Algorithmic Variants of Frank-Wolfe for Convolutional Neural Network Pruning: Computational Study

Hamza Shili[1] **Natasha Patnaik**[2] **Isabelle Ruble**[2]
Kathryn Jarjoura[2] **Daniel Suarez Aguirre**[2]

[1]Rice University, Computer Science Department

[2]Rice University, Computational Applied Math & Operations Research Department

Abstract

We design and implement a series of computational experiments aimed at comparing variants of the Frank-Wolfe (FW) algorithm for pruning large convolutional neural networks. This is motivated by the "Lottery Ticket Hypothesis", which suggests the existence of smaller sub-networks within larger pre-trained networks that perform comparatively well (if not better). Whilst most literature in this area focuses on Deep Neural Networks more generally, we specifically consider Convolutional Neural Networks for image classification tasks. With the MNIST dataset, we find that using a Frank-Wolfe approach with momentum for pruning results in a sparser model that achieves greater accuracy than both the original network and the sub-network obtained with simple backward elimination pruning.

1 Problem Statement and Notation

Convolutional neural networks are ideal models for image classification and related computer vision tasks. However, large models with many parameters pose serious challenges - both in terms of training time and memory. In this paper, we motivate and outline a series of computational experiments aimed at evaluating different variants of the Frank-Wolfe (FW) algorithm for neural network pruning, intending to find well-performing sparser models.

Notation - From Deep Neural Networks to Convolutional Neural Networks. To represent an arbitrary neural network with N neurons, we follow the notation used in [13] and [14]. Let $\mathbf{x} \in \mathbb{R}^d$ denote the input vector and $\theta_\mathbf{i}$ for $i \in [N] = \{1, ..., N\}$ denote the parameters/ weights associated with the i-th neuron. With these inputs, the i-th neuron can be represented as a function $\sigma(\mathbf{x}, \theta_\mathbf{i})$.

Although this function $\sigma(\cdot, \theta_i)$ can take on general forms, [13] and [14] consider the specific case of a two-layer network for simpler analysis (a representative example of more complex architectures). In the two-layer case, we can write $\sigma(\mathbf{x}, \theta_\mathbf{i}) = b_i \cdot \sigma_+(\mathbf{a_i}^T \mathbf{x})$, where $\theta_\mathbf{i} = [b_i \ \ \mathbf{a_i}]$ denotes the concatenation of the second layer weight, b_i, and the vector of first layer weights, $\mathbf{a_i} \in \mathbf{R}^d$, respectively. The function $\sigma_+(\cdot)$ represents the activation function.

Accordingly, we use the following notation for the output of a neural network containing N neurons:

$$f_{[N]}(\mathbf{x}, \Theta) = \frac{1}{N} \sum_{i=1}^{N} \sigma(\mathbf{x}; \theta_\mathbf{i}) \tag{1}$$

Preprint. Under review.

Given a dataset $D = \{\mathbf{x}^{(j)}, y^{(j)}\}_{i=1}^{m}$ consisting of m observations, each with attributes $\mathbf{x}^{(j)} \in \mathbb{R}^d$ and desired output $y^{(j)} \in \mathbb{R}$, we can train the neural network to discover the set of weights Θ that minimizes the following loss function:

$$\mathcal{L}\big[f_{[N]}(\cdot, \Theta)\big] = \frac{1}{2}\sum_{j=1}^{m}\big(f_{[N]}(\mathbf{x}^{(j)}, \Theta) - y^{(j)}\big)^2 \tag{2}$$

While existing analysis in this area uses the representative example of a two-layer network, Convolutional Neural Networks have a more complex architecture, which includes convolutional layers and pooling layers. To extract local features, convolutional layers apply filters that learn to detect specific patterns. The output of a convolutional layer l, denoted $\mathbf{h}^{(l)}$, is obtained by convolving the input feature maps $\mathbf{h}^{(l-1)}$ with learnable filters $\mathbf{W}^{(l)}$ and applying an activation function, ψ, that introduces non-linearity (such as ReLU):

$$\mathbf{h}^{(l)} = \psi\big(\mathbf{W}^{(l)} * \mathbf{h}^{(l-1)}\big) \tag{3}$$

Such convolutional layers are typically succeeded by pooling layers, which reduce spatial dimensions of the feature maps to create a more compact representation while retaining essential information. For instance, max pooling will simply select the maximum value within a given local region:

$$\mathbf{h}_{\text{pool}}^{(l)} = \text{Max}\big\{\mathbf{h}^{(l)}\big\} \tag{4}$$

Three Stage Approach for Neural Network Sparsification The classic Stochastic Gradient Descent (SGD) algorithm is commonly used for Neural Network training, with the model's weights $\Theta = \{\theta_1, ..., \theta_N\}$ typically being unconstrained [11]. As outlined in [8], [13], and [14], the computational cost and memory requirements for training large-scale, dense models as N increases necessitates a more practical three-step procedure for finding smaller sub-networks: *(i) Pretraining* - minimize (1) by performing SGD over Θ, *(ii) Pruning* - finding a sub-network from the original model which performs well, and *(iii) Retraining* - performing SGD again to maintain desired accuracy, either using the newer model's weights as the initial parameters (as in [14]) or starting with a new random parametrization (as in [8]).

In particular, the pruning stage is concerned with finding a subset of neurons $S \subset [N]$ which minimizes the loss of the newly defined sub-network: $f_S(\mathbf{x}, \Theta) = {}^1\!/_{|S|}\sum_{i \in S}\sigma(\mathbf{x}; \theta_\mathbf{i})$. In particular, we would like this sub-network's accuracy to be comparable to the original dense model:

$$\mathcal{L}\big[f_{[S]}\big] \geq \mathcal{L}\big[f_{[N]}\big] - \delta \text{ for some } \delta \in \mathbb{R}^+ \tag{5}$$

This allows us to find sub-networks that perform relatively well (as compared to the original dense model) [13]. Furthermore, in [14], they find that pruning dense models guarantees finding more accurate sparse networks than directly training smaller models can, which is in line with the Lottery Ticket Hypothesis [5].

2 Background

Lottery Ticket Hypothesis The Lottery Ticket Hypothesis (LTH), proposed by Frankle and Carbin in 2019, suggests that within randomly-initialized, dense neural networks, there exist sparse subnetworks, or "winning tickets," capable of achieving test accuracy comparable to the original network in a similar number of iterations. These winning tickets benefit from fortuitous initializations that facilitate effective training. Justified by the LTH, a structured three-stage approach to neural network sparsification emerges.

Firstly, in the identification stage, pruning techniques are employed to systematically remove connections from the network, thereby identifying these winning tickets. Pruning enables significant reductions in model size by removing unnecessary parameters while preserving the network's essential functionality. Frankle and Carbin's reserach consistently found that winning tickets are less than $10 - 20\%$ of the size of several fully-connected and convolutional architectures for MNIST and CIFAR10 datasets [5].

Secondly, in the optimization stage, winning tickets are leveraged to design more efficient training schemes. By focusing on training these sparse subnetworks from the start, the aim is to accelerate convergence and improve overall training dynamics. This stage involves devising novel training algorithms and optimization strategies tailored to the unique properties of winning tickets, enabling faster learning and improved generalization.

Lastly, in the refinement stage, iterative pruning techniques are employed to further enhance the performance of the identified winning tickets. Iterative pruning outperforms one-shot pruning in identifying winning tickets by repeatedly training, pruning, and refining the network over multiple rounds. This iterative process results in networks that learn faster and achieve higher test accuracy, ultimately leading to more efficient and effective deep learning models [5].

These implications suggest that certain sparse architectures with fortuitous initializations possess inherently better training properties and offer opportunities for efficient compression, which involves reducing the size of neural networks while maintaining performance. This structured approach not only advances the field of neural network pruning and optimization but also lays the foundation for more efficient and effective deep learning models.

Suitable Feasible Regions Frank-Wolfe methods can offer computational efficiency by minimizing a linear objective over the feasible set without needing costly projections [12]. This advantage extends to scenarios with polyhedral and nuclear norm ball constraints, simplifying the linear subproblems and streamlining solutions. Additionally, their projection-free nature makes them popular, particularly in machine learning, where they surpass methods requiring projection at each iteration. As such, Frank-Wolfe methods are appropriate when the optimization problem's constraints define convex, differentiable, and compact feasible regions[6]. Convexity ensures that the feasible region forms a convex set, enabling efficient exploration of the solution space. Differentiability allows for the use of gradient information to guide the optimization process towards the optimal solution. Compactness ensures that the feasible region is bounded, facilitating convergence towards a global optimum.

These essential characteristics of feasible regions crucial for the effective implementation of Frank-Wolfe algorithms, particularly within the realm of large-scale optimization, has been studied as the effectiveness of the algorithm was shown to be intricately tied to the convexity and compactness of the feasible region, coupled with the ability to solve linear optimization subproblems efficiently within this region [3]. Specifically, Frank-Wolfe methods excel in scenarios where constraints define bounded convex regions, enabling swift exploration of the solution space without necessitating costly projection operations. This alignment fits the needs of neural network pruning tasks, where a loss function subject to constraints on network sparsity is minimized. The convexity of the feasible region also ensures a well-defined optimization problem, an essential factor in steering the iterative pruning process efficiently towards an optimal solution. Additionally, the compactness property guarantees convergence towards a global optimum, ensuring the algorithm's resilience in traversing the solution landscape. Furthermore, the differentiability of the feasible region enables the computation of gradients, essential for guiding the pruning process towards optimal sparse architectures efficiently. Compactness ensures a bounded search space, preventing the optimization process from straying towards infeasible solutions.

By satisfying these properties, the optimization problem in neural network pruning mirrors the characteristics of problems well-suited for Frank-Wolfe methods. The iterative nature of Frank-Wolfe-style approaches, where solutions are iteratively refined based on linear approximations of the objective function within the feasible region, aligns with the iterative pruning process. Moreover, the reliance on gradients to navigate the search for optimal sparse architectures underscores the significance of differentiability, a core property of Frank-Wolfe methods.

3 Convolutional Neural Network Pre-training and Pruning

Convolutional Neural Networks Convolutional Neural Networks (CNNs) are a class of neural networks primarily utilized for image classification. The defining feature of CNNs is the convolution

operation, which involves sliding filters over the input data to extract and learn important features such as edges and textures. Key elements of CNNs include:

- **Convolutional Layers**: Detect features by applying various filters to the input.
- **Activation Functions**: Introduce non-linearity into the model, enabling it to learn complex patterns. Eg. ReLU
- **Pooling Layers**: Reduce the dimensionality of each feature map while retaining the most important information.
- **Fully Connected Layers**: After the convolutional and pooling layers extract and reduce features, these layers classify the input based on the detected features by outputting probabilities over the classes.

Resource Constrained Neural Networks and Pruning Paradigm While machine learning and deep learning applications are increasing at an exorbitant rate, their real-world implementation is delayed by their high computational requirements in the form of cutting-edge hardware, large memory, and high energy use. [2], [9] With this context, resource-constrained solutions arise. Henceforth, we will show how pruning can help to improve efficiency in training and inference.

Neural network pruning consists of simplifying a neural network by setting some weights to zero according to some rule. For example, a basic pruning rule may consist of removing weights smaller than a certain threshold. Overall, the goal of running is to reduce model size and complexity without compromising performance. In general, the pruning process is the following:

- **Pre-Training**: Train NN to get a baseline model.
- **Pruning**: Introduce non-linearity into the model, enabling it to learn complex patterns.
- **Re-Training**: Reduce the dimensionality of each feature map while retaining the most important information.

Note that pruning and re-training may be done in an iterable manner; this is known as the tunning stage. Also, under some methods, re-training may not be necessary given the needs of the user

Within the perspective of resource-constrained neural networks, a pruned network is smaller than the baseline model and this achieves better efficiency for inference tasks. From the angle of training, pruning may also be beneficial because rather than training a large network to a full extent, the large network is trained only up to a point from which a sufficiently good subnetwork is identified. Therefore, even when considering pre-training and re-training, pruning may lead to using fewer resources as, after pruning, the rest of training is done in a smaller network.

Finally, note that this framework also applies in the case of CNNs, which are the base of our computational study.

Frank-Wolfe for Pruning To find well-performing smaller sub-networks within dense models that are pre-trained via stochastic GD, Frank-Wolfe-style approaches can be used [13] [14], where the desired sparsity level is treated as a constraint. Thus, even though Neural Network training itself is a non-convex optimization problem, the pruning stage can be reformulated as a constrained convex optimization problem, with Frank-Wolfe-style approaches being used [13].

Frank-Wolfe in Machine Learning Context The classic FW algorithm (and related projection-free methods) provides an alternative to projected Gradient Descent for convex-constrained optimization problems. Its low per-iteration complexity and suitability for complicated constraints makes it very effective in the context of large-scale machine-learning problems [10].

FW replaces the projection step of projected GD with a linear minimization sub-problem that can be described as follows:

We form the next iterate by taking a convex combination between (i) the previous iterate and (ii) an extreme point of the feasible region that best approximates the gradient direction. [10] This ensures we retain feasibility within each iteration, without using a projection step. Thus, FW is particularly advantageous when the traditional projection step in projected GD is computationally expensive [6]. See the pseudocode description for the classic deterministic variant below:

Algorithm 1 Classic (Deterministic) Frank-Wolfe Algorithm [4], [6]

Input: $x_0 \in \mathcal{C} :=$ initial starting point; $T :=$ number of iterations
for $t = 0, 1, ..., T$ **do**
  $v_t \leftarrow \text{argmin}_{x_t \in \mathcal{C}} \langle x_t, \nabla f(x_t) \rangle$ ▷ Approximate gradient direction, maintaining feasibility
  $\gamma \leftarrow \frac{2}{t+2}$
  $x_{t+1} \leftarrow (1-\gamma)x_t + \gamma v_t$ ▷ Convex combination: previous iterate and v_t
end for

In Algorithm (1), we can also substitute the linear minimization $v_t = \text{argmin}_{v_t \in \mathcal{C}} \langle x_t, \nabla f(x_t) \rangle$ with an approximate approach, as opposed to solving for the minimizer exactly [6]. Another simple modification is to perform a line search within each iteration t for the optimal γ required to find the best next iterate on the line segment connecting x_t and v_t [6].

4 Pruning Algorithms

Simple Pruning The first algorithm takes a naive approach to pruning. First, all of the weights of the model are retrieved and sorted based on magnitude. A certain percentage of the weights with the lowest magnitude are set to zero based on a user-defined *pruning percentage*.

Algorithm 2 Simple Pruning

Input: $M \in \mathcal{M}$, a neural network model; $p \in [0, 1]$, pruning percentage; D, training dataset; E, epochs for retraining
$W \leftarrow$ weights of M ▷ Retrieve model weights
$n \leftarrow |W|$ ▷ Total number of weights
$n_p \leftarrow \text{round}(p \cdot n)$ ▷ Compute number of weights to prune
$W_f \leftarrow$ flattened W ▷ Concatenate all weights into a vector
idxs $\leftarrow$ indices sorted by $|W_f|$
$I_p \leftarrow$ first n_p elements of idxs
for $i \in I_p$ **do**
  $W_f[i] \leftarrow 0$ ▷ Set pruned weights to zero
end for
Update W of M with W_f
Set optimizer, loss, and metrics for M
Retrain M on D for E epochs
return M

Frank-Wolfe Pruning The second algorithm considers a randomly chosen subsample of the weights and takes gradient information into account when selecting weights to prune. Rather than pruning a fixed percentage of the weights, the pruning mask is applied based on a user-defined *target sparsity level*. This method takes into account cases in which smaller weights are actually more integral to the structure of the CNN and thus is more flexible than the simple pruning algorithm.

Algorithm 3 Frank-Wolfe Pruning

Input: $M \in \mathcal{M}$, neural network model; D, training dataset; s, target sparsity level; N, number of pruning iterations; S, subsample size; E, epochs for retraining

Initialize optimizer (SGD)
for $i = 1$ to N **do**
  $G \leftarrow$ zeros (shape of M's trainable weights)
  $subsample \leftarrow$ sample S instances from D
  for each (x, y) in $subsample$ **do**
    $G \leftarrow G + \nabla \text{loss}(M, x, y, \text{optimizer})$
  end for
  if $i \mod \left(\frac{N}{2}\right) = 0$ **then**
    Apply pruning to M based on G and sparsity s
  end if
  Retrain M using D for E epochs
end for
return M

Frank-Wolfe Pruning + Momentum The third algorithm has two main modifications compared to the basic frank-wolfe method: dynamic sparsity and momentum. The *target sparsity level* is determined by the user. However, the sparsity of the model is increased iteratively which potentially minimizes large decreases in model performance at each iteration. Additionally, a user-defined *momentum* vector is applied to the gradients in order to further differentiate the weights and stabilize the algorithm.

Algorithm 4 Frank-Wolfe Pruning + Momentum

Input: $M \in \mathcal{M}$, neural network model; D, training dataset; s_{init}, initial sparsity; s_{final}, final sparsity; m, momentum parameter; N, number of iterations; E, fine-tuning epochs; S, subsample size; Optimizer

$\Delta s \leftarrow \frac{s_{\text{final}} - s_{\text{init}}}{N}$
$s \leftarrow s_{\text{init}}$
for $i = 1$ to N **do**
  $G \leftarrow$ zeros (shape of M's trainable weights)
  $subsample \leftarrow$ sample S instances from D
  for each (x, y) in $subsample$ **do**
    $G \leftarrow G + \nabla \text{loss}(M, x, y, \text{Optimizer})$
  end for
  $G \leftarrow m \cdot G$ ▷ Apply momentum to gradients
  Apply pruning to M based on G and sparsity s
  if $E > 0$ **then**
    Retrain M using D for E epochs
  end if
  $s \leftarrow \min(s + \Delta s, s_{\text{final}})$
end for
return M

5 Numerical Experiments

Datasets For our computational study, we consider image classification problems. Here we used the MNIST [7] dataset as our baseline. This dataset contains handwritten digits, with 60,000 examples in the training set and 10,000 examples on the test set. The digits are centered in 28x28 grayscale images. The only pre-processing applied to the data is normalization of the grayscale by dividing the value of each pixel by 255, ensuring that the pixels used as input for the model are between 0 and 1.

Network Architecture For our numerical analysis, we employ a CNN designed for the classification of 28x28 images into 10 categories, as is the case for the MNIST dataset. The architecture is detailed in Table 1.

Layer Type	Configuration	Purpose
Convolutional	32 filters of size 3x3, ReLU activation	Captures basic features from input images, introduces non-linearity.
Max Pooling	Pool size of 2x2	Reduces spatial dimensions, making the detection of features somewhat invariant to scale and orientation changes.
Convolutional	64 filters of size 3x3, ReLU activation	Increases the complexity of the model to capture more detailed features.
Max Pooling	Pool size of 2x2	Further reduces spatial dimensions, focusing on the most important features.
Flatten	—	Transforms 3D feature maps into 1D feature vectors.
Dropout	Dropout rate of 0.5	Prevents overfitting by randomly setting input units to 0 during training.
Dense	10 units, Softmax activation	Outputs the probability distribution over the ten classes.

Table 1: CNN Architecture for MNIST Classification

Results and Analysis The effectiveness of different Frank-Wolfe pruning methods is evaluated in terms of accuracy, loss, computational efficiency, and complexity reduction, as shown in Figures 1, 2, and 3.

Model Accuracy and Loss: As depicted in Figure 1, model accuracy improves with increased pretraining epochs across all methods, which is expected due to the enhanced learning from more extensive training data. The Frank-Wolfe (FW) method with momentum consistently outperforms other methods in both accuracy and loss metrics, indicating its robustness. Even with limited pretraining, the FW + momentum method maintains higher accuracy than the base model, highlighting its effectiveness in constrained environments. Notably, the simple pruning method often yields accuracy and loss metrics that are comparable or inferior to those of the base model, suggesting that this approach may undermine performance.

Computational Efficiency: Figure 2 illustrates the inference time for each method. The FW methods achieve lower inference times compared to the simple pruning method, reinforcing their suitability for resource-limited settings. This efficiency is crucial for applications requiring real-time processing.

Model Complexity: The comparison of non-zero parameter percentages in Figure 3 exhibits the FW methods' superior ability to reduce model complexity, particularly in scenarios with fewer pretraining epochs. The FW + momentum method consistently shows a lower percentage of non-zero parameters compared to the basic FW method, especially before extensive pretraining, making it an advantageous choice in resource-constrained conditions.

