tensor Reorganization* method proposed by the authors. Figure 6 illustrates the different layers in the Predictor model.

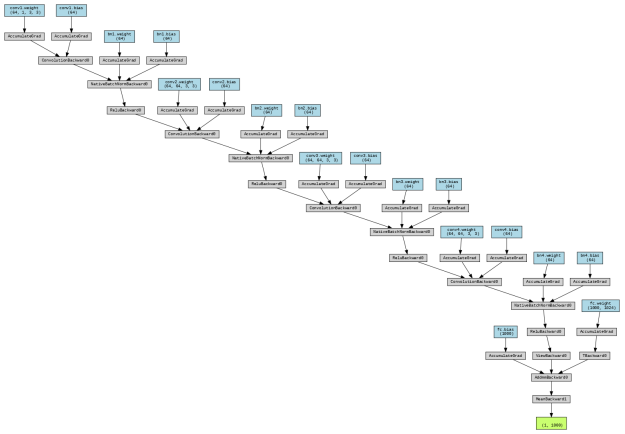

Fig. 6. Different layers in the Predictor model.

This architecture efficiently predicts gradients while maintaining a compact model size, ensuring it works well in gradient prediction tasks within larger frameworks like ADA-GP.

V. Implementation Setup

For the software implementation for ADA-GP, we have considered four networks - VGG11, VGG13, VGG16 [11],

and Mobilenet-V2 [10]. The networks were modified for the CIFAR-10 [7] dataset classification, and the final classifier layer was adjusted to output 10 classes. The training process utilizes an exhaustive configuration with a fixed seed of 42 for reproducibility across PyTorch, CUDA, NumPy, and Python's random module. The system employs a dual optimization strategy with an SGD optimizer for the main model with a learning rate of 0.001 and momentum 0.9. It also uses the Adam optimizer for the predictor with a learning rate of 0.0001. Both optimizers implement ReduceLROnPlateau schedulers that reduce the learning rate by 0.5 when loss plateaus, with patience of 3 epochs. The CIFAR-10 dataset undergoes preprocessing with ToTensor transformation and normalization using mean and standard deviation values of (0.5, 0.5, 0.5), loaded in batches of 32 samples with 2 worker processes and shuffling enabled. Training consists of 20 epochs with a 5-epoch warmup, utilizing a predictor model featuring a fixed output size of 10000 and including an adaptive max pooling layer with an output of 4x4 spatial dimensions, while it has 64 hidden channels in its convolutional layers. The architecture incorporates a PredictorModel with four convolutional layers, each followed by batch normalization and ReLU activation, and implements CrossEntropyLoss for the main model training and MSE loss for predictor training. The system includes comprehensive checkpoint functionality to save and load training states, thus enabling training resumption, and maintaining detailed loss and accuracy logging throughout.

VI. Evaluation Methods

A. Accuracy Analysis

1) Model Accuracy: We evaluate the accuracy of the proposed method across four different deep learning models using CIFAR-10 and compare it with the baseline backpropagation approach. The CIFAR-10 dataset is a widely used benchmark in computer vision, consisting of 60,000 color images categorized into 10 distinct classes, including airplanes, cats, and cars. Each image has a resolution of 32x32 pixels, and the dataset is evenly split into 50,000 training images and 10,000 testing images. CIFAR-10 is commonly used to assess the performance of machine learning models in image recognition and classification tasks.

Model	Without ADA-GP	With ADA-GP
VGG-11	78.93	77.04
VGG-13	79.76	77
VGG-16	80.09	77.65
MobileNet	82.22	79.12

TABLE I
ADA-GP's test accuracy compared to baseline (traditionally trained) models

2) Predictor Accuracy: To evaluate our predictor model's accuracy, we use MS (Mean Squared) Loss which is a commonly used metric in machine learning for model evaluation and optimization. The Mean Squared Loss (commonly referred to as Mean Squared Error or MSE) calculates the average squared difference between the true and predicted values. It is defined as:

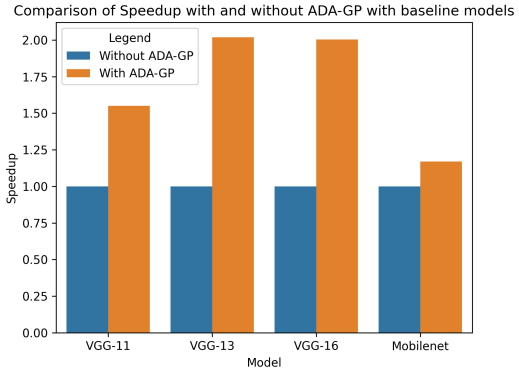

Fig. 7. Speedup comparison of DNN models with and without ADA-GP.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where:

- n: The number of data points in the dataset.
- y_i: The true value for the i-th data point.
- $\hat{y}_i$: The predicted value for the i-th data point.