6 Conclusions

Conclusions This computational study evaluated three neural network pruning methods for models with limited pretraining, which is a common scenario within resource-constrained environments. Interestingly, the results indicate that the simple pruning technique is ineffective in improving the model and is detrimental to model performance in some cases. Of the methods studied, the Frank-Wolfe (FW) + momentum technique proved the most effective. It demonstrated superior performance across all four metrics of model accuracy, loss, computational efficiency, and complexity reduction. While the basic FW method also achieved better performance than the base model, the results indicate that incorporating momentum into the FW pruning approach can significantly benefit the model, especially in cases where computational resources are limited.

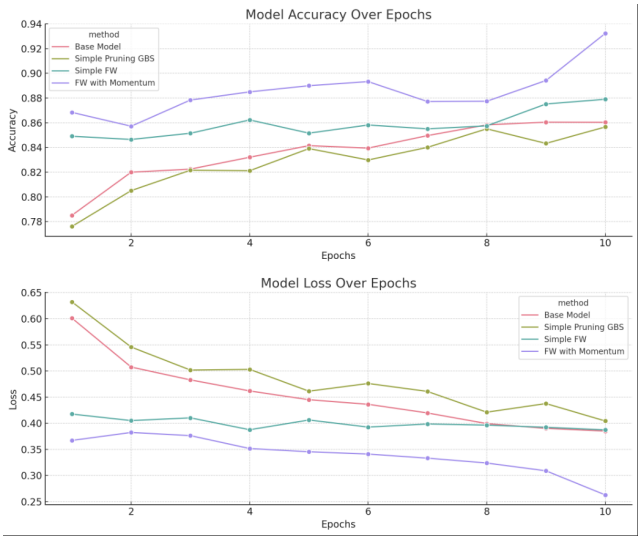

Figure 1: Accuracy and loss of neural network models pruned using different methods, across multiple pretraining epochs. This figure highlights the trade-offs between accuracy, loss, and pretraining duration.

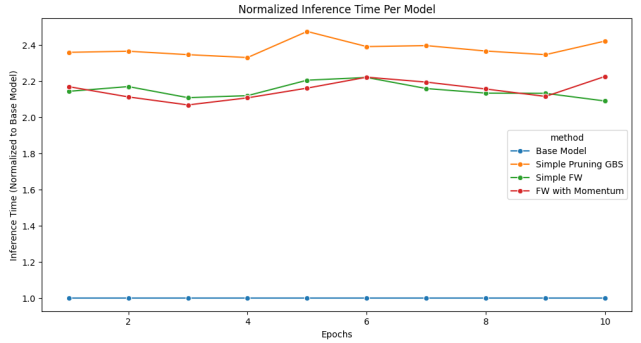

Figure 2: Normalized inference time for pruned neural networks, illustrating the computational efficiency per pruning method with varying pretraining epochs. This figure evaluates the trade-off between pruning efficiency and computational performance.

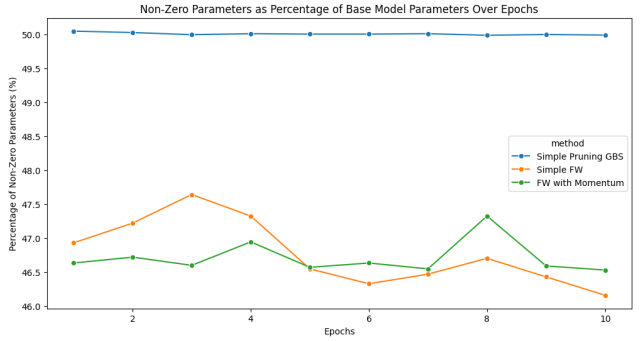

Figure 3: Percentage of non-zero parameters in neural networks post-pruning, relative to the unpruned base model, across different levels of pretraining. This graph demonstrates the effectiveness of each pruning method in reducing model complexity.

7 Further Research

Generalizability and Comparison with Other Pruning Methods Despite the positive results attained with our numerical experiments, it is still to be seen if these results can be generalized to other settings. With access to more computational resources, the performance of FW pruning may be evaluated by using optimizers other than Stochastic Gradient Descent (SGD), such as ADAM. Moreover, to further test the generalizability of our FW pruning methods, their performance could be tested on established architectures such as ResNet, AlexNet, or VGG. Likewise, one could also test the generalizability from varying the dataset by using datasets such as Fashion-MNIST, CIFAR-10, or ImageNet. Overall, despite promising results, more computational resources are needed to test the performance of our pruning methods in a general setting.

Moreover, performing these tests would allow us to compare performance against other pruning methods. To do this, we should suggest using a wide benchmark, such as the one available with ShrinkBench [1], to allow a systematic review of the models' performance.

Structured Pruning The pruning paradigm we considered before, which masks certain weights as zeros, is known as unstructured pruning. On the other hand, we have structured pruning, which removes entire units, such as neurons, layers, or filters from the NN/CNN. That is, structured pruning changes the architecture of the network rather than masking weights. Combining the algorithms developed in this paper with structured pruning may lead to even stronger numerical results as model size and complexity could be reduced to a greater extent.

Greedy Forward Selection [14]. Another pruning methodology that could be considered for future research is greedy forward selection. Instead of traditional pruning methods that use backward elimination to remove redundant and Byzantine neurons from the larger network, greedy forward selection begins with an empty model, and sequentially adds neurons from the original network that result in the greatest immediate decrease in the loss function [14]. The main advantage of this scheme is that there is no need to prune after pre-training, as a subnetwork is found while training. Hence, this approach may be of particular interest in resource-constrained environments.

References

[1] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[2] Ting-Wu Chin, Cha Zhang, and Diana Marculescu. Layer-compensated pruning for resource-constrained convolutional neural networks, 2018.

[3] Lijun Ding and Madeleine Udell. Frank-wolfe style algorithms for large scale optimization, 2018.

[4] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[6] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 427–435, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[8] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018.

[9] Mariusz Pietrołaj and Marek Blok. Resource constrained neural network training, Jan 2024.

[10] Sebastian Pokutta. The frank-wolfe algorithm: A short introduction. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 126:3–35, 2024.

[11] Sebastian Pokutta, Christoph Spiegel, and Max Zimmer. Deep neural network training with frank-wolfe. 2020.

[12] Haoyue Wang, Haihao Lu, and Rahul Mazumder. Frank-wolfe methods with an unbounded feasible region and applications to structured learning, 2021.

[13] Cameron R. Wolfe, Fangshuo Liao, Qihan Wang, Junhyung Lyle Kim, and Anastasios Kyrillidis. How much pre-training is enough to discover a good subnetwork? 2023.

[14] Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. 2020.

Continual learning

Afroditi Kolomvaki
Department of Computer Science, Rice University
ak203@rice.edu

Abstract

In order to deal with the dynamics of the real world, an intelligent system should be able to improve over time by continuously acquiring more and more knowledge through data, regardless of the data distribution but most importantly, without forgetting what it has learnt thus far. Streaming learning, i.e. learning from a single data point at a time, has the potential to facilitate real-time adaptation to newly available data. Many continual learning approaches either freeze a significant amount of the model's parameters or depend on a great amount of data for pre-training so as to initialize the parameters of the model before streaming begins. However, these practices can have a detrimental effect to the model's performance. In this project, we have studied Cold Start Streaming Learning (CSSL), a streaming learning system that overcomes the aforementioned limitations as it updates all the parameters of the model in an end-to-end fashion, leveraging its representational power to the maximum possible degree. CSSL also does not require pre-training in order to preform well, as it can begin streaming even from random initialization without missing out on performance, which constitutes an important asset given how expensive offline base initialization procedures can be.

As mentioned earlier, the goal of continual learning is to constantly adjust to new data in order to acquire new knowledge. This differs from traditional problem settings, where the objective is to reach a fixed point of convergence. A fundamental challenge in continual learning is determining what information should be retained from previous data to prevent catastrophic forgetting (5), and what information needs to be updated to maintain alignment with the objective function. A neural network can achieve this by employing a step-size vector, which controls the extent to which gradient samples influence the adjustment of network weights, suggesting that adaptive learning rate schedules could play a crucial role in continual learning.

1 Introduction

1.1 Background

"Online learning" is a term we use broadly to refer to a wide array of techniques where the training process is carried out in a sequential fashion and data is fed into the model one sample or a small batch at a time, as opposed to the traditional offline learning approach where the entire dataset is available during training. Due to the benefits that real-time adaptation has, online learning is a topic that has gained popularity and different variations of it have been studied. One of them is batch-incremental learning (12; 2), which involves training the model with batches of data in a sequential manner. These batches are typically drawn from distinct sets of classes or tasks within a dataset. However, this approach has a significant drawback: it is necessary to wait for a substantial amount of data to accumulate before the model can be updated through a computationally intensive and time-consuming offline training process on the newly acquired data. This latency hinders real-time model updates, as the system cannot adapt to new information as soon as it becomes available. As a result, batch-incremental learning may not be suitable for applications that require immediate model adaptation

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

to rapidly changing environments or data streams. To minimize delay, CSSL utilizes a streaming learning approach where each data sample is processed only once and the entire dataset is learned in one iteration (number of epochs is 1) (7). This method involves brief, online adjustments, ensuring that learning happens instantaneously. Moreover, streaming learning methods have the flexibility to handle groups of data rather than just individual items (14). In contrast, batch-incremental learning approaches often experience significant performance drops when dealing with smaller batches of data (8). Therefore, the streaming learning framework, which has been investigated for its application in deep neural networks in recent studies (7; 8; 9), is versatile and offers the potential for efficient, low-overhead updates to deep networks with new data.

2 Methodology

The model CSSL uses (e.g. ResNet18), can begin streaming by either utilizing a pre-trained set of parameters or a random initialization. To mitigate the issue of catastrophic forgetting, this approach leverages a replay buffer $\mathcal{R}$ paired with sophisticated data augmentation. At each iteration, the algorithm processes a new data example $D_t = (x_t, y_t)$ retrieved from the dataset D_t, and combines it with $\mathcal{B}$ randomly selected samples from the replay buffer. It then performs a stochastic gradient descent (SGD) update using this combined data, and stores the new example in the buffer for future reuse. At this point, we should note that the system may see a data point only once throughout its lifetime despite the fact that it was stored in the replay buffer after it became available for the first time. This happens because we sample $\mathcal{B}$ samples from a replay buffer with capacity $\mathcal{C}$, where $\mathcal{C} > \mathcal{B}$ and also at some point old data points have to be discarded from the replay buffer once it has reached its maximum capacity so that the new data points can be stored.

2.1 Definition of Streaming Learning

Streaming learning involves processing a continuous flow of data denoted as $\mathcal{D} = \{x_t, y_t\}$ from $t = 1, ..., n$ and the training process guidelines are the following:

- Each data example is unique and appears only once in $\mathcal{D}$.
- The sequence of data in $\mathcal{D}$ is arbitrary and may not follow an independent and identically distributed (iid) pattern.
- The model's performance can be evaluated at any point in time during the streaming process.

These requirements don't presuppose anything about the model's initial state before streaming starts. However, traditional approaches typically require an offline base initialization step before streaming begins, whereas CSSL can begin the streaming process directly from either random or pre-trained parameters, giving it the advantage of initiating streaming without needing to see any data from the stream first.

2.2 Problem Setting

Here we address the problem of image classification using streaming, building on previous studies (8). The experiments usually involve class-incremental streaming where samples from each unique class in the dataset are fed to the model sequentially. Future work involves integrating adaptive learning rates into the model, as explained in section 4, and study its performance in different streaming scenarios (class-incremental, non i.i.d.), expanding it to the case where the input to the model is not a single data point (combined with $\mathcal{B}$ samples from the replay buffer) but a batch of new data points.

2.3 Evaluation Metric

The performance of CSSL is assessed using Ω_{all} (7), which is calculated as follows:

$$\Omega_{all} = \frac{1}{T} \sum_{t=1}^{T} \frac{\alpha_t}{\alpha_{\text{offline},t}}$$

Here,

- α_t represents the streaming performance at the t-th testing event

- $\alpha_{\text{offline},t}$ is the performance during offline testing at the same event
- T is the total number of testing events

Finally, Ω_{all} measures and compares the overall performance of streaming (while it is happening) to the performance obtained during offline training. For example, let's suppose that there are two streaming events, one after the model has been exposed to $\frac{1}{3}$ of the dataset (testing event $t = 1$) and one after all the dataset has been observed (end of streaming - (testing event $t = 2$). To compute Ω_{all}, we calculate the accuracy α_1 and α_2 of the streaming model at testing events $t = 1$ and $t = 2$ respectively and we also train the model offline, over two separate datasets, one containing the same $\frac{1}{3}$ fraction of the dataset mentioned earlier and one with the entirety of the dataset, which yields $\alpha_{\text{offline},1}$ and $\alpha_{\text{offline},2}$ respectively and perform the following computation:

$$\Omega_{all} = \frac{1}{2}\left(\frac{\alpha_1}{\alpha_{\text{offline},1}} + \frac{\alpha_2}{\alpha_{\text{offline},2}}\right)$$

The higher the value of Ω_{all} is, the better the performance.

2.4 Cold Start Streaming Learning

CSSL is illustrated in Figure 1 and encapsulated in algorithm 2. In a nutshell, CSSL first initializes the neural network's weights $\mathbf{W}$ either with random values or using pre-trained parameters ($Initialize$). For each new data point x_{new}, y_{new} of $\mathcal{D}$ that becomes available to the system, $\mathcal{B}$ samples from the replay buffer $\mathcal{R}$ get uniformly selected ($ReplaySample$). Then, the new data sample, gets concatenated with the data samples from the Replay Buffer ($\mathcal{X}, \mathcal{Y} := \{x_{new}\} \cup \mathcal{X}_{replay}, \{y_{new}\} \cup \mathcal{Y}_{replay}$) and the new batch of $\mathcal{B} + 1$ samples undergoes a process of data augmentation ($Augment$) before being used for training during the streaming phase ($StreamingUpdate$). The new data sample is stored in the Replay Buffer and occasionally, a random sample from the buffer is discarded when $\mathcal{R}$ has reached its capacity C ($ReplayEvict$). In the continual learning scenario, $|\mathcal{D}| = \infty$

Algorithm 1: Cold Start Streaming Learning (CSSL)

$\mathbf{W} := Initialize()$
$\mathcal{R} := \emptyset$
for $t = 1, 2, ..., |\mathcal{D}|$ **do**
  $x_{new}, y_{new} := \mathcal{D}_t$
  $\mathcal{X}_{replay}, \mathcal{Y}_{replay} := ReplaySample(\mathcal{R}, \mathcal{B})$
  $\mathcal{X}, \mathcal{Y} := \{x_{new}\} \cup \mathcal{X}_{replay}, \{y_{new}\} \cup \mathcal{Y}_{replay}$
  $StreamingUpdate(\mathbf{W}, Augment(\mathcal{X}, \mathcal{Y}))$
  $ReplayStore(\mathcal{R}, (Compress(x_{new}, y_{new}))$
  if $|\mathcal{R}| > C$ **then**
    $ReplayEvict(\mathcal{R})$
  end
end

2.4.1 Replay Buffer

The Replay Buffer $\mathcal{R}$ has a fixed size, denoted with its capacity C, and is used for storing complete compressed images along with their corresponding labels as the system navigates the data stream. New data can be added to R as long as the number of items in $\mathcal{R}$, is less than C. However, when $\mathcal{R}$ reaches its capacity, some of its elements must be removed ($ReplayEvict$ in Algorithm 2) to accommodate new data points that will become available as the streaming process proceeds. A straightforward policy for removing items, which is computationally efficient and its performance is comparable to more complex strategies, has been used in previous studies (8), and involves:

(i) identifying the class with the highest number of examples in the buffer, and

(ii) randomly selecting an example from this class to remove

This policy in CSSL because

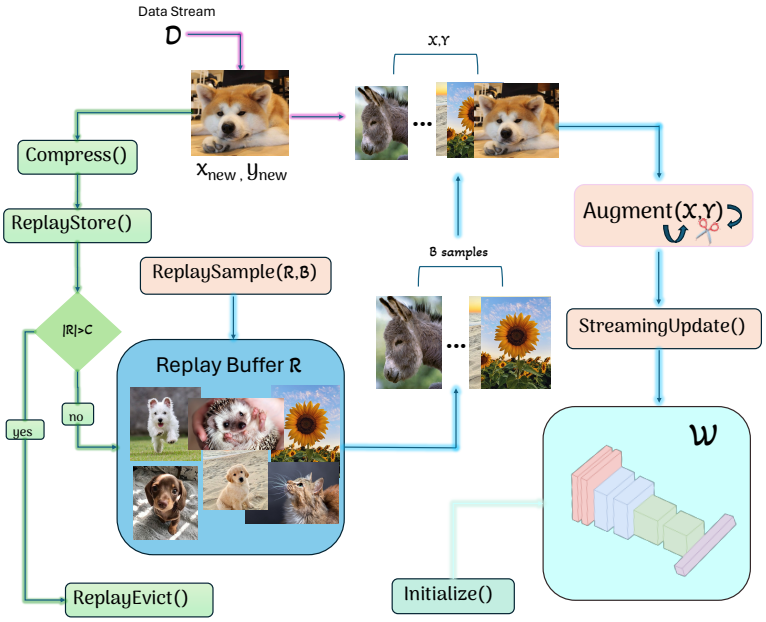

Figure 1: Illustration of CSSL.

2.4.2 Data Compression

In order to decrease memory usage, the data is compressed before being added to the Replay Buffer $\mathcal{R}$ ($Compress(x_{new}, y_{new})$ in 2). Previous approaches have minimized the memory footprint of replay data by freezing many network layers during the data streaming process, utilizing pre-trained feature representations and learned quantization modules (9; 8). Unlike these methods, CSSL stores full images, which may require more memory, but does not fix any network parameters, which enables the model to maximize its representational power.

CSSL examines various data-independent compression methods, including resizing images, quantizing the integer values of pixels, and saving images with JPEG compression on the disk. These techniques greatly lower memory costs while preserving performance levels. Nevertheless, storing full images in the replay buffer still results in greater memory usage compared to previous methods, which may not be suitable for environments with limited memory. However, many streaming applications, such as pre-labeling for data annotation for example, typically operate on cloud servers where memory capacity is less of an issue. For these types of scenarios, CSSL provides better performance than earlier methods, provided there is enough memory available for replay.

2.4.3 Model Updates

When processing a new example from dataset $\mathcal{D}$, CSSL adjusts the weights of the model using the $StreamingUpdate$ method which implements a basic stochastic gradient descent (SGD) update (as described in Algorithm 2). This update uses both the new data sample and $\mathcal{B}$ samples from the replay buffer obtained via $ReplaySample$, which selects data from the replay buffer $\mathcal{R}$ uniformly. Although alternative sampling strategies have been explored, they offer limited benefits at scale (1; 8). Streaming updates involve processing a combination of new and replayed data through a data augmentation pipeline (referred to as Augment in Algorithm 2) before this batch of data is fed to the neural network. Developing a robust and sophisticated data augmentation pipeline is fundamental to achieving CSSL's remarkable performance. More specifically, while previous studies have employed basic augmentation techniques (2; 13; 8), CSSL has investigated more advanced methods such as data interpolation (17; 16) and augmentation policies that are learned (3). Specifically, CSSL integrates random cropping and flipping, Cutmix, Mixup, and Autoaugment into a unified, sequential augmentation policy.

3 Benefits of CSSL

- **Full Plasticity**

  CSSL is an end-to-end approach, adjusting all model parameters with every update throughout the streaming process. Keeping network parameters fixed is detrimental to the learning process, as illustrated in figure 2 which depicts the scenario of a ResNet18 model pre-trained on ImageNet where various proportions of network parameters have been frozen during fine-tuning on CIFAR10/100. We observe that the final accuracy consistently decreases as the ratio of frozen parameters increases. Despite high-quality pre-training, fixing network parameters not only limits the model's capacity for representation but also hinders the adaptation of network representations to new data, making end-to-end training more advantageous.

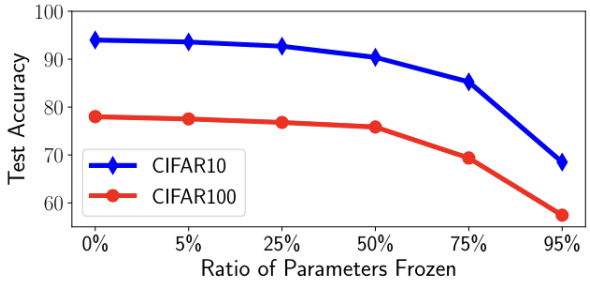

Figure 2: Test accuracy of ResNet18 models that were first pre-trained on ImageNet, and then fine-tuned on CIFAR10/100, with different proportions of frozen parameters.

- **Pre-training is not necessary**

  Previous streaming techniques typically rely on base initialization, where network parameters and other components are adjusted using a portion of the data before the streaming starts. However, this offline base initialization process is costly and it also makes the model's performance reliant on having access to a sufficient amount of pre-training data (6). The streaming performance suffers when there is insufficient base initialization data which will most likely result in subpar performance, if there is limited data available or none at all when streaming commences.

  CSSL does not rely on pre-training as it trains the network in an end-to-end manner during the streaming process. This approach simplifies the CSSL training pipeline to merely initializing and then training. This straightforwardness facilitates the easy implementation and deployment of CSSL in real-world applications. It's worth noting that CSSL can start streaming from a set of pre-trained model parameters, which often improves performance. However, the advantage of CSSL is that pre-training is entirely optional, with the network parameters being updated continuously during streaming.

4 Adaptive Learning Rates and Continual Learning

The performance of optimization algorithms in machine learning is greatly affected by their meta-parameters, i.e. hyperparameters, which are usually determined by a search process, like grid search or other trial-and-error techniques, before the training phase begins. Yet, the search for these meta-parameters has a considerably higher computational expense compared to training with already optimized meta-parameters (4; 10). Meta-parameter optimization aims to make this process more efficient by concurrently adjusting the meta-parameters throughout the training phase, thus moving away from the inefficient and often suboptimal trial and error methods towards a more compact and automated optimization process. In the field of continual learning, where environments are dynamic and loss functions are constantly changing, it's crucial to optimize meta-parameters, such as step sizes, to adapt to optimal values that evolve over time.

4.1 Quantifying an adaptive learning rate

4.1.1 Optimization theory background

One of the most fundamental optimization algorithms for minimizing a function $f(x) : \mathbb{R}^d \to \mathbb{R}$ is Gradient Descent, introduced by Cauchy et al. in 1847. Assuming that f is globally L-smooth, gradient Descent iterates with a step size η_t according to the following update rule:

$$x_{t+1} = x_t - \eta_t \nabla f(x_t) \tag{1}$$

while L-smoothness can be mathematically described as follows:

$$||\nabla f(x) - \nabla f(y)|| \leq L \cdot ||x - y|| \quad \forall x, y \in \mathbb{R}^d \tag{2}$$

The optimal step size for gradient descent is for $\eta_t = \frac{1}{L}$ which, when f is convex, guarantees the following convergence rate:

$$f(x_{t+1}) - f(x^*) \leq \frac{L||x_0 - x^*||}{2(2t+1)} \tag{3}$$

4.1.2 Adaptive step size algorithm

The requirement for global L-smoothness in equation (2) must be fulfilled for all x and y, which may result in a finite but arbitrarily large value for L. This makes the step size small, thus causing a slow convergence as demonstrated in equation (3). This challenge can be addressed by utilizing a step size for GD that relies on the local smoothness of f, which by definition, is smaller than the global L described in equation (2) (11). In other words, the learning rate adapts to local geometry of f, ensuring convergence depending only on the smoothness within the vicinity of a solution.

Mathematically, this step size is described described as:

$$\eta_t = \min\left\{ \frac{\|W_t - W_{t-1}\|}{2\|\nabla f(W_t) - \nabla f(W_{t-1})\|}, \sqrt{1 + \theta_{t-1}\eta_{t-1}} \right\}$$

Since we are working with neural networks, the x in (1) now denotes the parameters of our model (its weights) so we replace x with W. The intuition behind the factor $\frac{\|W_t - W_{t-1}\|}{2\|\nabla f(W_t) - \nabla f(W_{t-1})\|}$, is that we take a region where f is locally L-smooth, and plug into (2) the L that stems from the optimal learning rate for GD which is

$$\eta_t = \frac{1}{L} \Rightarrow$$
$$L = \frac{1}{\eta_t} \tag{4}$$

so substituting (4) in (2) we get:

$$||\nabla f(x) - \nabla f(y)|| \leq \frac{1}{\eta_t} \cdot ||x - y|| \quad \forall x, y \in \mathbb{R}^d \Rightarrow \tag{5}$$

$$\eta_t \leq \frac{||x - y||}{||\nabla f(x) - \nabla f(y)||} \tag{6}$$

And so we can set

$$\eta_t = \frac{||x - y||}{2||\nabla f(x) - \nabla f(y)||} \tag{7}$$

Algorithm 2: Adaptive learning rate algorithm

Initialize : $W_0, \eta_0 > 0, \theta_0 = +\infty$
$W_1 = W_0 - \eta_0 \nabla f(W_0)$
for $t = 1, 2, ...$ **do**
  $\eta_t = \min\left\{ \frac{\|W_t - W_{t-1}\|}{2\|\nabla f(W_t) - \nabla f(W_{t-1})\|}, \sqrt{1 + \theta_{t-1}\eta_{t-1}} \right\}$
  $W_{t+1} = W_t - \eta_t \nabla f(W_t)$
  $\theta_t = \frac{\eta_t}{\eta_{t-1}}$
end

Regarding the initialization values, in order for the algorithm to converge, we need to use a small value for η_0, for example set $\eta_0 = 0.0001$ whereas we need to set a large value for θ_0, for example $\eta_0 = 10000$.

For the first iteration of the algorithm, we can calculate $W_1 = W_0 - \eta_0 \nabla f(W_0)$ as we know the initialization values W_0, η_0. Then, for $t = 1$ we are going to calculate the value of the step size of the next iteration, η_1, as follows:

$$\eta_1 = \min\left\{ \frac{\|W_1 - W_0\|}{2\|\nabla f(W_1) - \nabla f(W_0)\|}, \sqrt{1 + \theta_0 \eta_0} \right\}$$

Using the above step size we can perform the next weight update of the model:

$$W_2 = W_1 - \eta_1 \nabla f(W_1)$$

Finally, we calculate the value of $\theta_1 = \frac{\eta_1}{\eta_0}$ that is going to be used for the calculation of η_2 during the following iteration, for $t = 2$ and so on.

5 Figures and Results

Figure 3 Illustrates the performance of CSSL under a class-incremental streaming learning process, measuring the accuracy of the model after seeing each class. As we can see, the baseline approaches exhibit high accuracy initially (while being exposed to data examples similar to the ones that were used during base initialization), but their performance drops significantly as the model gets exposed to more data, especially after processing data not relevant to the pre-training process. In contrast, CSSL with random initialization starts with relatively low performance, which gradually improves as more and more data is observed, finally reaching a stable performance plateau that outperforms all baseline methods. When starting from a pre-trained parameter initialization and transitioning to a streaming environment, this initial period of poor performance is eliminated, and the model can still achieve a stable plateau of higher performance.

Interestingly, not only does CSSL not suffer from this accuracy drop that baseline methods exhibit, but its performance keeps on improving as more and more data is getting streamed. This implies that end-to-end training helps eliminate biases towards the base initialization data. Moreover, CSSL's ability to maintain a stable performance level shows that streaming models don't necessarily deteriorate as the data stream diverges from the initial data, whereas CSSL's adaptability enables model representations to evolve in response to changing data, ensuring consistent performance over time, highlighting how beneficial full plasticity can be.

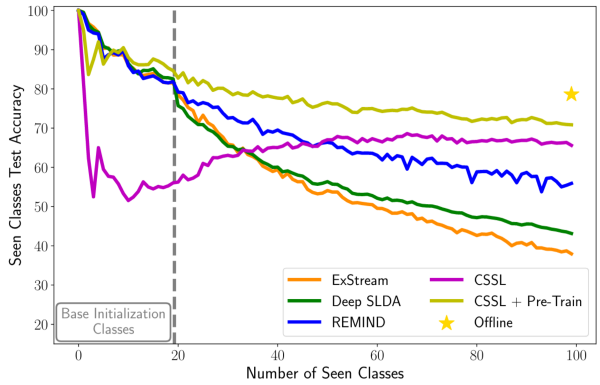

Figure 3: Streaming performance of class incremental learning on CIFAR100 after learning each new class as presented in (15), Figure 5.

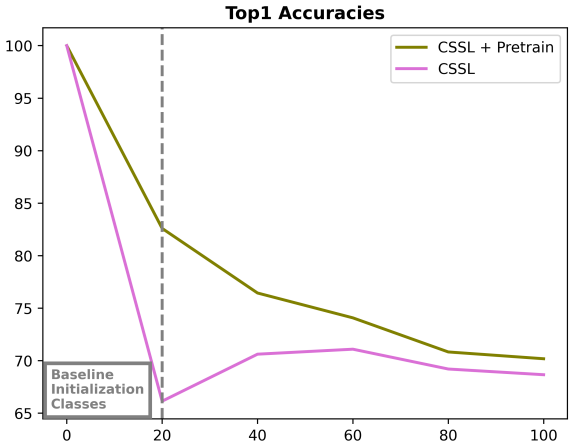

Figure 4: Reproduction of the green and purple lines of Figure 3.

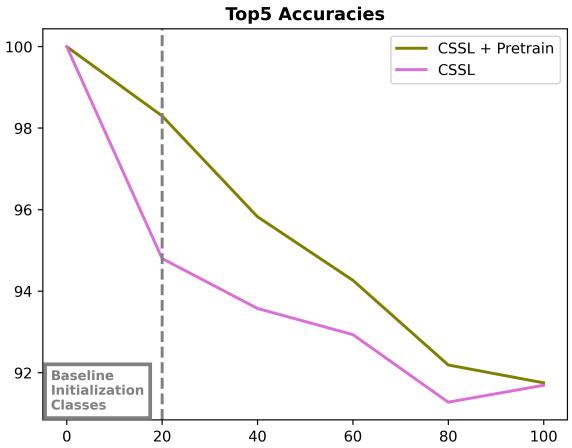

Figure 5: Reproduction of the green and purple lines of Figure 3 but with top-5 accuracy instead of top-1

References

[1] R. Aljundi, M. Lin, B. Goujaud, and Y. Bengio. Gradient based sample selection for online continual learning. *arXiv preprint*, 2019.

[2] F. M. Castro, M. J. Marín-Jiménez, N. Guil, C. Schmid, and K. Alahari. End-to-end incremental learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 233–248, 2018.

[3] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint*, 2018.

[4] G. E. Dahl, F. Schneider, Z. Nado, N. Agarwal, C. S. Sastry, P. Hennig, S. Medapati, R. Eschenhagen, P. Kasimbeg, D. Suo, et al. Benchmarking neural network training algorithms. *arXiv preprint*, 2023.

[5] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

[6] J. Gallardo, T. L. Hayes, and C. Kanan. Self-supervised training enhances online continual learning. *arXiv preprint*, 2021.

[7] T. L. Hayes, N. D. Cahill, and C. Kanan. Memory efficient experience replay for streaming learning. *arXiv e-prints*, Sept. 2018.

[8] T. L. Hayes, K. Kafle, R. Shrestha, M. Acharya, and C. Kanan. Remind your neural network to prevent catastrophic forgetting. In *European Conference on Computer Vision*, pages 466–483. Springer, 2020.

[9] T. L. Hayes and C. Kanan. Lifelong machine learning with deep streaming linear discriminant analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 220–221, 2020.

[10] H. Jin. Hyperparameter importance for machine learning algorithms. *arXiv preprint*, 2022.

[11] Y. Malitsky and K. Mishchenko. Adaptive gradient descent without descent. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 6702–6712. PMLR, 2020.

[12] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. Incremental classifier and representation learning. *arXiv preprint arXiv:1611.07725*, 2017.

[13] V. Verma, A. Lamb, C. Beckham, A. Najafi, I. Mitliagkas, D. Lopez-Paz, and Y. Bengio. Manifold mixup: Better representations by interpolating hidden states. In *International Conference on Machine Learning*, pages 6438–6447. PMLR, 2019.

[14] S. Wang, L. L. Minku, and X. Yao. A systematic study of online class imbalance learning with concept drift. *IEEE Transactions on Neural Networks and Learning Systems*, 29(10):4802–4821, 2018.

[15] C. R. Wolfe and A. Kyrillidis. Cold start streaming learning for deep networks, 2022.

[16] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6023–6032, 2019.

[17] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. Mixup: Beyond empirical risk minimization. *arXiv preprint*, 2017.

Solving Large-Scale Linear Regression with Distributed Feature Subsampling

Albert S. Zhu
Department of Computer Science
Rice University
Houston, TX 77005
asz3@rice.edu

Abstract

The abstract paragraph should be indented ½ inch (3 picas) on both the left- and right-hand margins. Use 10 point type, with a vertical spacing (leading) of 11 points. The word **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the abstract. The abstract must be limited to one paragraph.

1 Introduction

Consider the following system of n equations in d variables $x_1, \dots, x_d \in \mathbb{R}^n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2d}x_d &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d &= b_n. \end{aligned}$$

As we know, we can represent this as the matrix-vector multiplication $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n\times d}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^n$. Solving problems of this form has been a foundational task in mathematics for centuries because of its wide-ranging applications in fields such as engineering, physics, economics, and computer science – and has become particularly relevant in recent years due to the explosion in popularity of machine learning-informed data analysis. Because of its ubiquity, numerous methods have already been developed to solve this problem, including but not limited to Gaussian elimination, matrix factorization techniques such as $\mathbf{LU}$ [6] and $\mathbf{QR}$ [3] decomposition, as well as iterative methods like Gauss-Seidel [4] and Successive Over-Relaxation (SOR) [7].

However, as datasets within the Internet age have grown exponentially in size, it has become increasingly important to develop new, numerically stable, and both computationally and memory-efficient techniques for solving large-scale systems. This report will provide an overview of a particular line of research in finding distributed, parallelizable linear regression algorithms that have been developed over the past several months.

2 Problem Setting

Recall that we wish to solve the problem $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x} \in \mathbb{R}^d$, given a data matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$ and vector $\mathbf{b} \in \mathbb{R}^n$. One popular reformulation of this problem in terms of numerical optimization, known as linear regression, is the following:

$$\min_{\mathbf{x}\in\mathbb{R}^d} \mathcal{L}(\mathbf{x}) := \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 .$$

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

A classical method of solving this problem is with gradient descent, in which we perform iterative updates of the form

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \mathbf{A}^\top (\mathbf{A}\mathbf{x} - \mathbf{b})$$

given some initial point $\mathbf{x}_0 \in \mathbb{R}^n$ and learning rate η. This gives rise to the following algorithm:

Algorithm 1 Linear Regression with Gradient Descent

Input: Samples $\mathbf{A} \in \mathbb{R}^{n \times d}$, labels $\mathbf{b} \in \mathbb{R}^n$, learning rate η, number of iterations T

1: Initialize $\mathbf{x}_0$
2: **for** $t = 0, \ldots, T-1$ **do**
3: Compute $\mathbf{x}_{t+1} := \mathbf{x}_t - \eta \mathbf{A}^\top (\mathbf{A}\mathbf{x}_t - \mathbf{b})$
4: **end for**
5: **return** $\mathbf{x}_T$

A popular alternative to vanilla gradient descent is that of stochastic gradient descent, in which we instead perform the update

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \alpha_{i_t} \left(\alpha_{i_t}^\top \mathbf{x} - \mathbf{b} \right),$$

where we randomly sample a row vector α_{i_t} from A at each iteration. In particular, we have the following alternative to Algorithm 1:

Algorithm 2 Linear Regression with Stochastic Gradient Descent

Input: Samples $\mathbf{A} \in \mathbb{R}^{n \times d}$, labels $\mathbf{b} \in \mathbb{R}^n$, learning rate η, number of iterations T

1: Initialize $\mathbf{x}_0$
2: **for** $t = 0, \ldots, T-1$ **do**
3: Sample i_t from unif$(1, n)$
4: Compute $\mathbf{x}_{t+1} := \mathbf{x}_t - \eta \alpha_{i_t} \left(\alpha_{i_t}^\top \mathbf{x} - \mathbf{b} \right)$
5: **end for**
6: **return** $\mathbf{x}_T$

The motivation behind this approach is that computing the full gradient of our objective function $\mathcal{L}(\mathbf{x})$ during each iteration can be expensive (especially if $\mathbf{A}$ is large-scale), so we can instead split our objective function into the average of several objective functions that are easier to compute the gradient of – and the reason why this works is because the gradient of each smaller objective function is by construction an unbiased estimator of the original objective's gradient.

Keeping these ideas in mind, our approach is guided by the following key principles:

1. Instead of sampling rows for our smaller objective functions like in SGD, what if we sampled columns instead, and multiple at a time instead of just one?
2. What if we want to distribute our computation to L local workers?

To achieve both of these objectives, we can modify Algorithm 1 in a distributed fashion by creating L subproblems of a similar form for each local worker, whose results can then be aggregated into a single global update. In particular, on the kth global iteration, the ℓth local worker will receive a subproblem of the form

$$\min_{\mathbf{x} \in \mathbb{R}^d} \mathcal{L}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x}) := \frac{1}{2} \left\| \mathbf{A}\mathbf{M}_k^{(\ell)}\mathbf{x} - \mathbf{b} \right\|_2^2, \tag{1}$$

and the solution $\hat{\mathbf{x}}_k^{(\ell)}$ to this subproblem will then be aggregated with the rest into a single global update as follows:

$$\mathbf{x}_{k+1} = \frac{1}{L} \sum_{\ell=1}^{L} \mathbf{x}_k^{(\ell)}.$$

Here, $\mathbf{M}_k^{(\ell)}$ is a diagonal *mask* matrix where each diagonal entry has a fixed chance of being active or not; in this fashion, we can feed each local worker a random subsample of the columns of $\mathbf{A}$.

3 Approach 1: Deregularized Dropout

Initially, one approach to solving each worker's subproblem could be to apply gradient descent for each local worker, i.e. we have the update

$$\mathbf{x}_{k,t+1}^{(\ell)} = \mathbf{x}_{k,t}^{(\ell)} - \eta \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\right)^\top \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\mathbf{x}_{k,t}^{(\ell)} - \mathbf{b}\right),$$

which we can run for T iterations for each local worker as in the following algorithm:

Algorithm 3 Distributed Linear Regression with Masked Features

Input: Samples $\mathbf{A} \in \mathbb{R}^{n\times d}$, labels $\mathbf{b} \in \mathbb{R}^n$, learning rate η, number of global iterations K, mask distribution $\mathcal{D}$, number of workers L, number of local iterations T

Initialize $\mathbf{x}_0$
for $k = 0, \ldots, K-1$ **do**
  Sample $\left\{\mathbf{m}_k^{(\ell)}\right\}_{\ell=1}^{L}$ with $\mathbf{m}_k^{(\ell)} \sim \mathcal{D}$ for all $\ell \in [L]$
  for $\ell = 0, \ldots, L-1$ **in parallel do**
    Set $\mathbf{x}_{k,0}^{(\ell)} := \mathbf{x}_k$
    for $t = 0, \ldots, T-1$ **do**
      Initialize $\mathbf{M}_k^{(\ell)}$ as the diagonal matrix given by $\mathbf{m}_k^{(\ell)}$
      Compute $\mathbf{x}_{k,t+1}^{(\ell)} := \mathbf{x}_{k,t}^{(\ell)} - \eta \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\right)^\top \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\mathbf{x}_{k,t}^{(\ell)} - \mathbf{b}\right)$
    end for
  end for
  Compute $\mathbf{x}_{k+1} := \frac{1}{L}\sum_{\ell=1}^{L} \mathbf{x}_{k,T}^{(\ell)}$
end for
return $\mathbf{x}_K$

First, suppose $T = 1$; then since $\mathbb{E}_{\mathbf{M}_k^{(\ell)}}[\mathcal{L}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x})] \neq \mathcal{L}(\mathbf{x})$, we immediately run into an issue: our local subproblem becomes a biased estimator of $\mathcal{L}$, and therefore will not converge properly. To fix this, we can try to modify the loss function to remove the bias (i.e. *deregularize* it). If each diagonal entry of $\mathbf{M}_k^{(\ell)}$ is drawn from the distribution $\alpha^{-1}\text{Bern}(\alpha)$, so that $\mathbb{E}[\mathbf{M}_{ii}] = 1$ and $\mathbb{E}[\mathbf{M}_{ii}] = \alpha^{-1}$, then we have the following result:

Theorem 1 (Removing Bias)**.** *For* $\mathcal{L}_\mathbf{M}(\mathbf{x}) := \frac{1}{2}\|\mathbf{A}\mathbf{M}\mathbf{x} - \mathbf{b}\|_2^2$*, we have*

$$\mathbb{E}_\mathbf{M}[\mathcal{L}_\mathbf{M}(\mathbf{x})] = \mathcal{L}(\mathbf{x}) + \frac{\alpha^{-1}-1}{2}\mathbf{x}^\top \textit{Diag}\left(\mathbf{A}^\top\mathbf{A}\right)\mathbf{x}.$$

Thus, if

$$\hat{\mathcal{L}}_\mathbf{M}(\mathbf{x}) := \frac{1}{2}\|\mathbf{A}\mathbf{M}\mathbf{x} - \mathbf{b}\|_2^2 - \frac{\alpha^{-1}-1}{2}\mathbf{x}^\top\left(\mathbf{M}\odot\mathbf{A}^\top\mathbf{A}\right)\mathbf{x}, \tag{2}$$

then $\mathbb{E}_\mathbf{M}\left[\hat{\mathcal{L}}_\mathbf{M}(\mathbf{x})\right] = \mathcal{L}(\mathbf{x})$.