MSE penalizes larger errors more heavily than smaller ones due to the squaring of the residuals $(y_i - \hat{y}_i)$. This makes it sensitive to outliers and effective in applications where minimizing large deviations is critical, such as regression tasks.

B. Speedup

We also calculate and compare the time taken to run the same number of epochs to train the DNN model using the traditional backpropagation method and the gradient prediction technique proposed in ADA-GP. In addition, we plot the speedup achieved for all four DNN models.

$$\text{Speedup of ADA-GP} = \frac{t_{Backpropagation}}{t_{ADA-GP}}$$

In our case, where we train the deep learning model using ADA-GP, the speedup formula is as follows:

$$\text{Speedup of ADA-GP} = \frac{t_o * e_{total}}{t_{bp} * e_{bp} + t_{gp} * e_{gp}}$$

where:

- t_o: Time per epoch without ADA-GP.
- t_{bp}: Time per epoch during which both the main model and the predictor are trained (Warm-Up and Phase-BP).
- t_{gp}: Time per epoch during which gradient prediction is used.
- e_{total}: Total number of training epochs without ADA-GP.
- e_{bp}: Total number of training epochs during which gradient prediction is not used.

- e_{gp}: Total number of training epochs during which gradient prediction is used.

VII. Results

As shown in Table II, ADA-GP achieves comparable test accuracies while providing a significant speedup in training. As shown in this table, on the CIFAR-10 dataset, ADA-GP effectively provided a speedup of up to 2x in the case of VGG-13 and an average speedup of 1.68x.

Model	Without ADA-GP	With ADA-GP	Speedup
VGG-11	5340	3432	1.55
VGG-13	7420	3660	2.02
VGG-16	8660	4320	2.004
MobileNet	10000	8480	1.17

TABLE II
MODEL PERFORMANCE COMPARISON WITH AND WITHOUT ADA-GP; TIME IS IN SECONDS.

Figures 4 and 5 provide an in-depth analysis of the MSE loss across different layers of each model throughout the training epochs. The figure shows that the MSE loss for each layer decreases during the Warm-Up and Phase-BP epochs but remains constant during Phase-GP. This aligns with the fact that the predictor model is trained only during the Warm-Up and Phase-BP phases.

VIII. Case Study: VGG11

A. Effect of batch size

The results in Table III show a clear trade-off between computational efficiency and model performance when training VGG11 with ADA-GP on CIFAR-10. As the batch size increases from 8 to 128, there is a significant decrease in training time, with total training time reducing from 12317.85s to 976.19s . However, this comes at the cost of model accuracy, which shows a strong negative correlation (-0.998) with batch size. The test accuracy gradually decreases from 78.81% at batch size 8 to 71.51% at batch size 128. The warm-up epoch time shows the most dramatic improvement, dropping from 1362.32s to 96.14s as batch size increases. Similarly, the Phase GP epoch time reduces from 118.27s to 17.26s. This suggests that while larger batch sizes offer substantial computational benefits, they may compromise the model's ability to generalize effectively, potentially due to less frequent weight updates and reduced exploration of the loss landscape.

Batch Size	Warm-up Epoch Time (s)	Phase GP Epoch Time (s)	Total time (s)	Test Accuracy (%)
8	1362.32	118.27	12317.85	78.81
16	686.13	63.75	6254.07	77.84
32	360.55	46.78	3445.76	77.04
64	185.32	25.18	1784.75	74.82
128	96.14	17.26	976.19	71.51

TABLE III
PERFORMANCE METRICS FOR DIFFERENT BATCH SIZES.