Using $\hat{\mathcal{L}}_\mathbf{M}$ as our loss function instead, this produces the following algorithm:

Algorithm 4 Distributed Linear Regression with Deregularized Dropout

Input: Samples $\mathbf{A} \in \mathbb{R}^{n \times d}$, labels $\mathbf{b} \in \mathbb{R}^n$, learning rate η, mask sampling probability α, number of global iterations K, mask distribution $\mathcal{D}$, number of workers L

1: Initialize $\mathbf{x}_0$
2: **for** $k = 0, \ldots, K-1$ **do**
3: **for** $\ell = 0, \ldots, L-1$ **in parallel do**
4: Sample diagonal matrices $\left\{\mathbf{M}_k^{(\ell)}\right\}_{\ell=1}^{L}$ with $\left[\mathbf{M}_k^{(\ell)}\right]_{ii} \sim \alpha^{-1}\text{Bern}(\alpha)$ for all $\ell \in [L]$
5: Compute $\mathbf{x}_k^{(\ell)} := \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\right)^\top \left(\mathbf{A}\mathbf{M}_k^{(\ell)}\mathbf{x}_k - \mathbf{b}\right) - \left(\alpha^{-1}-1\right)\left(\mathbf{M} \odot \mathbf{A}^\top\mathbf{A}\right)\mathbf{x}$
6: **end for**
7: Compute $\mathbf{x}_{k+1} := \mathbf{x}_k - \frac{\eta}{L}\sum_{\ell=1}^{L}\mathbf{x}_k^{(\ell)}$
8: **end for**
9: **return** $\mathbf{x}_K$

Unfortunately, this deregularization technique only works for the $T = 1$ case, and if we were to try to extend it for $T > 1$, we would have to produce increasingly complex derivations and therefore we deemed this technique to be untenable to extend to multiple local iterations.

4 Approach 2: Adaptive Step Size Schemes

For this second approach, we decided to investigate if there was any experimental potential in using an adaptive step size scheme for each local worker in their individual learning processes, since originally we were just using the classical $\frac{1}{\sigma}$ step size from optimization theory (where σ is the largest singular value of $\mathbf{A}\mathbf{M}_k^{(\ell)}$). We tested the following 2 schemes:

4.1 Minimizing Loss at Each Step

The idea behind this first scheme was to simply select η to minimize the loss at each gradient descent update; in other words, we wanted

$$\arg\min_{\eta \in \mathbb{R}} \mathcal{L}_{\mathbf{M}_k^{(\ell)}}\left(\mathbf{x}_{k,t+1}^{(\ell)}\right) = \mathcal{L}_{\mathbf{M}_k^{(\ell)}}\left(\mathbf{x}_t^{(\ell)} - \eta\nabla\mathcal{L}_{\mathbf{M}_k^{(\ell)}}\left(\mathbf{x}_{k,t}^{(\ell)}\right)\right).$$

One straightforward to solve this is by setting $\frac{\partial}{\partial\eta}\left[\mathcal{L}_{\mathbf{M}_k^{(\ell)}}\left(\mathbf{x}_{k,t+1}^{(\ell)}\right)\right] = 0$, which gives

$$\eta_{k,t}^{(\ell)} = \frac{\left\|\mathcal{L}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x}_{k,t}^{(\ell)})\right\|_2^2}{\left\|\mathbf{A}\mathbf{M}_k^{(\ell)}\nabla\mathcal{L}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x}_{k,t}^{(\ell)})\right\|_2^2}.$$

4.2 Decaying Polynomial Roots

The second adaptive step size scheme we tried was by finding the roots to the polynomial $\sum_{t=0}^{T} a_t(-1)^{T-t}x^t$, where $a_T = 1$, $a_t = \binom{T}{t}\alpha^{T-t-1}\eta^{T-t}$ for $0 \le t \le T-1$, and $\eta = \frac{1}{\sigma}$. The idea was to use decaying step sizes $\eta_1, \ldots, \eta_T$ that could somewhat approximate the classical step size η, so the coefficients were derived from the following equalities:

$$\sum_{i=1}^{T}\eta_i = \binom{T}{1}\eta$$

$$\sum_{1 \le i < j \le T}\eta_i\eta_j = \binom{T}{2}\alpha\eta^2$$

$$\vdots$$

$$\prod_{i=1}^{T}\eta_i = \binom{T}{T}\alpha^{T-1}\eta^T.$$

Unfortunately, neither of these schemes ended up achieving good experimental results (either converging to around 0.1 error or diverging), and the second scheme in particular frequently had complex roots, so we decided to try to overhaul the theory behind our approach entirely.

5 Approach 3: Federated Learning

The similarities between our problem setting and that of federated learning were immediately evident: in federated learning, the objective is to minimize a function f of the form

$$f(x_1, \ldots, x_L) = \frac{1}{L} \sum_{\ell=1}^{L} f_\ell(x_\ell),$$

by giving L local workers the problem of optimizing $f_\ell(x_\ell)$ while enforcing a *consensus* condition to converge to a common value x by the end of local training. Federated learning is also especially appealing because settings in which it can be applied are generally characterized by data heterogeneity, massive scale, and privacy constraints, all of which are relevant within our general distributed approach given by 1.

Although there are already many existing federated learning frameworks for solving convex optimization problems like FedProx [1], FedPD [8], and FedSplit [2], we decided to try to apply the approach outlined by FedDR [5] since it seemed to fit our problem setting the best. FedDR utilizes the eponymous Douglas-Rachford splitting technique along with randomized block-coordinate strategy to handle nonconvex federated objectives, and for our problem, we have the updates

$$\begin{aligned} \mathbf{u}_k &= \text{prox}_{L\eta\hat{\mathcal{L}}}\left(\mathbf{v}_k\right) \\ \mathbf{v}_{k+1} &= \text{prox}_{L\eta\delta_{\mathcal{S}}}\left(2\mathbf{u}_k - \mathbf{v}_k\right), \end{aligned}$$

where $\mathbf{u}_k = \left[\mathbf{x}_k^{(1)}, \ldots, \mathbf{x}_k^{(L)}\right] \in \mathbb{R}^{(L+1)d}$ consists of the concatenated solutions obtained from each local worker and $\hat{\mathcal{L}}(\mathbf{u}_k) := \frac{1}{L}\sum_{\ell=1}^{L} \hat{\mathcal{L}}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x}_k^{(\ell)})$, using the same deregularized objective function $\hat{\mathcal{L}}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x})$ from 2 in aggregation to approximate $\mathcal{L}$.

Observing that we only need to consider the active entries for each local worker, we can then set the consensus condition for worker ℓ to $\mathbf{M}^{(\ell)}\mathbf{x}^{(\ell)} = \beta\mathbf{M}^{(\ell)}\mathbf{x}^{(0)}$ for some fixed $\mathbf{x}^{(0)}$ and $\beta \neq 0$. We can also split up $\mathbf{v}_k$ as $\left[\mathbf{y}_k^{(0)}, \ldots, \mathbf{y}_k^{(L)}\right]$. After a lot of expanding and simplifying, we eventually get the update

$$\begin{aligned} \mathbf{x}_k^{(\ell)} &\in \underset{\mathbf{x}\in\mathbb{R}^d}{\arg\min}\, \hat{\mathcal{L}}_{\mathbf{M}_k^{(\ell)}}(\mathbf{x}) + \frac{\alpha^2}{2\eta}\left\|\mathbf{M}_k^{(\ell)}\left(\mathbf{y}_k^{(\ell)} - \mathbf{x}\right)\right\|_2^2 \\ \mathbf{y}_{k+1}^{(\ell)} &= \left(\frac{1}{\alpha\beta^2}\mathbf{I} + \sum_{\ell=1}^{L}\mathbf{M}_k^{(\ell)}\right)^{-1}\left(\left(\frac{1}{\alpha\beta^2}\mathbf{I} - \sum_{\ell=1}^{L}\mathbf{M}_k^{(\ell)}\right)y_k^{(\ell)} + 2\sum_{\ell=1}^{L}\mathbf{M}_k^{(\ell)}\mathbf{x}_k^{(\ell)}\right). \end{aligned}$$

This provides the following algorithm:

Algorithm 5 Distributed Linear Regression with Consensus

Input: Samples $\mathbf{A} \in \mathbb{R}^{n \times d}$, labels $\mathbf{b} \in \mathbb{R}^n$, learning rate η, number of global iterations K, mask sampling probability α, number of workers L, number of local iterations T, model drifting regularization β

Initialize $\mathbf{x}_0$
for $k = 0, \ldots, K-1$ **do**
  Sample diagonal matrices $\left\{\mathbf{M}_k^{(\ell)}\right\}_{\ell=1}^{p}$ with $\left[\mathbf{M}_k^{(\ell)}\right]_{ii} \sim \text{Bern}(\alpha)$ for all $\ell \in [L]$
  for $\ell = 1, \ldots, p$ **do**
    $\mathbf{x}_{k,0}^{(\ell)} := \mathbf{M}_k^{(\ell)} \mathbf{x}_k$; $\mathbf{A}_k^{(\ell)} = \mathbf{A}\mathbf{M}_k^{(\ell)}$
    for $t = 0, \ldots, T-1$ **do**
      $\mathbf{x}_{k,t+1}^{(\ell)} := \mathbf{x}_{k,0}^{(\ell)} + \frac{\eta}{\alpha}\mathbf{A}_k^{(\ell)\top}\mathbf{b} - \frac{\eta}{\alpha^2}\left(\mathbf{A}_k^{(\ell)\top}\mathbf{A}_k^{(\ell)} - (1-\alpha)\text{Diag}\left(\mathbf{A}_k^{(\ell)\top}\mathbf{A}_k^{(\ell)}\right)\right)\mathbf{x}_{k,t}^{(\ell)}$
    end for
  end for
  $\mathbf{x}_{k+1} := \left(\frac{1}{\alpha\beta^2}\mathbf{I} + \sum_{\ell=1}^{p}\mathbf{M}_k^{(\ell)}\right)^{-1}\left(\left(\frac{1}{\alpha\beta^2}\mathbf{I} - \sum_{\ell=1}^{p}\mathbf{M}_k^{(\ell)}\right)\mathbf{x}_k + 2\sum_{\ell=1}^{p}\mathbf{x}_{k,T}^{(\ell)}\right)$
end for
return $\mathbf{x}_K$

Testing this algorithm experimentally, we discovered that varying the condition number κ, mask sampling probability α, and number of local iterations, we had the following results:

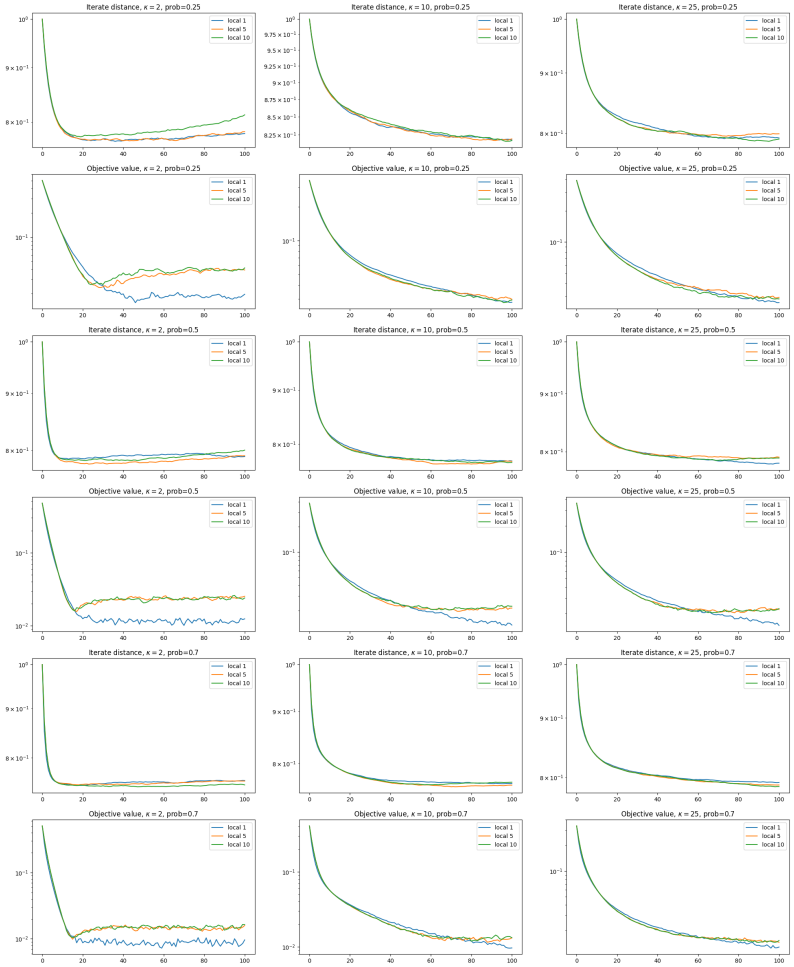

Figure 1: Results of 5 for a randomly generated matrix with $n = 200, d = 500$

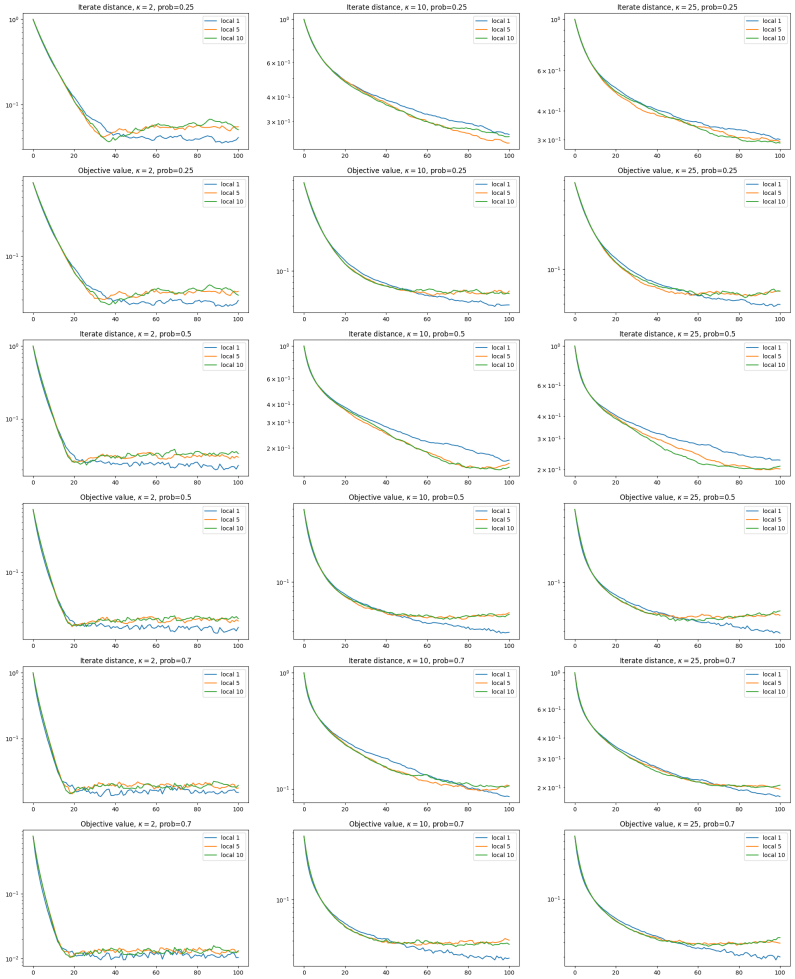

Figure 2: Results of 5 for a randomly generated matrix with $n = 500, d = 200$

For both settings, experiments were conducted with a step size of $\eta = \frac{1}{2\sigma}$, half of the classical step size. So far, the convergences have been somewhat high but preliminary observations are that most of the time performing more local iterations does not seem beneficial for better convergence; this could potentially be due to the approximation of $\mathcal{L}$ being too inaccurate, or perhaps due to some issues with using a constant step size. Future directions could include introducing more variance in how $\mathbf{A}$ is generated, as well as testing decaying step sizes; for the latter, preliminary results seem to indicate that halving the step size after some number of global iterations can achieve better convergences (below 0.01), but it's still unclear how each of the hyperparameters are interacting with each other.

References

[1] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. Federated optimization in heterogeneous networks, 2020.

[2] R. Pathak and M. J. Wainwright. Fedsplit: An algorithmic framework for fast federated optimization. *CoRR*, abs/2005.05238, 2020.

[3] E. Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. i. teil: Entwicklung willkürlicher funktionen nach systemen vorgeschriebener. *Mathematische Annalen*, 63:433–476, 1907.

[4] L. Seidel. Ueber ein verfahren die gleichungen, auf welche die methode der kleinsten quadrate führt, sowie lineäre gleichungen überhaupt, durch successive annäherung aufzulösen. *Abh. Bayer. Akad. Wiss. Math.-Naturwiss. Kl.*, 11(3):81–108, 1874.

[5] Q. Tran Dinh, N. H. Pham, D. Phan, and L. Nguyen. Feddr –randomized douglas-rachford splitting algorithms for nonconvex federated composite optimization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 30326–30338. Curran Associates, Inc., 2021.

[6] A. M. Turing. Rounding-Off Errors in Matrix Processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 01 1948.

[7] D. Young. Iterative methods for solving partial difference equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111, 1954.

[8] X. Zhang, M. Hong, S. Dhople, W. Yin, and Y. Liu. Fedpd: A federated learning framework with adaptivity to non-iid data. *IEEE Transactions on Signal Processing*, 69:6055–6070, 2021.

Probabilistically Compressed Embedding Tables

Ayush Sachdeva
Department of Computer Science
Rice University
Houston, TX 77005
as216@rice.edu

Abstract

As large language models become larger, they require more computational and memory resources to train and run inference on. This resource-hungry nature of large language models limits their utility in resource-constrained environments like mobile devices. The usual solution to this problem is to run models on the cloud but this has various privacy and latency concerns for certain use cases. This paper focuses on reducing the size of the embedding table in these models which can have a significant impact on the memory footprint of compact large language models. We detail the recent advancements in this area in this survey paper and also introduce a new embedding method based on probabilistic count-sketch algorithms.

1 Introduction

In natural language processing, pre-trained transformer language models such as BERT (1), RoBERTa (2), and ALBERT (3) have been established as the standard models for a variety of language tasks like text classification, sequence labeling, and text generation. These models usually comprise a series of transformer-based layers that act as the backbone and can be plugged in with different output layers and fine-tuned for different tasks.

These models have been shown to have excellent predictive and generative power but have also been recognized to have a high memory footprint due to their large number of parameters. Previous works (**?**) have shown that increasing the number of parameters in the model increases its representational ability and accuracy on different tasks. This has resulted in the size of state-of-the-art language models increasing by many orders of magnitude.

However, the size of these models presents a challenge when we try to deploy them on resource-constrained environments like a mobile device or a watch. This is an important problem as as models become more frequently used on mobile applications, there might be various applications of large language models that need to use the private data of users and hence, can't be processed on the cloud. Instead, these models will need to be run on-device to respect user privacy. Another common use case is when we want to run these models on-edge for lower latency in certain enterprise IoT solutions.

As the number of applications using large language models in these environments increases, we also expect the memory available for each model to decrease further limiting resources. The common practice is to fine-tune a model for each different task from a pre-trained backbone, so we have a different model for each different application on the device. So along with reducing the memory footprint of these models individually, we also want to identify frameworks that allow us to solve multiple language tasks with a single base model.

In this paper, we want to answer the following questions in the context of resource-constrained environments:

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

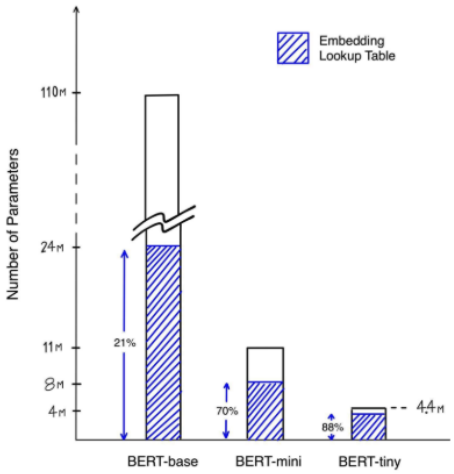

Figure 1: Embedding Table sizes for BERT and compact versions (7)

- Is it possible to use a large language model (or a model with comparable performance) without the massive disk, memory, and compute requirements?
- Is there a modular way to use a single pre-trained backbone for different language tasks on a device with limited memory resources?

The first question has been a topic of active research in recent years. Most of the work has focused on areas like model pruning (4), quantization (5), and distillation (6). These works have shown significant improvements in the efficiency of memory usage but have not considered compressing the parameters stored in the embedding layer of the model. Also, note that the memory footprint of a model can be determined by the memory used to store its parameters. Hence, the memory footprint is determined by the number of parameters used in the model, assuming that a parameter has a fixed predetermined size.

Our approach is focused on replacing the classical embedding table with a probabilistic alternative which takes up less memory. Earlier works (7) have shown that the embedding table in large language models makes up for a significant number of parameters, especially in compact models which are smaller versions of the original models (after model compression based on knowledge-distillation) e.g. BERT-Tiny (8) is a compact version of BERT. Figure 1 (7) shows the percentage of parameters in the classical embedding tables of BERT-base, and its compact versions: BERT-mini and BERT-tiny. Note that even for BERT-base, the embedding table takes up a significant % of the parameters, but this % becomes much more exaggerated in distilled models occupying almost 90% of the model. Hence, any compression in the size of the embedding table would have a significant impact on the memory footprint of the overall model.

Various approaches to table compression have been explored for language models including dynamic embeddings, hashing, etc. We discuss some of these models in detail in Section 3. Inspired by these probabilistic ideas, we establish an embedding algorithm based on the popular probabilistic count-sketch algorithm in Section 4.

For the second question, we explore previous work (9) that proposes a framework for using a single backbone for multiple tasks combining the popular LoRA (10) framework with an embedding table alternative in Section 3.

2 Background

A classical embedding table is a lookup that maps every token in the vocabulary to a vector in some embedding space via an embedding matrix. This embedding matrix is trainable and has one unique embedding vector for each token in the vocabulary. Due to this one-to-one mapping, the embedding

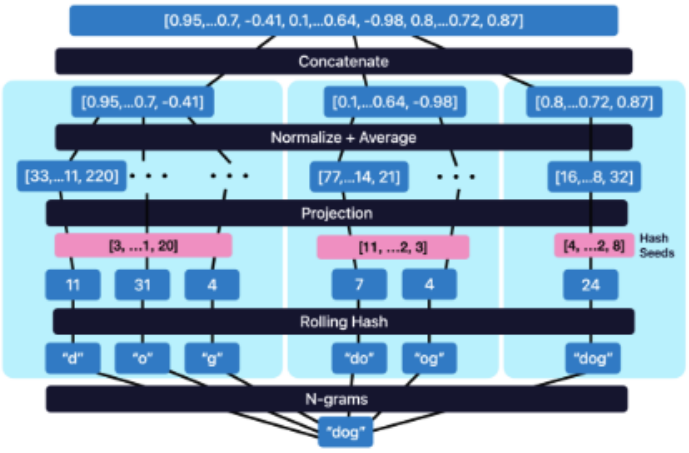

Figure 2: Computation of token embedding in EELBERT (7)

table takes scales linearly with the vocabulary size. This embedding layer is the first layer that the tokenized input goes through before getting to the transformer architecture in a vanilla pre-trained transformer model. Additionally, for a pre-training task like masked language modeling (1), this layer is also coupled with the output layer which predicts the masked tokens using the embedding table.

For the rest of the paper, we assume an understanding of the transformer architecture and vanilla pre-trained transformer models like BERT (1). We also use W_q, W_k, W_v and W_o to refer to the query/key/value/output projection matrices in the self-attention module.

3 Related Work / Survey

This section discusses different ways to replace the classical embedding table. This line of research can be divided into two groups:

- Post-training compression: Compressing the learned embedding table after training
- Pre-training compression: Training the model with a compressed embedding table

This section and the rest of the paper focus on the latter of those groups. Subsection 2.1 will discuss an embedding technique that dynamically computes the embeddings instead of using a memory-occupying table. Subsection 2.2 will discuss a technique of compressing embedding tables in recommendation systems that can be easily adapted to natural language processing. Note that we will only be focusing on the embedding alternatives from these papers and will skip over other aspects that might be less relevant.

We also discuss frameworks for more efficient use of a single backbone for different tasks in this section. Specifically, we discuss LoRA (10) in Subsection 2.3, a popular technique to reduce the memory footprint of backbone fine-tuning. Subsection 2.4 focuses on combining LoRA with the embedding tables alternatives we discussed in earlier subsections.

3.1 EELBERT: Tiny Models through Dynamic Embeddings (7)

This paper eliminates the trainable embedding matrix by computing dynamic (and mostly deterministic) embeddings for the tokens in the vocabulary. The process of computing these embeddings dynamically is shown in Figure 2:

- Divide input token into n-grams (i.e. subsequences of length $1, 2, \ldots, n$ where the token is of length n)

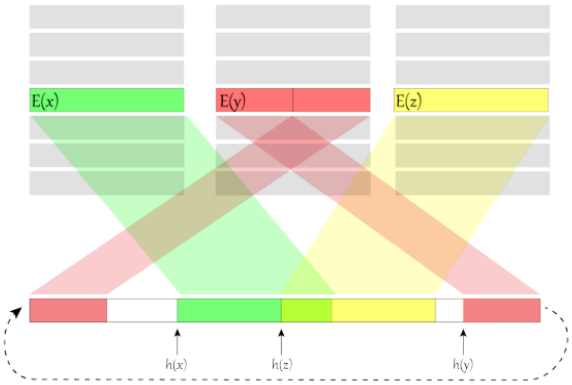

Figure 3: Computation of token (for tokens x, y, z) embedding in ROBE-D (11)

- For each n-gram, compute the hash value using a rolling hash function and pre-initialized random hash seeds
- For each $i \in [1, n]$, compute a projection matrix P_i as the outer product of a vector of the hashed values of each i-gram and a subset of the random hash seeds.
- Average out each projection matrix P_i across the rows to get an embedding vector e_i for each $i \in [1, n]$
- Concatenate e_i for all i to get the final embedding for the input token

Pre-training and fine-tuning BERT (1) with this embedding layer yields a model with comparable metrics to BERT across different language tasks. Specifically, for BERT-base, the paper observes a reduction of 21% in terms of memory and a 1.5% reduction in the GLUE score of the model. Note that this alternative model is the same as BERT except in the input embedding layer.
//Hence, this suggests that we can replace the trainable embedding table with a dynamic embedding with no impact on performance. Further ablation studies in the paper suggest that using a completely random hash function for the embeddings will not work, especially in smaller model sizes.

Note that the significant impact of this dynamic embedding is the increase in latency. This impact is less noticeable in large models but can be as high as $2.3x$ for compact models like BERT-Tiny.

The paper concludes that using n-gram pooling hash function to compute the dynamic embeddings is a feasible alternative to having a classical embedding table if the increased latency is acceptable.

3.2 Random Offset Block Embedding (ROBE) for compressed embedding tables in deep learning recommendation systems (11)

Deep learning recommendation systems also use an embedding table for categorical tokens which are the same as embedding tables in our context. Hence, it is a good exercise to explore ideas for embedding table compression from the recommendation system domain which is a more mature area in terms of efficiency.

This paper emphasizes the importance of achieving orders of magnitude more reduction in the memory footprint of the model to have a significant impact on the utility of these compact models. Inspired by other works using weight sharing, this paper proposes a new algorithm to replace the embedding table called ROBE (Random Offset Block Embedding). Instead of storing an embedding table, ROBE maintains a single array for learned parameters which is a compressed representation of an embedding table. This array is shared across the model and can be used to generate embeddings.

To replace an embedding table of size $|V| \times D$, we can use ROBE-D where D is the dimension of the embedding. The learnable matrix is reduced to a circular array M of size m. We also

have two independent hash functions: $h : \mathbb{N} \to \{0, \ldots, m-1\}$ and $g : \mathbb{N} \times \mathbb{N} \to \{-1, 1\}$.

The computation of the embedding for token x in ROBE-D is done in the following manner (shown in Figure 3):

- Hash token x using h to get the starting index for the initial embedding
- Let your initial embedding i.e. $P(x)$ be D-dimensional sub-array of the circular array M starting at position $h(x)$. Note that as M is circular if $h(x) + D \geq m$, we circle back to the beginning of M to complete $P(x)$.
- Compute $G(x) = \{g(x, 1), g(x, 2), \ldots, g(x, D)\} \in \{-1, 1\}^D$
- The final embedding is the element-wise product of $G(x)$ and $P(x)$ i.e. $E(x) = G(x) \circ P(x)$

The ROBE-D algorithm is further adapted to use concatenated smaller chunks of the array instead of a single chunk by adding more hash functions. The paper shows higher performance in terms of memory latency and irregular memory accesses for ROBE-Z.

As the paper is focused on recommendation systems, most of the results and analysis are geared towards that community. So the paper shows that this method of compression is more effective than other techniques like quantization, other hashing techniques, MD Embeddings (12), etc. by many orders of magnitude. Specifically, by choosing appropriate values for m and Z, the paper shows a memory compression of up to $1000x$ compared to the embedding look-up table. This is a very promising result as there is little or no change in the baseline metric used to measure the effectiveness of the overall deep learning recommendation system model.

Due to the high degree of compression and similarities between the trainable embedding table in recommendation systems and language models, this is an interesting area to explore. Specifically, to implement a ROBE-D/Z compressed embedding table algorithm for language models and test the kind of memory-performance trade-off we achieve.

3.3 LoRA: Low-Rank Adaptation of Large Language Models (10)

This paper and the LoRA algorithm form the base for a variety of techniques trying to increase the efficiency of fine-tuning and inference using large language models. This paper is important to explore for the second question we are trying to answer in this report: Is there a modular way to use a single pre-trained backbone for different language tasks on a device with limited memory resources?

LoRA aims to replace a different model for every language task with a single backbone (with frozen weights) and many small "LoRA" modules for different tasks. This reduces the memory required per task and makes task-switching between different tasks more efficient. Also, LoRA modules make fine-tuning more efficient.

The fine-tuning task can be expressed as updating all relevant weight matrices in a transformer model via a gradient update step which adds ΔW_i to each matrix $W_i \in \mathbb{R}^{d \times k}$. Storing this ΔW_i for every weight matrix is inefficient. However, previous work (13) shows that pre-trained language models have a low "intrinsic dimension" and can still learn when projected to a smaller subspace. Hence, the paper argues we can represent ΔW_i as a low-rank decomposition i.e. $\Delta W_i = BA$ where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$ and $r << min(d, k)$. Note that the paper chooses to freeze the MLP modules and only train the attention weight matrices during fine-tuning.

So the LoRA modules comprise the A, B matrices with a chosen value for r that achieves the memory reduction required. For fine-tuning, we initialize A as a random Gaussian and B as zero so $BA = 0$ at the beginning of training. We can fine-tune our backbone for any downstream task by keeping track of a low-rank representation of the weight matrices.

The paper shows that LoRA modules have comparable accuracy to fine-tuned models with the same backbones for pre-trained models like GPT-3, RoBERTa, etc. Additionally, as the updates can be stored in low-rank matrices, this allows us to reduce the trainable parameters, which is directly proportional to the memory occupied by each fine-tuned model in addition to the backbone, by many orders of magnitude.

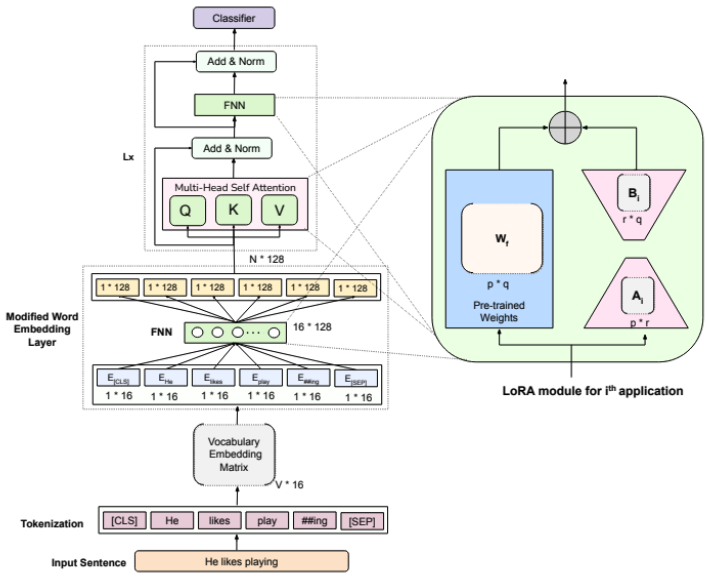

Figure 4: NanoBERT Architecture with intermediate embedding size $= 16$ (9)

3.4 NanoBERT: An Extremely Compact Language Model (9)

This paper explores an alternative to the classical embedding lookup table but more importantly, emphasizes the importance of combining this with LoRA fine-tuning modules for maximum efficiency.

As this paper was initially written for large language models, the motivations are similar to EELBERT (7) but they define a new way of computing embeddings. Specifically, they replace the embedding matrix with an intermediate embedding matrix which maps each token to an embedding with a smaller size. As the two dimensions of the embedding matrix are $|V|$, the vocabulary size, and d, the size of the embedding, this directly reduces the number of parameters in the model. However, we still want the final embeddings to have the original (larger) size, as a smaller embedding size can decrease the representational ability of the model.

In the overall NanoBERT architecture, the first few layers show how the embeddings for a sequence of tokens can be computed. This is a simple module that can be used to reduce the embedding matrix size for any transformer-based language model. The embedding computation is specified below::

- For each token, get the intermediate embedding (with the smaller size, say 16) from the intermediate embedding matrix.
- Pass each embedding through a feedforward neural network (FNN) which has input size as the small embedding size, say 16, and output size as the large embedding size, say 128.
- The computed embedding after passing through the FNN is the final embedding and has size $= 128$ i.e. the original (large) embedding size of the model.

This paper also explores ways to combine LoRA with these embedding matrix alternatives. Usually, the updates to the embedding matrix are not represented as the product of low-rank matrices so we can not use LoRA with the embedding matrix. However, note that the architecture in Figure 4 shows the use of LoRA modules for the embedding layer which has a large FNN matrix along with the attention modules. Hence, the only part of the model that can not be fine-tuned using the LoRA algorithm is the small modified embedding matrix, which is a relatively small matrix. This reduces the size of the model when used for a single language task by reducing the size of the embedding matrix, and reduces the overall size of all models when a single backbone is being used for multiple tasks with a combination of LoRA and the reduction in the size of the embedding matrix.

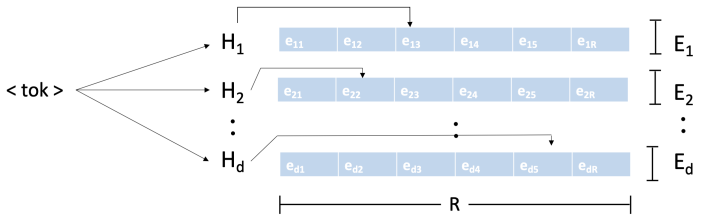

Figure 5: Count Sketch Embedding Algorithm to compute embedding for a token. The final embedding is an element-wise median of each of the d chosen embeddings.

The results in the paper suggest that the accuracy of the pre-trained BERT-Tiny model with the modified embeddings is comparable (or better) than the model with classical embeddings. Note that in this case, the size of a classical embedding is 128 and the size of an intermediate embedding for NanoBERT is 16. Hence, the size of the model is reduced by about 75%.

4 Count-sketch Embeddings

As explained earlier, there are a few different ways of reducing the memory footprint of the embedding matrix:

1. Replace embedding matrix by computing embeddings dynamically. This eliminates the memory used by the matrix but increases latency.
2. Reduce the dimensionality of each embedding vector as each row of the embedding matrix has the same size as the dimensionality of the embedding vector.
3. Reduce the number of tokens/elements that we are mapping into the embedding matrix as the number of rows in the embedding matrix is determined by the number of elements.

We noticed that there were no algorithms that were reducing the size of the embedding matrix by reducing the number of elements mapping into the matrix i.e. method 3 in the list above. To fill this gap, we propose Count-sketch embeddings, inspired by the classic Count-sketch method (14) which models statistics of a data stream by hashing into a much smaller array than the stream length and using probabilistic guarantees to decrease the error from the actual statistic.

Our algorithm takes advantage of previous works (15) which have shown that attention sparsity in various pre-trained language models is as high as 99% in deeper layers of the network. Hence, there is a small set of tokens that is "very" influential for the model that we can call the "heavy-hitter tokens" in parallel to the heavy-hitter elements of a data stream in the context of the classic Count-sketch algorithm.

For our algorithm to compute the embeddings for a token, x, we need to have d independent hash functions, $H_1, H_2, \ldots, H_d$ and d matrices with R embeddings each, $E_1, E_2, \ldots, E_d$. Note that d and R are hyperparameters that can be chosen such that $dR << |V|$ where $|V|$ is the size of the vocabulary. So we reduce the embedding table by a factor of $\frac{|V|-dR}{|V|}$. We can compute the embedding in the following manner (as shown in Figure 5):

- Compute the hash values of x using $H_1, \ldots, H_d$ modulo R which gives us a position in each of the $E_1, \ldots, E_d$ array.
- Select an embedding from each matrix at the position specified by the hash value i.e. the ith embedding would be $E_i[H_i(x)\%R]$.
- The final embedding is the elementwise median of each of the d chosen embeddings.

As for previous works, this embedding computation can be used with any transformer-based language model. Additionally, we can use LoRA for fine-tuning different language tasks using the same backbone in the same manner as in NanoBERT (9).

Table 1: Resulting comparing BERT-base and Count-BERT

Model	BERT-mini	Count-BERT
Model Size	44.8MB	16.79MB
SST-2 (Acc.)	0.851	0.5092
QNLI (Acc.)	0.827	0.5054
RTE (Acc.)	0.552	0.5271
MRPC (Acc.)	0.701	0.6838
QQP (Acc.)	0.864	0.6318
MNLI (Acc.)	0.719	0.3533
GLUE Score	0.753	0.5351

5 Experimentation

To ensure that using our algorithm to compute embeddings doesn't affect the accuracy of the original models on basic NLP tasks, we use our algorithm with BERT-mini. We pre-train the model on the English Wikipedia Corpus and report the accuracies compared with the original model in Table 1.

Note that we can not pre-train the model in the same way as BERT-mini because the input embedding table is used again for the masked language modeling output loss computation. Hence, for pre-training we maintain a classical embedding table only for loss computation which is not used for fine-tuning.

6 Results

Unfortunately, our results show that even though we achieved a significant reduction in model size, the accuracy for various language tasks significantly suffers. Hence, the current version of the count-sketch embedding algorithm is not an effective replacement for the embedding table.

7 Conclusion

Through this project, we have explored different ways of replacing the classical embedding table in transformer-based language models to reduce the memory footprint of these models, especially for deployment in resource-constrained environments. We also focused on approaches that can be combined with LoRA to provide further compression in cases where the same backbone can be used for different language tasks. Additionally, we proposed a novel algorithm to replace the classical embedding table with Count-sketch embeddings. However, our current results indicate that this algorithm is not an effective replacement as it has a significant impact on the accuracy of the model for different language tasks.