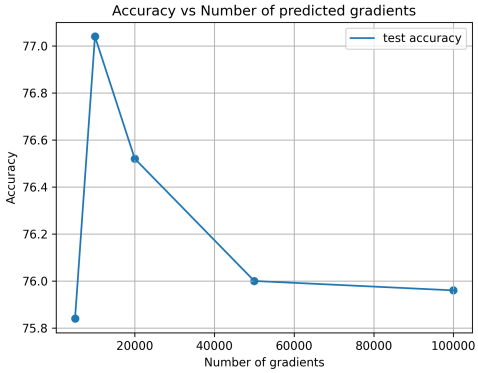

Fig. 8. Model accuracy vs The number of output gradients predicted.

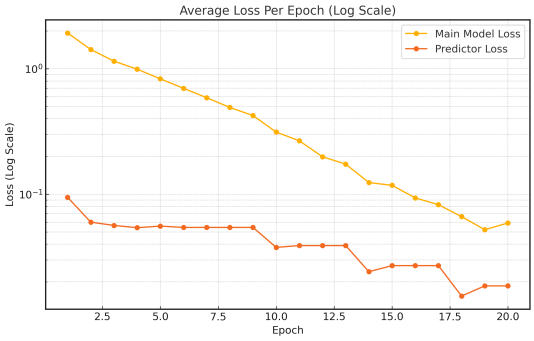

Fig. 9. Main model and Predictor model loss over the epochs.

B. Effect of predictor output size

In our approach, the model predicts a fixed-sized gradient output, which is interpolated to match the target output size. This differs from the original paper, where the maximum number of gradients across layers is predicted, and a subset is used. Selecting an appropriate gradient size is critical to balancing performance and efficiency. A large output size may lead to slower training, underfitting, and increased computational cost for the predictor, while a small output size can increase interpolation and concatenation overhead. To address this, we experimented with various output sizes to optimize the trade-offs between training time, efficiency, and model performance. Figure 8 guided our decision to set the fixed output size to 10,000, as it provided the optimal balance between performance and efficiency.

C. Loss trends

Figure 9 demonstrates the effective training of both the main model and predictor model across the ADA-GP framework phases: Warmup, Phase BP, and Phase GP. During Warmup, the predictor learns from backpropagated gradients without influencing the main model. In Phase BP, the predictor refines its gradient predictions while assisting the main model's optimization. Finally, in Phase GP, the predictor actively contributes to training the main model.

The consistent reduction in both main model and predictor losses illustrates the predictor's ability to support and accelerate the optimization of the main model, highlighting the synergy between the two.

IX. CONCLUSION

In this work, we implemented ADA-GP, a gradient-prediction approach to speed up DNN training while maintaining accuracy. We developed a predictor model that first learns from backpropagated gradients and then predicts the gradients. This technique eliminates the need for backpropagation, saving a significant amount of training time. Our predictor model uses a fixed output size and employs a tensor remapping technique to predict the large number of gradients required for different layers of the DNN model. We experimented with four DNN models: VGG-11, VGG-13, VGG-16, and MobileNet-V2. Our results indicate that ADA-GP can achieve an average speedup of 1.68x while maintaining high accuracy. To derive an optimal solution, we also varied different parameters such as batch size, predictor output size, and the number of epochs, and monitored the trends.

X. ACKNOWLEDGMENTS

We extend our gratitude to Professor Kim for providing us with the opportunity to work on this research project, which challenged us to think creatively and tackle problems we had never anticipated. We also express our sincere appreciation for our teaching assistant Sabuj Laskar, whose invaluable guidance helped us navigate obstacles throughout the project implementation and development process. Portions of this project were carried out with the advanced computing resources provided by Texas A&M High Performance Research Computing.