8 Acknowledgments

This work was done under the guidance of Dr. Anshumali Shrivastava and Dr. Anastasios Kyrillidis.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," ACL Anthology. https://aclanthology.org/N19-1423/

[2] Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach," arXiv.org, Jul. 26, 2019. https://arxiv.org/abs/1907.11692

[3] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," arXiv.org, Sep. 26, 2019. https://arxiv.org/abs/1909.11942

[4] M. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning," in Proceedings of the 5th Workshop on Representation Learning for NLP, 2020. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.18653/v1/2020.repl4nlp-1.18

[5] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8BERT: Quantized 8Bit BERT," in 2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS), Dec. 2019. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1109/emc2-nips53020.2019.00016

[6] X. Jiao et al., "TinyBERT: Distilling BERT for Natural Language Understanding," in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.18653/v1/2020.findings-emnlp.372

[7] G. Cohn, R. Agarwal, D. Gupta, and S. Patwardhan, "EELBERT: Tiny Models through Dynamic Embeddings," arXiv.org, Oct. 31, 2023. https://arxiv.org/abs/2310.20144

[8] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-Read Students Learn Better: On the Importance of Pre-training Compact Models," arXiv.org, Aug. 23, 2019. https://arxiv.org/abs/1908.08962

[9] K. Maity, A. T. Chaulwar, V. Vala, and R. S. Guntur, "NanoBERT: An Extremely Compact Language Model," in Proceedings of the 7th Joint International Conference on Data Science amp; Management of Data (11th ACM IKDD CODS and 29th COMAD), Jan. 2024. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1145/3632410.3632451

[10] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," arXiv.org, Jun. 17, 2021. https://arxiv.org/abs/2106.09685

[11] A. Desai, L. Chou, and A. Shrivastava, "Random Offset Block Embedding Array (ROBE) for CriteoTB Benchmark MLPerf DLRM Model: 1000× Compression and 3.1× Faster Inference," arXiv.org, Aug. 04, 2021. https://arxiv.org/abs/2108.02191

[12] A. A. Ginart, M. Naumov, D. Mudigere, J. Yang, and J. Zou, "Mixed Dimension Embeddings with Application to Memory-Efficient Recommendation Systems," in 2021 IEEE International Symposium on Information Theory (ISIT), Jul. 2021. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1109/isit45174.2021.9517710

[13] A. Aghajanyan, L. Zettlemoyer, and S. Gupta, "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning," arXiv.org, Dec. 22, 2020. http://arxiv.org/abs/2012.13255

[14] K. G. Larsen, R. Pagh, and J. Tětek, "CountSketches, Feature Hashing and the Median of Three," arXiv.org, Feb. 03, 2021. https://arxiv.org/abs/2102.02193

[15] Z. Zhang et al., "H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models," arXiv.org, Jun. 24, 2023. https://arxiv.org/abs/2306.14048

QuantumGame: A Game-Theoretic Approach to Excited State Computation on Quantum Computers

Jason Han
Department of Computer Science
Rice University
Houston, TX 77005
jason.han@rice.edu

Abstract

Computing the excited states of a molecule is intractable for large systems, but methods to do so on quantum computers scale tractably with the number of desired states to compute. This problem is equivalent to the PCA problem, for which distributed approaches have been developed. One of these is EigenGame, a game-theoretic approach to finding eigenvectors where each eigenvector reaches a Nash equilibrium in parallel. In this paper, we adapt the EigenGame objective function for use on quantum computers to explore a novel method of computing excited states. We show preliminary results that using the EigenGame objective allows us to converge to excited states sequentially, and we outline next steps to create a distributed algorithm.

1 Introduction

Understanding the excited states of a molecule is one of the main problems in modern electronic structure theory. Solving this problem on a classical computer would require large amounts of resources due to the exponential scaling of the Hilbert space. As a result, researchers have proposed using quantum computers, which can efficiently represent and transform quantum states [2] [8]. The potential to compute excited states on near-term, noisy quantum computers has yielded much research in this area in recent years [6].

One of the most prominent algorithms employed in quantum chemistry problems for estimating the ground state energy of a molecule is the Variational Quantum Eigensolver (VQE) [9]. It is based on the variational principle in quantum mechanics, which states that given a Hamiltonian $\hat{H}$, the ground state energy E_0 is always upper bounded by the expectation of $\hat{H}$ with respect to a trial wavefunction $|\psi\rangle$:

$$E_0 \leq \langle\psi|\hat{H}|\psi\rangle \tag{1}$$

As a result, we can find E_0 by finding the $|\psi(\theta)\rangle$ that minimizes $\langle\psi(\theta)|\hat{H}|\psi(\theta)\rangle$:

$$E_0 \approx \min_{\theta}\langle\psi(\theta)|\hat{H}|\psi(\theta)\rangle \tag{2}$$

This variational approach can be extended to compute the k-lowest eigenstates of a Hamiltonian $\hat{H}$, which are the k-lowest excited states. In a paper titled "Variational Quantum Computation of Excited States" by Higgott et al. [5], the authors propose a variational approach to excited state computation

Preprint. Under review.

based on the idea of eigenvalue deflation. As computing the excited states is equivalent to finding the eigenvalues and eigenvectors of $\hat{H}$, they do so by iteratively using the standard VQE objective function as in equation 2, with an additional penalty term to encourage orthogonality of excited states:

$$E_k \approx \min_{\theta_k} \langle \psi(\theta_k) | \hat{H} | \psi(\theta_k) \rangle + \sum_{i=1}^{k-1} \beta_i |\langle \psi(\theta_k) | \psi(\theta_i) \rangle|^2 \tag{3}$$

Although this approach has strong theoretical backing and can achieve high accuracy, one drawback is that this approach is sequential (other sequential methods have also been proposed to compute excited states, such as Orthogonal State Reduction VQE by Xie et al. [10]). As a result, researchers have explored attempts to parallelize computation of excited states, one of which being the Subspace-Search VQE (SSVQE) method by Nakanishi et al. [7] which attempts to find the subspace spanned by the K-lowest eigenstates. To do so, a unitary is optimized to minimize the *average* energy of the resulting states. In particular, the following cost function is minimized for a parameterized unitary $U(\theta)$ and initial orthogonal states $\psi_i, 1 \leq i \leq k$:

$$\min_{\theta} \sum_{i=1}^{k} \langle \psi_i | U^\dagger(\theta) \hat{H} U(\theta) | \psi_i \rangle \tag{4}$$

where the eigenstates are computed via another parameterized quantum circuit in the optimized eigenspace.

To replace the additional variational step that finds eigenvectors in the optimized eigenspace, a method called the Quantum Parallelized VQE (QP-VQE) by Hong et al. [6] uses a different cost function and embeds all desired excited states in one quantum circuit. This approach weights each eigenstate with $w_i > w_{i+1}$, which is bounded below by the weighted sum of the Eigenenergies [6]:

$$\min_{\theta} \sum_{i=1}^{k} w_i \langle \psi_i | U^\dagger(\theta) \hat{H} U(\theta) | \psi_i \rangle \geq \sum_{i=1}^{k} w_i E_i \tag{5}$$

based on the variational principle. This method optimizes a unitary acting on a mixed quantum state, which requires additional quantum resources for initial state preparation, ancilla qubits, as well as potentially higher depth ansatz due to the mixed quantum state being optimized.

In this work, our objective is to explore a parallelizable approach to computing excited states that limits the depth and number of qubits for quantum circuits in favor of more quantum computation on smaller circuits. In addition, we present a method for computing off-diagonal terms of a Hamiltonian on a quantum computer, which has not previously been explored and opens up a new way of approaching the problem of excited states computation.

2 Related Work

In this section, we will discuss a parallelizable, game-theoretic approach to classical PCA computation called EigenGame, and we will discuss potential advantages of viewing the excited states problem from a game-theoretic perspective.

As the problem of computing excited states of a Hamiltonian $\hat{H}$ by solving the Schrödinger equation is equivalent to finding the top k eigenvalues of $\hat{H}$, it is useful to explore classical techniques for diagonalizing matrices and apply these insights to the excited states problem. Notably, one of the most common methods for computing the top k eigenvalues of a square matrix M is the power deflation method, which iteratively computes the excited states by finding the next highest eigenvalue after the previous ones have been removed. A direct analog is applied to quantum computation for excited states, as seen in the Variational Quantum Deflation algorithm in equation 3 [5].

Another approach that has garnered attention due to its novel perspective on the PCA problem is a paper titled "EigenGame: PCA as a Nash Equilibrium" by Gemp et al. [3]. Instead of viewing

eigenvalue computation from a deflation perspective, the authors approach the problem from a game theoretic perspective.

To motivate the intuition behind the objective function in EigenGame, we briefly review the eigenvalue problem underlying PCA. Given a symmetric matrix $M \in \mathbb{R}^{d\times d}$, our goal is to find a matrix of eigenvectors $V \in \mathbb{R}^{d\times d}$ such that $MV = V\Lambda$ where $\Lambda \in \mathbb{R}^{d\times d}$ is diagonal. As V is orthonormal, we can observe that:

$$V^T MV = V^T V\Lambda = \Lambda \tag{6}$$

We define $\hat{V}$ to be our approximation of the eigenvectors V and define a matrix $R(\hat{V}) = \hat{V}^T M\hat{V}$. Each player wants to maximize its Rayleigh quotient $R_{ii} = \langle \hat{v}, M\hat{v}\rangle$ and minimize the off diagonal terms of R ($R_{ij} = \langle \hat{v}_i, M\hat{v}_j\rangle$), so that R is diagonal and approaches Λ (note that $R(V) = \Lambda$ by equation 6).

Thus, if we let each approximate eigenvector $\hat{v}_i \in \mathbb{R}^d$ be a player in the game, where $\hat{v}_i$ denotes the i-th largest eigenvector, we can define their utility function as follows:

$$\max_{\langle \hat{v}_i, \hat{v}_i\rangle = 1} u_i(\hat{v}_i|\hat{v}_{j<i}) = \underbrace{\langle \hat{v}_i, M\hat{v}_i\rangle}_{R_{ii}} - \underbrace{\sum_{j<i} \frac{\langle \hat{v}_i, M\hat{v}_j\rangle^2}{\langle \hat{v}_j, M\hat{v}_j\rangle}}_{\frac{R_{ij}^2}{R_{jj}}} \tag{7}$$

where we divide use R_{ij}^2 to get a nonnegative penalty for the off-diagonal terms and divide by R_{jj} for normalization of the penalty. The authors argue that the orthogonality constraint is satisfied by the term $\frac{\langle \hat{v}_i, M\hat{v}_j\rangle^2}{\langle \hat{v}_j, M\hat{v}_j\rangle}$ as if $\hat{v}_i, \hat{v}_j$ are eigenvectors, then $\langle \hat{v}_i, M\hat{v}_j\rangle = \langle \hat{v}_i, \lambda_j\hat{v}_j\rangle = \lambda_j\langle \hat{v}_i, \hat{v}_j\rangle = 0$. The authors showed that the top k eigenvectors form the unique solution for the game described in equation 7. By taking the gradient of the utility function $u_i(\hat{v}_i|\hat{v}_{j<i})$ with respect to $\hat{v}_i$, the authors describe both a sequential and distributed algorithm for updating the eigenvectors and show convergence of the sequential algorithm.

One advantage of the EigenGame algorithm over power deflation is its potential for parallelization. As the authors observed that the players become effectively stationary as they approach the eigenvectors, a distributed approach holds the potential for further speedup. Furthermore, if there is noise in computation of the objective function, EigenGame has the potential to be more robust as each player is constantly adjusting to converge to an eigenvector, whereas power deflation accumulates errors from one iteration to the next.

3 Methods

3.1 Quantum EigenGame Formulation

These advantages of EigenGame over classical deflation motivate us to explore applying EigenGame to the excited states problem. Specifically, EigenGame's parallelizability holds the potential to improve upon existing methods for calculating excited states on quantum computers as mentioned in section 1. In this section, we will motivate a novel approach to computing excited states, which we call *QuantumGame*.

To formulate the quantum problem, one key observation is that we are working in a complex Hilbert space, where the Hamiltonian $\hat{H} \in \mathbb{C}^{d\times d}$ and vectors $|\psi\rangle \in \mathbb{C}^d$. We can use much of the same theoretical basis as in the EigenGame problem to motivate an objective function by finding analogs in the complex space. Specifically, we need the utility function to be real-valued (so our classical optimizer can operate on real vector spaces). To do so, we take advantage of the following observations: the Hamiltonian $\hat{H}$ is Hermitian (meaning $\hat{H} = \hat{H}^\dagger$), so $\langle\psi|\hat{H}|\psi\rangle \in \mathbb{R}$. So, $\hat{H}$ is analogous to being symmetric in the real vector space case, and we let $M \leftarrow \hat{H}$ in our Quantum EigenGame problem. However, to use the EigenGame utility function as described in equation 7, we

need to compute $\langle\phi|\hat{H}|\psi\rangle$ where $|\phi\rangle \neq |\psi\rangle, |\phi\rangle, |\psi\rangle \in \mathbb{C}^d$. As $\langle\phi|\hat{H}|\psi\rangle \in \mathbb{C}$, we propose making this value real-valued by simply taking its magnitude: $\langle\phi|\hat{H}|\psi\rangle \leftarrow |\langle\phi|\hat{H}|\psi\rangle|$.

We now define an objective function for the quantum analog to the EigenGame utility function seen in equation 7, where we minimize our objective to find the k-lowest eigenstates. We denote $|\hat{\psi_i}\rangle \in \mathbb{C}^{d\times d}$ to be candidate i-th smallest eigenvector, and $\langle\hat{\psi_i}| = |\hat{\psi_i}\rangle^\dagger$ (using standard bra-ket notation):

$$\min_{\langle\hat{\psi_i}|\hat{\psi_i}\rangle=1} \tilde{u}_i(|\hat{\psi_i}\rangle, |\hat{\psi_{j<i}}\rangle) = \langle\hat{\psi_i}|\hat{H}|\hat{\psi_i}\rangle - \sum_{j<i} \frac{|\langle\hat{\psi_i}|\hat{H}|\hat{\psi_j}\rangle|^2}{\langle\hat{\psi_j}|\hat{H}|\hat{\psi_j}\rangle} \tag{8}$$

In the EigenGame paper, they also take the gradient of the utility function to determine how to update each vector. Although we can perform a similar step here, we have not yet figured out how to implement this step in quantum computers because it is unclear how to perform scalar multiplication and vector addition due to norm-preserving properties in quantum mechanics. It is worth noting that, if we can implement a well-defined update to the candidate eigenstates, then we may be able to prove convergence for our algorithm, which no algorithm run on quantum computers currently has due to the complicated landscape of these objective functions based on a given Hamiltonian $\hat{H}$.

3.2 Implementation on Quantum Computers

To implement an algorithm on solving this problem on quantum computers, we adopt a hybrid-classical variational approach as seen in every other near-term quantum algorithm for computing excited states. A diagram of our method can be found in Figure In particular, we will run the inner product calculations seen in the objective function 8 on quantum machines, and combine these real-valued calculations with a classical optimizer on a classical machine.

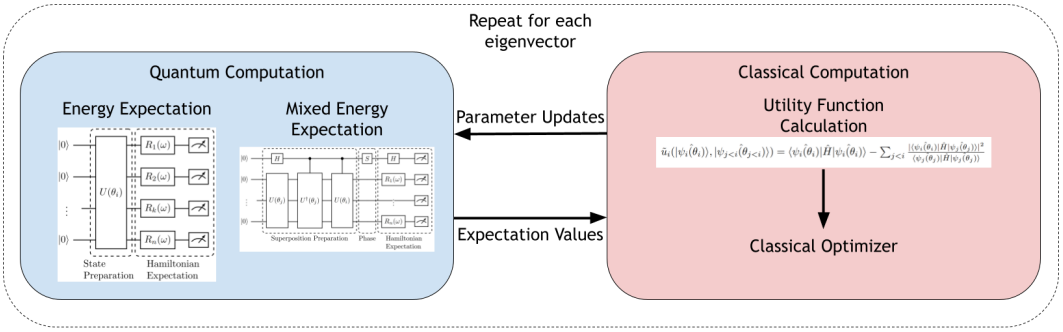

Figure 1: We propose a hybrid-quantum classical algorithm for the excited states problem using a variant of the objective function from EigenGame. For each eigenvector, we calculate two quantities on quantum machines: the energy expectation for that quantum state, as well as the 'mixed' energy expectation of the current eigenstate with respect to the other candidate eigenstates. We then combine these values to form our utility cost function, which our classical optimizer minimizes to update the parameters θ_i for each eigenvector $|\psi_i(\hat{\theta}_i)\rangle$.

To compute terms of the form $\langle\hat{\psi_i}|\hat{H}|\hat{\psi_i}\rangle$, we can follow the standard approach of taking the expectation value of the Hamiltonian with respect to the prepared quantum state $|\psi_i\rangle$. To compute terms of the form $\langle\hat{\psi_i}|\hat{H}|\hat{\psi_j}\rangle$, however, no clear quantum algorithm has been proposed. Drawing inspiration from the SWAP test, it is the job of our work to propose a novel quantum algorithm for computing $\langle\hat{\psi_i}|\hat{H}|\hat{\psi_j}\rangle$ [1]. The quantum circuit computing this value can be found in Fig. 2.

3.3 Mixed Energy Expectation Computation

We will now elaborate in detail the math behind how this quantum circuit computes the value $\langle\phi|\hat{H}|\psi\rangle$. We can run two versions of this circuit, one with the S gate and one without, to compute $Re(\langle\phi|\hat{H}|\psi\rangle)$ and $Im(\langle\phi|\hat{H}|\psi\rangle)$, respectively.

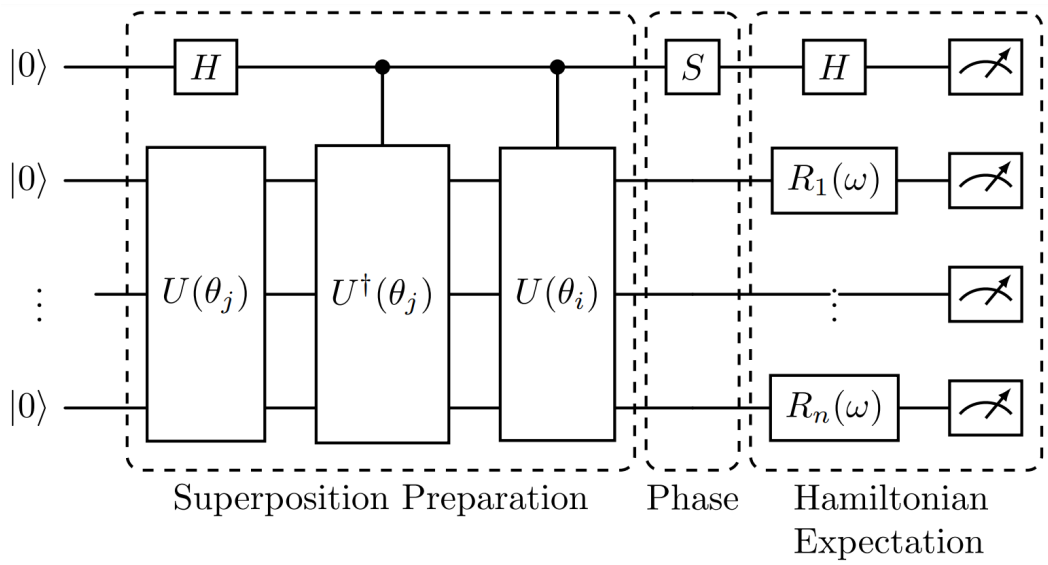

Figure 2: We propose a novel quantum circuit for computing mixed terms of the form $\langle\phi|\hat{H}|\psi\rangle$, where $|\phi\rangle = U(\theta_i)|0\rangle$ and $|\psi\rangle = U(\theta_j)|0\rangle$. Our method uses quantum superposition and interference to extract this mixed term, and consists of three steps. We first create a superposition of $|\psi\rangle$ and $|\phi\rangle$, and add a phase to the qubit depending on whether we want to measure the imaginary component of $\langle\phi|\hat{H}|\psi\rangle$. We then compute the expectation value of our Hamiltonian with our ancilla qubit, which interferes the quantum states.

Assuming that $|\psi\rangle, |\phi\rangle \in \mathbb{C}^d$ where $|\phi\rangle = U(\theta_i)|0\rangle$ and $|\psi\rangle = U(\theta_j)|0\rangle$, after the 'Superposition Preparation' in Fig. 2, if we let $\mathbf{\Psi}$ represent the state of our quantum system, then $\mathbf{\Psi}$ can be written as:

$$\mathbf{\Psi} = \frac{1}{\sqrt{2}}(|\psi 0\rangle + |\phi 1\rangle) \tag{9}$$

We then apply the phase gate S to $\mathbf{\Psi}$ depending on whether we want to measure the imaginary component of $\langle\phi|\hat{H}|\psi\rangle$. We will proceed our analysis without applying the S gate to compute $Re(\langle\phi|\hat{H}|\psi\rangle)$, although the analysis will proceed symmetrically if the S gate is applied to compute $Im(\langle\phi|\hat{H}|\psi\rangle)$.

We then apply the H gate again to our quantum system, putting it in the state:

$$\mathbf{\Psi} = \frac{1}{2}(|\psi 0\rangle + |\psi 1\rangle + |\phi 0\rangle - |\phi 1\rangle) \tag{10}$$

We then compute the expectation value of our Hamiltonian, measuring the effects of our ancilla:

$$\langle \mathbf{\Psi} | \hat{H} \otimes Z | \mathbf{\Psi} \rangle = \frac{1}{4} \langle (|\psi 0\rangle + |\psi 1\rangle + |\phi 0\rangle - |\phi 1\rangle) | \hat{H} \otimes Z | (|\psi 0\rangle + |\psi 1\rangle + |\phi 0\rangle - |\phi 1\rangle) \rangle \quad (11)$$

$$= \frac{1}{4} (\langle \psi | \hat{H} | \psi \rangle + \langle \phi | \hat{H} | \psi \rangle - \langle \psi | \hat{H} | \psi \rangle + \langle \phi | \hat{H} | \psi \rangle + \langle \psi | \hat{H} | \phi \rangle + \langle \phi | \hat{H} | \phi \rangle \quad (12)$$

$$+ \langle \psi | \hat{H} | \phi \rangle - \langle \phi | \hat{H} | \phi \rangle) \quad (13)$$

$$= \frac{1}{4} (2\langle \phi | \hat{H} | \psi \rangle + 2\langle \psi | \hat{H} | \phi \rangle) \quad (14)$$

$$= \frac{1}{4} (4Re(\langle \phi | \hat{H} | \psi \rangle)) \quad (15)$$

$$= Re(\langle \phi | \hat{H} | \psi \rangle) \quad (16)$$

as desired.

We now are able to compute all the values in the QuantumGame utility function 8, and add these values classically and use a classical optimizer (in this work, we used the Sequential Least Squares Linear Programming, or SLSQP, optimizer) to minimize the utility function. We would like to note that other choices of classical optimizers and quantum circuit structures to achieve the desired result may yield better results, but this is ongoing work and we focus on presenting the ideas behind our work in this paper.

4 Results

We performed preliminary testing of our method on the H_2 molecule to compute the smallest 3 eigenvalues of its corresponding Hamiltonian. We use a two-qubit quantum circuit exhibiting linear entanglement with 8 parameters, and use the Sequential Least Squares Linear Programming (SLSQP) optimizer, which is effective in achieving orthogonality constraints and computing gradients for fast convergence. We plot the convergence of QuantumGame compared to Variational Quantum Deflation (VQD), which is a standard algorithm for computing excited states sequentially, for the lowest 3 eigenstates in Fig. 3.

We see that QuantumGame achieves a faster convergence compared to VQD in terms of number of calls to the objective function (392 versus 647 calls to the objective function). QuantumGame and VQD exhibit identical convergence for the first eigenvalue, which is expected because, for the first eigenvalue, the optimization problem is identical. However, for states 2 and 3, we see that QuantumGame achieves a faster convergence for both states. For state 2, we see that QuantumGame effectively minimizes the expectation value without increasing the objective function, allowing it to converge at a faster rate. For state 3, although QuantumGame 'overshoots' by finding a state that minimizes the expectation energy too much, we see that it very effectively factors in the 'off-diagonal' penalty found in equation 8 to find the third eigenvalue faster than VQD. In contrast, VQD has a slower convergence for the third eigenvalue, where it slowly converges to the third eigenvalue.

These results are promising to show how QuantumGame can improve upon VQD, but an important consideration is the amount of quantum resources used in QuantumGame versus in VQD, which we will cover in the next section.

5 Discussion

Based on our current implementation, for the i-th vector in QuantumGame, we have to run expectation values of the Hamiltonian for i quantum circuits (computing the expectation of the Hamiltonian for the state $|\hat{\psi}_i\rangle$ and $|\hat{\psi}_{j<i}\rangle$), whereas VQD only requires one Hamiltonian expectation value. This is an additional cost we must pay for QuantumGame because we are utilizing additional information from $\hat{H}$ to guide the optimizer in the objective function. Based on the preliminary experiment that we ran, we see that incorporating $\hat{H}$ into the objective function appears to be beneficial, but additional experimental testing needs to be performed to see whether or not the tradeoff in calls to the objective function versus incorporating the Hamiltonian in the objective function is worth it. Additionally, as mentioned in 3.3, computing the value $\langle \hat{\psi}_i | \hat{H} | \hat{\psi}_j \rangle$ requires two quantum circuits to be run (to

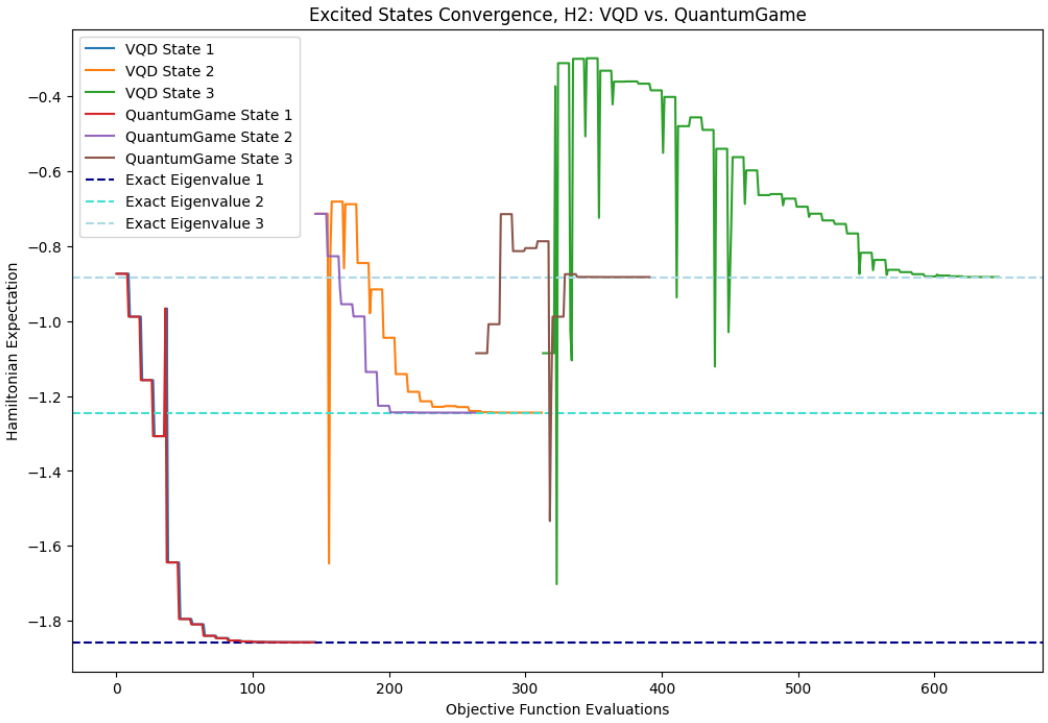

Figure 3: We plot the convergence of Variational Quantum Deflation (VQD) and QuantumGame based on the number of calls to the objective function. We see that QuantumGame achieves a faster convergence than VQD.

compute the real and imaginary components of the value), but assuming that the sizes of the quantum circuits are not too large, this cost should not be too significant because these circuits can be run in parallel, and there exist smaller quantum devices which we can use to parallelize this computation.

Regarding the depth of the circuit for mixed energy expectation calculation as seen in Fig. 2, this circuit does have a higher depth than the quantum circuit used to calculate $\langle\hat{\psi}_i|\hat{\psi}_j\rangle$ used in VQD, especially because the controlled unitary gates may be difficult to implement for general unitaries. In VQD, they use a compute-uncompute method, which applies the state preparation gates for ψ_j first followed by the adjoint of each gate used to prepare ψ_i applied in reverse order), but we note that an alternative circuit can be used for mixed energy expectation calculation which contains twice as many qubits and uses conditional SWAP gates instead to superimpose the qubits [5] [4]. Depending on whether or not additional qubits are available on the quantum machine, the alternative circuit can be employed which would have a strictly lower depth than the Compute-Uncompute method. Constructing a shallow circuit that efficiently computes this mixed energy expectation is a clear next step for this work. The other differences of the QuantumGame approach outlined in Fig. 1 compared to VQD have a constant-time impact.

Overall, our preliminary results are promising to show both the convergence of the QuantumGame approach and its potential to achieve faster convergence than VQD. Our current research efforts are focused on the following question: another advantage of EigenGame is that it can compute eigenvalues in *parallel*, so can we leverage that strength for QuantumGame as well? Preliminary experiments do not show faster convergence when eigenvalues are computed in parallel (the optimization appears to be fruitless until the previous eigenvectors converge), and we are currently performing additional work in this area to explore parallelization of QuantumGame.

6 Conclusion and Future Work

In this work, we presented a novel approach to the problem of computing excited states on a quantum computer by drawing inspiration from a game-theoretic formulation to the PCA problem proposed in the EigenGame paper. Preliminary results suggest effective convergence of EigenGame, and our work provides additional research questions to be explored:

Parallel Computation of Excited States. As was mentioned in Sec. 5, QuantumGame holds the potential for parallelization just as EigenGame does. The question we ask is, "When can we achieve an 'approximate enough' eigenvector so that future candidate eigenvectors can begin productive optimization?" We can attempt delayed starts, detection of "barren plateau's", or assigning higher weights to eigenvectors which have run for more iterations (and likely are closer to the desired eigenvector).

Accuracy. How accurate is QuantumGame compared to other methods of computing excited states? Rigorous benchmarking for multiple methods on a diverse set of molecules would be required to create more robust analysis and draw better insights on the behavior of QuantumGame, and how we might improve QuantumGame based on its shortcomings.

Quantum Resource Cost. One cost of the current implementation of QuantumGame is the additional quantum resources needed for computing the objective function compared to existing quantum algorithms. Additional quantum algorithm design may help reduce the quantum resource cost of our method while still ensuring its accuracy.

Mixed Energy Expectation. One novel contribution of our work that has not been seen in other works (to our knowledge) is the incorporation of the Hamiltonian in the penalty of the objective function. As we saw in our preliminary results, incorporating the Hamiltonian may lead to an objective function that can be optimized faster due to the additional information about the problem that we have. This new perspective opens up additional work for exploring how the Hamiltonian can be used as a penalty in the objective function to obtain more information about the problem, as well as weighing tradeoffs to see if this choice is worth the additional computational cost.

7 Acknowledgements

We would like to thank Professor Anastasios Kyrillidis for the idea for the project and his advice and mentorship throughout the project. We would also like to thank Professor Tirthak Patel for reviewing the ideas in the paper and sharing areas that can be improved. We used ChatGPT, an AI language model developed by OpenAI, for partial assistance in writing and for the research process, and all such texts were verified and edited for correctness.

References

[1] Harry Buhrman, Richard Cleve, John Watrous, and Ronald de Wolf. Quantum fingerprinting. *Physical Review Letters*, 87(16), September 2001. ISSN 1079-7114. doi: 10.1103/physrevlett.87.167902. URL http://dx.doi.org/10.1103/PhysRevLett.87.167902.

[2] J. Ignacio Cirac and Peter Zoller. Goals and opportunities in quantum simulation. *Nature Physics*, 8(4):264–266, April 2012. doi: 10.1038/nphys2275.

[3] Ian Gemp, Brian McWilliams, Claire Vernade, and Thore Graepel. Eigengame: Pca as a nash equilibrium, 2021.

[4] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, March 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-0980-2. URL http://dx.doi.org/10.1038/s41586-019-0980-2.

[5] Oscar Higgott, Daochen Wang, and Stephen Brierley. Variational quantum computation of excited states. *Quantum*, 3:156, July 2019. ISSN 2521-327X. doi: 10.22331/q-2019-07-01-156. URL http://dx.doi.org/10.22331/q-2019-07-01-156.

[6] Cheng-Lin Hong, Luis Colmenarez, Lexin Ding, Carlos L. Benavides-Riveros, and Christian Schilling. Quantum parallelized variational quantum eigensolvers for excited states, 2023.

[7] Ken M. Nakanishi, Kosuke Mitarai, and Keisuke Fujii. Subspace-search variational quantum eigensolver for excited states. *Physical Review Research*, 1(3), October 2019. ISSN 2643-1564. doi: 10.1103/physrevresearch.1.033062. URL http://dx.doi.org/10.1103/PhysRevResearch.1.033062.

[8] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, August 2018. ISSN 2521-327X. doi: 10.22331/q-2018-08-06-79. URL http://dx.doi.org/10.22331/q-2018-08-06-79.

[9] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. The variational quantum eigensolver: A review of methods and best practices. *Physics Reports*, 986:1–128, November 2022. ISSN 0370-1573. doi: 10.1016/j.physrep.2022.08.003. URL http://dx.doi.org/10.1016/j.physrep.2022.08.003.

[10] Qing-Xing Xie, Sheng Liu, and Yan Zhao. Orthogonal state reduction variational eigensolver for the excited-state calculations on quantum computers. *Journal of chemical theory and computation*, 18(6), June 2022. doi: https://doi.org/10.1021/acs.jctc.2c00159. URL https://pubmed.ncbi.nlm.nih.gov/35621354/.

Sequential Low-Rank Recovery

Mahtab Vandchali
Computer Science
Rice University
ma202@rice.edu

Alireza Azizi
Electrical and Computer Engineering
Rice University
aa152@rice.edu

Abstract

The evolution of natural language processing (NLP) is increasingly characterized by the use of large-scale pre-training on general domain data, followed by fine-tuning for specific tasks. However, as models become larger, the traditional approach of fine-tuning all parameters becomes computationally infeasible. To address this challenge, Low-Rank Adaptation (LoRA) is introduced. This method retains the original weights of pre-trained models and integrates trainable low-rank matrices within each Transformer layer, significantly reducing the number of parameters that need adjustment during fine-tuning.

While LoRA has proven effective for fine-tuning, its potential in the pre-training phase is less understood. A novel method that extends LoRA to the pre-training process is introduced, detailing the unique challenges and limitations encountered. This novel solution is LoRA-the-Explorer (LTE), which utilizes a bi-level optimization algorithm to facilitate parallel training of multiple low-rank structures across different computing nodes, enhancing the scalability and efficiency of model training.

In addition to these innovations in model training, we also tackle the issue of communication overhead in distributed training environments. We introduce PUFFERFISH, a framework that employs compressed stochastic gradients, achieved through methods such as sparsification or quantization, to reduce the data transmitted during training. PUFFERFISH not only lowers communication costs but also avoids additional computational burdens typically associated with gradient compression, maintaining the accuracy of state-of-the-art models. This framework is designed for easy integration into existing deep learning platforms, requiring minimal modifications for implementation.

Collectively, these advancements—LoRA, LTE, and PUFFERFISH—represent significant steps forward in optimizing both the efficiency and effectiveness of training large-scale NLP models, addressing key challenges in model scalability, training speed, and operational overhead.

1 Introduction

1.1 Low-Rank Adaptation (LoRA)

In the field of natural language processing (NLP), a common practice involves using a large-scale pre-trained language model for various downstream applications. Typically, adapting these models to specific tasks is done through fine-tuning, where all the model's parameters are updated. This method, though straightforward, means that each adapted model retains the same number of parameters as the original, leading to high storage and computation costs.

To address this issue, many researchers have started exploring ways to only adjust some of the model's parameters or to add external modules tailored to new tasks. This approach significantly

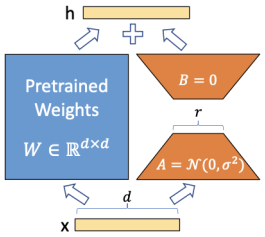

Figure 1: This is how LoRA reparametrization works. Only A and B are trained.

reduces the number of parameters that need to be stored and managed for each specific task, thereby enhancing operational efficiency. However, these methods often introduce delays during model inference and sometimes do not achieve the same level of performance as the traditional fine-tuning method, presenting a trade-off between efficiency and effectiveness.

Inspired by the work of Li et al. (2018a) and Aghajanyan et al. (2020), which indicated that models with a large number of parameters actually function within a surprisingly low intrinsic dimension, we developed a new method called Low-Rank Adaptation (LoRA). Our approach is based on the hypothesis that changes in model weights during adaptation exhibit a low "intrinsic rank." LoRA specifically allows us to indirectly train some dense layers of a neural network by optimizing low-rank matrices that represent these changes, all while keeping the pre-trained weights fixed.

One of the major benefits of using LoRA is its operational efficiency. It allows us to use a single shared pre-trained model to create many smaller, task-specific LoRA modules. By freezing the shared model, we can quickly switch between tasks by simply replacing the specific matrices used for each task, as shown in Figure 1. This method significantly cuts down on storage needs and reduces the overhead associated with switching tasks, potentially making the deployment of adaptive NLP models much more practical in real-world settings.

1.2 LoRA The Explorer (LTE)

LoRA-The-Explorer expands the usage of the low-rank adapters for pre-training and would be a good competitor to standard pre-training from scratch. This can bring many advantages to the table, like lower memory requirements and lower communication between the contributing nodes, which is appealing for band-limited frameworks. These positive points make it possible for a single consumer-grade GPU to overcome the computational complexities of training a large deep-learning model. LoRA The Explorer leverages both data and model parallelism by training on different shards of the data distribution and by storing different copies of the LoRA parameters.

A single LoRA head cannot achieve the performance of standard optimization algorithms for pre-training unless it has a rank of $r = \min\{m, n\}$ where m and n are dimensions of the original weight matrix. The drawback of $r = \min\{m, n\}$ is that it increases the memory consumption. While the case of $r \ll \min\{m, n\}$ can help to decrease required memory to $\mathcal{O}(r(m + n))$ compared to $\mathcal{O}(mn)$ for the standard model, setting r to $\min\{m, n\}$ ends up to a worse scenario for memory usage and compromises the memory efficiency. This observation implies the need for multiple Low-Rank Adapters to achieve the performance of standard pre-training and assure memory efficiency. High-rank matrices can be decomposed into a linear combination of multiple low-rank matrices. A simple illustration of that is the singular value decomposition, which transforms a given matrix into the summation of rank-1 matrices formed by the outer product of the right and left singular vectors. This intuition forms the basis for multi-head LoRA parameterization where each head estimates a low-rank model, then all the low-rank weights can be added to make a higher rank estimate for the pre-training problem.

1.3 PUFFERFISH:

In the realm of modern machine learning, distributed model training, particularly data parallel training, has become essential for achieving significant speed enhancements in various applications. This

method allows a model to be trained simultaneously across multiple computing nodes, theoretically promising substantial reductions in training time. However, the actual performance improvements often do not meet these ideal expectations, primarily due to communication overheads that become more pronounced as model complexity increases.

A major bottleneck in this process arises from the frequent need to update and transmit gradients—the changes computed for model parameters after each batch of data is processed—across these nodes. As state-of-the-art models now include hundreds of billions of parameters, the data volume for these updates is massive. To tackle these challenges, recent research has focused on gradient compression techniques, such as low-precision training and sparsification, which aim to reduce the amount of data that needs to be communicated.

Despite these advancements, gradient compression often introduces new issues, such as increased computational demands for compressing the gradients, insufficient utilization of gradient data, and the need for extensive modifications to integrate these techniques into existing deep learning frameworks efficiently. Given these complications, our study proposes integrating gradient compression directly into the model's architecture, potentially simplifying the training process while retaining communication efficiency. We explore this by initially training a standard, full-rank model for a fraction of the total training epochs and then converting it to a more manageable, low-rank format using Singular Value Decomposition (SVD). This approach not only aims to maintain model accuracy but also reduces the overhead associated with traditional gradient compression methods, offering a promising avenue for efficient and effective distributed training.

2 Literature Review

2.1 Low-Rank Adaptation (LoRA)

We discuss the straightforward design of Low-Rank Adaptation (LoRA) and its practical advantages, which are applicable to any dense layers in deep learning models.

In deep learning, neural networks consist of numerous dense layers that perform matrix multiplication using fully-ranked weight matrices. According to research by Aghajanyan et al. (2020), pre-trained language models can still effectively learn despite being reduced to a lower-dimensional subspace, indicating a low "intrinsic dimension." Building on this concept, we hypothesize that during model adaptation, the weight updates also exhibit a low "intrinsic rank."