XI. DIVISION OF WORK

The project began with all members engaging in paper reading and discussions to build a foundational understanding of the research. We conducted a literature survey to identify related works. Initially, we divided the baseline models among the group, with each member being assigned a different model. ResNet-50 and ResNet-150 baseline models were implemented but were excluded from the final results. Due to the high computational resources required for implementing the predictor model for ResNet-50 and ResNet-150, we decided to use lighter networks, such as VGG11, VGG13, VGG16, and MobileNetV2. Weekly meetings were held to discuss progress on papers and code implementation. We updated the initial baseline codes on the GitHub repository. Then, we developed the initial PyTorch implementation of ADA-GP for smaller DNN models and divided the incremental changes among ourselves. Different approaches were assigned to each member until we achieved a final version. Once a working version of the ADA-GP code was completed, experiments were divided among the group, with each member working on a separate model or performance metric. Members also generated graphs and tables for their assigned experiments. For the final report, we initially divided the sections and wrote them individually. Later, we collectively made changes to create a comprehensive report. All members updated the GitHub repository with the code for their respective models. For the presentation slides, we collectively decided on the content to be included and created them accordingly. The project concluded with all members rehearsing and refining the final presentation.

XII. REPOSITORY AND PRESENTATION VIDEO

Link to the presentation
Link to the Github repository

REFERENCES

[1] G. Biau, B. Cadre, and L. Rouvière, "Accelerated gradient boosting," *arXiv preprint arXiv:1803.02042v1*, 2018.

[2] B. A. Erol, A. Majumdar, J. Lwowski, P. Benavidez, P. Rad, and M. Jamshidi, *Improved Deep Neural Network Object Tracking System for Applications in Home Robotics*. Cham: Springer International Publishing, 2018, pp. 369–395. [Online]. Available: https://doi.org/10.1007/978-3-319-89629-8_14

[3] L. Guan, D. Li, Y. Shi, and J. Meng, "Xgrad: Boosting gradient-based optimizers with weight prediction," *arXiv preprint arXiv:2305.18240*, 2023.

[4] Y. He, J. Li, O. Ruwase, and L. Zhong, "Accelerating distributed deep learning training with gradient compression," *arXiv preprint arXiv:1808.04357*, 2018.

[5] T. Iqbal and S. Qureshi, "The survey: Text generation models in deep learning," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 6, Part A, pp. 2515–2528, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157820303360

[6] V. Janfaza, S. Mandal, F. Mahmud, and A. Muzahid, "Ada-gp: Accelerating dnn training by adaptive gradient prediction," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1092–1105. [Online]. Available: https://doi.org/10.1145/3613424.3623779

[7] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[8] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[10] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] Z. Wang and S. H. Nelaturu, "Accelerated cnn training through gradient approximation," in *Proceedings of the 46th International Symposium on Computer Architecture*. IEEE, 2019.

XIII. Python Implementation

```python
# Define Tensor Reorganization
def tensor_reorganization(tensor):
    """
    Generalize tensor reorganization for both 2D, 3D, and 4D tensors.

    Args:
        tensor (torch.Tensor): Input tensor (e.g., activations, weights).
            - 4D tensor: (batch_size, channels, height, width).
            - 3D tensor: (batch_size, output_features, input_features).
            - 2D tensor: (output_features, input_features).

    Returns:
        torch.Tensor: Reorganized tensor in 4D format.
    """
    if tensor.dim() == 4:  # For convolutional weights (4D tensors)
      return tensor.permute(1, 0, 2, 3).mean(dim=1, keepdim=True)  # Result: (1, channels, height, width)

    elif tensor.dim() == 3:  # For FC layer weights with batch dimension (3D tensors)
        return tensor.mean(dim=0, keepdim=True).unsqueeze(0)  # Result: (1, 1, output_features,
    input_features)

    elif tensor.dim() == 2:  # For FC layer weights without batch dimension
        return tensor.unsqueeze(0).unsqueeze(0)  # Result: (1, 1, output_features, input_features)

    elif tensor.dim() == 1:  # For FC layer biases
        return tensor.unsqueeze(0).unsqueeze(0).unsqueeze(0)  # Result: (1, 1, 1, output_features)

    else:
        raise ValueError(f"Unsupported tensor dimension: {tensor.dim()}. Only 2D, 3D, and 4D tensors are
    supported.")
```

Code 1. Tensor reorganization logic

```python
# Define Predictor Model
class PredictorModel(nn.Module):
    def __init__(self, hidden_channels, pool_output_size, fixed_output_size):
        """
        Predictor Model for ADA-GP with fixed output size.

        Args:
            hidden_channels (int): Number of hidden channels for the convolutional layers.
            pool_output_size (tuple): Output size of the AdaptiveMaxPool2d layer (e.g., (4, 4)).
            fixed_output_size (int): Fixed size of the predicted gradient.
        """
        super(PredictorModel, self).__init__()
        self.pool = nn.AdaptiveMaxPool2d(pool_output_size)  # Dynamically reduce spatial dimensions
        self.conv1 = nn.Conv2d(1, hidden_channels, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(hidden_channels)
        self.conv2 = nn.Conv2d(hidden_channels, hidden_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(hidden_channels)
        self.conv3 = nn.Conv2d(hidden_channels, hidden_channels, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(hidden_channels)
        self.conv4 = nn.Conv2d(hidden_channels, hidden_channels, kernel_size=3, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(hidden_channels)

        # Dynamically calculate the flattened size after pool and conv2
        flattened_size = hidden_channels * pool_output_size[0] * pool_output_size[1]
        self.fc = nn.Linear(flattened_size, fixed_output_size)

    def forward(self, x, output_size):
        """
        Forward pass of the predictor.

        Args:
            x (torch.Tensor): Input tensor (from activations or reorganized gradients).
            output_size (torch.Size): Target size of the gradient tensor.

        Returns:
            torch.Tensor: Predicted gradients reshaped to match `output_size`.
        """
        x = self.pool(x)
        x = torch.relu(self.bn1(self.conv1(x)))
```

```
        x = torch.relu(self.bn2(self.conv2(x)))
        x = torch.relu(self.bn3(self.conv3(x)))
        x = torch.relu(self.bn4(self.conv4(x)))
        x = torch.flatten(x, 1)
        # print(x.shape)
        pred_chunk = self.fc(x)
        pred_chunk = torch.mean(pred_chunk, dim=0, keepdim=True)
        # print(pred_chunk.shape)
        return pred_chunk
```

Code 2. The predictor model used for gradient prediction

```
from torch.optim.lr_scheduler import ReduceLROnPlateau

loss_dictionary = {} # for saving loss values per layer

use_checkpoint = False
def train_adagp(model, predictor, dataloader, num_epochs=20, warmup_epochs=5, lr=0.001, fixed_output_size
    =1000, save_dir="./checkpoints", device="cuda", start_epoch=0):
    model.to(device)
    predictor.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer_model = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    optimizer_predictor = optim.Adam(predictor.parameters(), lr=lr * 0.1)

    scheduler_model = ReduceLROnPlateau(optimizer_model, mode='min', factor=0.5, patience=3, verbose=True)
    scheduler_predictor = ReduceLROnPlateau(optimizer_predictor, mode='min', factor=0.5, patience=3,
    verbose=True)


    if use_checkpoint:
        # Define the checkpoint path
        checkpoint_path = "/content/checkpoints/checkpoint_epoch_1_Warm-Up.pt"

        try:
            # Check if the checkpoint file exists
            if os.path.exists(checkpoint_path):
                # Load the checkpoint
                last_epoch, last_phase = load_checkpoint(
                    model, predictor, optimizer_model, optimizer_predictor, scheduler_model,
    scheduler_predictor, checkpoint_path
                )
                print(f"Resumed training from epoch {last_epoch + 1}, phase: {last_phase}")

                # Update training parameters if necessary
                start_epoch = last_epoch + 1  # Resume from the next epoch
            else:
                print("Checkpoint file does not exist. Starting training from scratch.")
                # Initialize training parameters
                start_epoch = 0
        except Exception as e:
            print(f"Error handling checkpoint: {e}")
            print("Starting training from scratch.")
            start_epoch = 0
    else:
        print("Checkpoint usage is disabled.")
        # Initialize training parameters
        start_epoch = 0

    activation_dict = {}
    ratio_sum = 4
    register_hooks(model, activation_dict)
    # counter = 0
    for epoch in range(start_epoch, num_epochs):
        phase = "Warm-Up" if epoch < warmup_epochs else "Phase-GP"

        if epoch == warmup_epochs:
          counter = 0

        if epoch > warmup_epochs:
          counter += 1
          if counter == ratio_sum:
            counter = 0
```

```
            phase = "Warm-Up"


        if phase == "Warm-Up":
            predictor.train()
        else:
            predictor.eval()

        epoch_loss = 0.0
        predictor_epoch_loss = 0.0
        correct_predictions = 0
        total_samples = 0
        start_time = time.time()

        for batch_idx, (inputs, labels) in enumerate(dataloader):
            inputs, labels = inputs.to(device), labels.to(device)

            # Warm-Up Phase
            if phase == "Warm-Up":
                optimizer_model.zero_grad()
                optimizer_predictor.zero_grad()

                outputs = model(inputs)
                loss = criterion(outputs, labels)
                epoch_loss += loss.item()

                loss.backward()
                optimizer_model.step()

                # print(activation_dict.keys())

                 # Train predictor
                for name, param in model.named_parameters():
                  # print(name)
                  param_key = '.'.join(name.split('.')[:-1])
                  # is_conv = 'conv' in name

                  if param.grad is not None and param_key in activation_dict:
                    # print(name)
                    activations = activation_dict[param_key]
                    # print(activations.shape)
                    if len(activations.shape) == 1:
                      continue
                    pred_grads = predict_and_fill(
                        predictor=predictor,
                        activations=activations,
                        output_size=param.grad.size(),
                        fixed_output_size=fixed_output_size
                    )
                    predictor_loss = nn.MSELoss()(pred_grads, param.grad.detach())
                    predictor_loss.backward()
                    predictor_epoch_loss += predictor_loss.item()

                    if name not in loss_dictionary:
                        loss_dictionary[name] = [0 for _ in range(start_epoch, num_epochs)]
                    else:
                        loss_dictionary[name][epoch] += predictor_loss.item()

                optimizer_predictor.step()

            elif phase == "Phase-GP":
                optimizer_model.zero_grad()
                layer_outputs = inputs

                # Perform a layer-by-layer forward pass
                for name, module in model.named_children():
                    # print(f"Layer name :{name}, layer_outputs.shape: {layer_outputs.shape}")
                    if isinstance(module, nn.Linear) or 'classifier' in name:
                      layer_outputs = layer_outputs.view(layer_outputs.size(0), -1)  # Flatten to (
    batch_size, features)

                    # print(f"Layer name :{name}, layer_outputs.shape: {layer_outputs.shape}")

                    layer_outputs = module(layer_outputs)  # Forward pass for the current layer
```

```
                    # print(layer_outputs.shape)


                    # Check if module has weights and requires gradient
                    if hasattr(module, "weight") and module.weight.requires_grad:
                        activations = tensor_reorganization(layer_outputs)
                        # print(activations.shape)
                        pred_grads = predict_and_fill(
                            predictor=predictor,
                            activations=activations,
                            output_size=module.weight.size(),
                            fixed_output_size=fixed_output_size
                        )
                        module.weight.grad = pred_grads  # Assign predicted gradients

                # Compute loss after full forward pass
                loss = criterion(layer_outputs, labels)
                loss.backward()
                optimizer_model.step()
                epoch_loss += loss.item()

            # Print batch-level stats
            if batch_idx % 10 == 0 or batch_idx == len(dataloader) - 1:
                print(f"Epoch [{epoch + 1}/{num_epochs}], Batch [{batch_idx + 1}/{len(dataloader)}], "
                      f"Loss: {loss.item():.4f}, Predictor Loss: {predictor_epoch_loss / (batch_idx + 1):.4
    f}")

            # Track accuracy
            if phase == "Warm-Up":
                _, predicted = torch.max(outputs, 1)
            else:
                # print(layer_outputs.shape)
                _, predicted = torch.max(layer_outputs, 1)

            # print(predicted.shape)
            correct_predictions += (predicted == labels).sum().item()
            total_samples += labels.size(0)

        # Epoch-level stats
        epoch_accuracy = 100.0 * correct_predictions / total_samples
        elapsed_time = time.time() - start_time
        print(f"Epoch [{epoch + 1}/{num_epochs}] | Phase: {phase} | Model Loss: {epoch_loss:.4f} | "
              f"Predictor Loss: {predictor_epoch_loss:.4f} | Accuracy: {epoch_accuracy:.2f}% | Time: {
    elapsed_time:.2f}s")

        scheduler_model.step(epoch_loss)
        if phase == "Warm-Up":
            scheduler_predictor.step(predictor_loss)
```

Code 3. The training loop which handles warm-up, phase-gp and phase-bp