To implement this, for a pre-trained weight matrix W_0 in a space $\mathbb{R}^{d\times k}$, we restrict its updates using a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank r is significantly smaller than d and k. During the training phase, W_0 remains fixed, not updated by gradients, whereas A and B are trainable. Both W_0 and $\Delta W = BA$ interact with the same input, and their outputs are added together to produce the final output h. Specifically, the output is calculated as:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

We depict this process of reparameterization in Figure 1. Initially, A is set using a random Gaussian distribution, and B is initialized to zero, making $\Delta W = BA$ start at zero at the beginning of training. The scaling factor $\Delta W x$ is adjusted by αr, where α remains constant. In terms of optimization, particularly with the Adam optimizer, adjusting α is akin to modifying the learning rate when the initialization is scaled appropriately. Therefore, we typically set α to the first value of r tested and do not further adjust it. This approach of scaling minimizes the necessity to recalibrate other training settings when r is changed, according to Yang & Hu (2021).

In essence, LoRA introduces a method to optimize neural network training by managing the complexity of weight adjustments, making the training process more efficient without sacrificing model performance.

2.2 LoRA The Explorer (LTE)

LoRA-The-Explorer's advantages include reduced memory demands and communication overhead between nodes, which is particularly beneficial for bandwidth-limited frameworks. As stated in the introduction, a single Low-Rank Adapter suffers from a rank deficiency for pre-training. This suggests the novel idea of using multiple parallel heads to compensate for the rank deficiency. Each

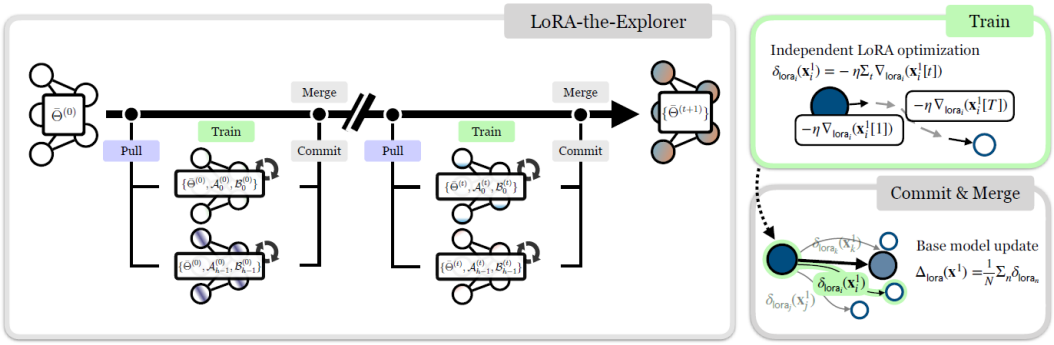

Figure 2: LTE diagram and its 3 steps: (1) train: each head is trained independently for T iterations on different mini-batches from the same distribution, (2) commit and merge: averaging the individual LoRA updates after aggregation, (3) LTE: applying updates to main weights and resetting the matrix B, then repeating the optimization for a new iteration with new LoRA parameters

head will be provided with a separate subset of the train data with the same statistical distribution and try to learn a low-rank estimate of the parameters with its own initialization. Then, the optimization results from each head are gathered and merged to update the whole model in a fashion similar to federated learning systems. The mathematical details of this method will be stated in the following subsection on Multi-head LoRA (MHLoRA).

2.2.1 Multihead LoRA

Multi-head LoRA reparameterizes the full-rank weights into a linear combination of N low-rank weight matrices denoted by B_n and A_n.

$$f_{MHLoRA}(x) = Wx + \frac{s}{N}\sum_{n=1}^{N} B_n A_n x$$

The parallel LoRA heads provide low-rank weights that are periodically merged into the full weights with simple linear combinations. We can trivially show that the dynamics of a single parallel LoRA head can estimate the direction of one step of MHLoRA by rewriting the minimization problem in a different way:

$$\underset{B_n, A_n}{\text{argmin}}\mathcal{L}\left(W + \tfrac{s}{N}\sum_{n=1}^{N} B_n A_n\right) \equiv \underset{\hat{B}_n, \hat{A}_n}{\text{argmin}}\mathcal{L}\left(\hat{W} + \tfrac{s}{N}\hat{B}_n\hat{A}_n\right)$$

when $\hat{W} = W + \frac{s}{N}\sum_{j\neq n} B_j A_j$, $\hat{W}$ accumulates all the information of the LoRA parameters at every iteration. So the single node $\hat{B}_n\hat{A}_n$ can approximate the direction a single step of the MHLoRA. This simple intuition illustrates that the exact gradient updates of the Multihead LoRA are recoverable, and since the gradient's rank tends to increase through the training, MHLoRA would be able to estimate higher rank updates compared to a single head.

2.2.2 LTE algorithm

In this section, we will explain the LoRA The Explorer algorithmically. In the training step, each LoRA head performs a gradient descent algorithm to minimize the objective function on its local data sampled from a similar distribution to the data of other participating nodes. This results to an overall update form of $\delta_{lora_n}(x) = -\eta\sum_t \nabla_{lora_n}(x[t])$ for each head. Then, In the commit and merge step, all of these updates are aggregated and averaged to update the whole model with $\Delta_{lora}(x) = \frac{1}{N}\sum_n \delta_{lora_n}(x)$. The updated whole weights are then sent to all the contributing heads to update their initialization and the same procedure repeats until convergence.

Algorithm 1 LoRA The Explorer(LTE)

Input: Dataset $\mathcal{D}_{train}$, model $\mathcal{F}$, loss function $\mathcal{L}$, parameters $\theta = \{W_0, \cdots, W_L\}$, merge scalar s, number of workers N, merge iterations T

while not converged **do**
  optional: quantize θ, Keep high precision copy
  for each worker n **do** (**in parallel**)
    if LoRA not initialized **then**
      $B_n, A_n \leftarrow LoRA\_parameterize(\mathcal{F})$
    else
      (optional) reset parameter $\mathcal{B}_n$ to zero
      Optimize $\mathcal{B}_n, \mathcal{A}_n$ for T iterations by minimizing $\mathbb{E}_{x,y\sim\mathcal{D}_{train}}[\mathcal{L}(\mathcal{F}(x), y)]$
    end if
    Synchronize by Communicating LoRA parameters
  end for
  for each worker n **do**
    for B_n, A_n in $\mathcal{B}_n, \mathcal{A}_n$ **do**
      Merge LoRA params $W_n \leftarrow W_n + \frac{s}{N} B_n A_n$
    end for
  end for
end while

2.3 PUFFERFISH

In this study, we introduce PUFFERFISH, a framework designed for efficient computation and communication in distributed training. PUFFERFISH adapts any deep neural network architecture into a pre-factorized low-rank format and trains this modified network to enhance both computational and communication efficiencies, eliminating the need for direct gradient compression. However, we have found that training these pre-factorized low-rank networks directly can lead to significant accuracy losses, particularly in large-scale machine learning applications. To address these losses, we have developed two strategies: (i) implementing a hybrid architecture and (ii) employing a technique known as vanilla warm-up training.

Hybrid network architecture. In the PUFFERFISH framework, low-rank factorization is employed to approximate the original weights of a neural network, denoted by $W_l \approx U_l V_l^\top$ for each layer l. This technique, however, introduces approximation errors which may accumulate and propagate from earlier to later layers, potentially affecting the overall model accuracy. To counteract this, the strategy adopted involves factorizing only the later layers of the network. This approach is particularly effective in Convolutional Neural Networks (CNNs), where the bulk of parameters is often located in the later layers, thereby allowing substantial model compression without significant accuracy loss.

In a network with L layers, rather than factorizing all layers, the first $K-1$ layers are kept intact, with only layers from K onward being factorized, where K is a hyper-parameter that helps balance compression efficiency against model accuracy. The choice of K is tuned for each model to optimize performance. Experimental results have shown that such a hybrid architecture can mitigate the loss in test accuracy, for instance, approximately 0.6% for a modified VGG-19 model with $K = 9$, demonstrating the efficacy of this approach in maintaining a favorable balance between model size and accuracy.

Vanilla warm-up training. Vanilla warm-up training is recognized as essential due to its significant impact on the final accuracy of a model. Research has shown that early training phases are crucial, and adjustments like sparsifying gradients or factorizing weights too early can permanently harm model accuracy. To counteract potential accuracy losses from early modifications, this work introduces a method called "vanilla warm-up training." In this approach, the network is initially trained in its full, unmodified state for a few epochs. After this initial phase, the model weights are then factorized using truncated Singular Value Decomposition (SVD) to initialize a low-rank version of the network. This technique leverages the stability of the early trained full-rank model to ensure a more reliable and accurate foundation for the subsequent, more compressed training phases.

Algorithm 2 PUFFERFISH Training Procedure

Require: Randomly initialized weights of vanilla N-layer architectures $\{W_1, W_2, \ldots, W_L\}$, and the associated weights of hybrid N-layer architecture $\{W_1, W_2, \ldots, W_{K-1}, U_K, V_K^\top, \ldots, U_L, V_L^\top\}$, the entire training epochs E, the vanilla warm-up training epochs E_{wu}, and learning rate schedule $\{\eta_t\}_{t=1}^{E}$

Ensure: Trained hybrid L-layer architecture weights $\{\hat{W}_1, \hat{W}_2, \ldots, \hat{W}_{K-1}, \hat{U}_K, \hat{V}_K^\top, \ldots, \hat{U}_L, \hat{V}_L^\top\}$

for $t = 1, \ldots, E_{wu}$ **do**
    Train $\{W_1, W_2, \ldots, W_L\}$ with learning rate schedule $\{\eta_t\}_{t=1}^{E_{wu}}$ ▷ vanilla warm-up training
end for
for $l = 1, \ldots, L$ **do**
    if $l < K$ **then**
        Copy the partially trained W_l weight to the hybrid network
    else
        $\tilde{U}_l \tilde{\Sigma}_l \tilde{V}_l^\top = \text{SVD}(W_l)$ ▷ Decomposing the vanilla warm-up trained weights

        $U_l = \tilde{U}_l \Sigma_l{}^{\frac{1}{2}}, V_l^\top = \Sigma_l{}^{\frac{1}{2}} \tilde{V}_l^\top$
    end if
end for
for $t = E_{wu} + 1, \ldots, E$ **do**
    Train the hybrid network weights $\{W_1, W_2, \ldots, W_{K-1}, U_K, V_K^\top, \ldots, U_L, V_L^\top\}$ with learning rate schedule $\{\eta_t\}_{t=E_{wu}}^{E}$ ▷ consecutive low rank training
end for

The PUFFERFISH Training Procedure, as detailed in Algorithm 2, outlines an advanced approach to training deep learning models. The process initiates with two principal sets of inputs: the weights of a conventional neural network, denoted as the vanilla network, alongside the weights for a specialized hybrid network architecture. The vanilla network undergoes initial training for a predefined number of epochs, represented as E_{wu}, following a prescribed learning rate schedule. This stage, known as vanilla warm-up training, is essential for establishing a robust baseline for the model's subsequent accuracy.

Subsequent to the warm-up phase, the procedure shifts focus to the initialization of the hybrid network. For layers up to the $K-1$th, weights are replicated directly from the vanilla model, hence maintaining their original full-rank structure. Conversely, for layers starting from the Kth layer onward, weights are transformed through a factorization process using Singular Value Decomposition (SVD), yielding a low-rank format. This technique effectively compacts the model's complexity, which is particularly advantageous for mitigating computation and communication expenditures during training, especially in distributed settings.

The concluding phase engages the hybrid network—now integrated with the newly initialized low-rank layers—in further training for the remaining epochs ($E - E_{wu}$). This stage is intended for the fine-tuning of the model within its revised low-rank configuration, with an objective to retain high model accuracy while ensuring computational economy. The end product is a hybrid model that has been methodically trained to leverage the benefits of both vanilla and low-rank structures. This aligns with the overarching goal of the PUFFERFISH framework, which is to streamline the training process of neural networks without impinging upon their efficacy.

3 Problem Statement

Let $X \in \mathbb{R}^{n \times d}$ be the matrix of an aggregation of n input data points, each with d features, generated by sampling each entry from the normal distribution $X_{ij} \sim \mathcal{N}(0, 1)$, and normalizing each row to have a unit Euclidean norm. Let $Y \in \mathbb{R}^{n \times m}$ be the matrix of an aggregation of n output data points, each with m entries, generated by passing the input data through a low-rank linear channel W^* and adding a little bit of additive white Gaussian noise (AWGN) to the result. The problem can be stated as estimating the low-rank model W^* by accessing the input and output data points like the

least-squares regression problem.

$$\min_{W \in \mathbb{R}^{d \times m}} \frac{1}{2} \|Y - XW\|_F^2 \tag{1}$$

We aim to develop a factorized model represented as $h(X) = XAB^T$, where A is a matrix with dimensions $d \times r$, and B is a matrix with dimensions $m \times r$. In this way, we can inherently force $W = AB^T$ to be low-rank with rank r. This modeling approach extends to various applications, including linear regression, shallow linear networks, robust Principal Component Analysis (PCA), and Low-Rank Adaptation (LoRA) with linear activations. The training process for this model is essentially about solving an optimization problem, where we aim to find the best matrices A and B that minimize the difference between our target Y and the product XAB^T, specifically:

$$\min_{A \in \mathbb{R}^{d \times r}, B \in \mathbb{R}^{m \times r}} \left\|Y - XAB^T\right\|_F^2 \tag{2}$$

In this project, we're particularly interested in scenarios where r is less than the minimum of m and d, which means that the resulting matrix $W = AB^T$ is low-rank. In such cases, each column of A (and correspondingly, each column of B) defines a direction in the low-rank subspace. Our goal is to investigate whether it's possible to sequentially discover these directional vectors using a method similar to deflation, which implicitly maintains orthogonality among them.

Now let's consider a sequence of decreasing values $\sigma_1, \sigma_2, \ldots, \sigma_r$, such as $\sigma_i = \frac{1}{i}$, and let $\Sigma \in \mathbb{R}^{r \times r}$ be the diagonal matrix with $\Sigma_{ii} = \sigma_i$. We generate matrices A^* and B^* as follows: $A^* = \hat{A}^* \Sigma^{\frac{1}{2}}$ and $B^* = \hat{B}^* \Sigma^{\frac{1}{2}}$, where $\hat{A}^* \in \mathbb{R}^{d \times r}$ and $\hat{B}^* \in \mathbb{R}^{d \times r}$ have orthonormal columns. To obtain matrices with orthonormal columns, one method is to generate a random matrix $M \in \mathbb{R}^{d \times m}$, perform Singular Value Decomposition (SVD) on M, and take the top-r left and right singular vectors. After generating A^* and B^*, we then generate the target matrix Y as follows: $Y = XA^*B^{*\top} + z$, where z is the additive white Gaussian noise mentioned above.

3.1 Simple Gradient Descent

First, we implement the simple gradient descent to solve the equation (1). If we do not force the algorithm to have low-rank updates in each iteration, we will estimate the matrix W^* with low error, but the result will not be a low-rank matrix. We can do iterative hard thresholding by keeping the largest r singular values of the update at each iteration. This would be called the projected gradient descent algorithm, and the results of our simulation show that it can calculate the low-rank approximation of W^* with low error. However, this method cannot extract the low-rank subspaces sequentially one by one.

3.2 Factorized Gradient Descent

We utilize Factorized Gradient Descent, a low-rank solver, to tackle the problem. Specifically, the function $f(A, B)$ we aim to minimize is given by:

$$f(A, B) = \frac{1}{2} \|Y - XAB^\top\|_F^2$$

This function represents the squared Frobenius norm of the difference between Y and the product $XAB^\top$, which we attempt to minimize. The update steps in Factorized Gradient Descent for A and B are formulated as follows:

$$A_{t+1} = A_t - \eta_A \nabla_A f(A_t, B_t)$$
$$B_{t+1} = B_t - \eta_B \nabla_B f(A_t, B_t)$$

Here, η_A and η_B represent the learning rates for A and B respectively. The gradients $\nabla_A f(A_t, B_t)$ and $\nabla_B f(A_t, B_t)$ drive the updates, reducing the error in each iteration of the algorithm.

The factorized gradient descent algorithm can be modified to estimate rank-1 matrices by choosing ab^T as the outer product of two vectors instead of matrices A and B. Then the result would be deflated from the data Y and the next orthogonal rank-1 estimates are calculated sequentially. In Figure (3), we can see how the error decreases as we continue to estimate more and more rank-1 matrices and add them up to converge to W^*

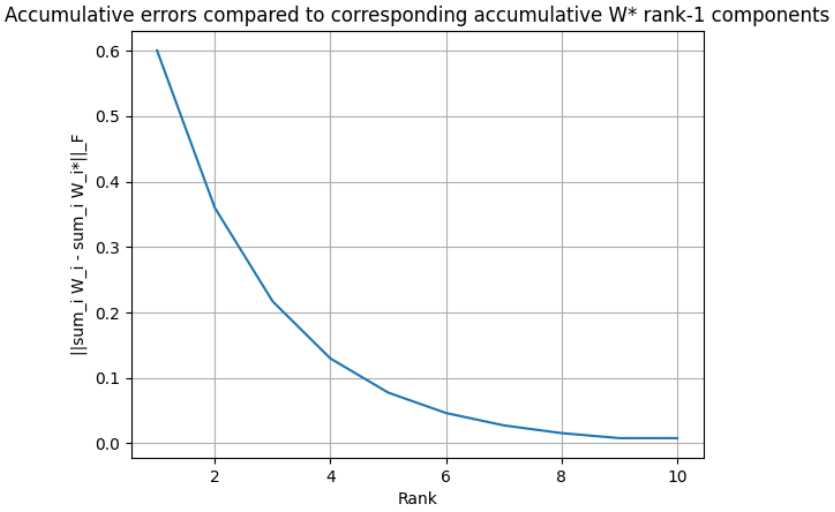

Figure 3: Error diagram, the Frobenius norm of the difference between rank-r approximate with gradient descent based algorithm and the rank-r approximate of W^* is decreasing as we approach the exact rank of W^*

References

[1] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In International Conference on Learning Representations, 2022.

[2] Minyoung Huh, Brian Cheung, Jeremy Bernstein, Phillip Isola, and Pulkit Agrawal. Training neural networks from scratch with parallel low-rank adapters, 2024.

[3] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. Pufferfish: Communication-efficient models at no extra cost, 2021.

DeepSpeed for Large-Scale Model Training and Inference

Sarah Motteler and Anna Tetreault
Rice University
smm34@rice.edu and art7@rice.edu

Abstract

While deep learning models offer solutions to many problems in today's world, they provide issues of their own that must be resolved. DeepSpeed is a deep learning optimization library that collects many features and functions that address these issues, such as parallelism, compression, training, and inference. By gathering all of these functions in one library, DeepSpeed is an easily usable and invaluable tool for model scientists and algorithm designers to improve their models' speed, scalability, and efficiency.

1 Introduction

Training complex deep learning models presents numerous challenges. In addition to devising sophisticated model architectures, model scientists must adeptly implement contemporary training methodologies, encompassing distributed training, mixed precision computation, gradient accumulation strategies, and systematic checkpointing mechanisms. Even with meticulous integration of these techniques, achieving optimal system performance and convergence rates remains difficult at best. The management of large models creates a uniquely daunting challenge: traditional data parallelism methodologies often falter due to memory constraints, but model parallelism techniques also pose formidable technical hurdles. Addressing these multifaceted challenges, DeepSpeed emerges as a pivotal framework designed to expedite the development and training of deep learning models. Through its innovative approach, DeepSpeed not only mitigates the inherent complexities associated with large model sizes but also enhances training efficiency and scalability, thereby facilitating accelerated progress in the field of deep learning research and application.

2 What is DeepSpeed?

DeepSpeed is a transformative deep learning optimization software suite, distinguished by its capacity to facilitate unparalleled speed and scale in both training and inference tasks within the realm of deep learning. Positioned as an accessible solution, DeepSpeed empowers users with a spectrum of functionalities aimed at enhancing efficiency and effectiveness in deep learning endeavors. Its repository on GitHub and its dedicated website serve as conduits for users to explore and engage with its features.

At its core, DeepSpeed embodies a fusion of pioneering system innovations tailored to reshape the landscape of large-scale deep learning training and inference. Among its flagship contributions is ZeRO (which stands for "Zero Redundancy Optimizer"), a groundbreaking approach that optimizes memory consumption and communication overhead, thereby enabling the effective training of large-scale models. Augmenting this innovation is 3D parallelism, which combines multiple subtypes of parallelism and harnesses model and data parallelism along with pipeline parallelism to further accelerate training processes.

Submitted for the Spring 2024 semester of COMP 414.

Moreover, DeepSpeed extends its impact to the realm of inference optimization through a convergence of advanced parallelism techniques, including tensor, pipeline, expert, and ZeRO-parallelism. By amalgamating these methodologies with high-performance custom inference kernels, communication optimizations, and heterogeneous memory technologies, DeepSpeed achieves unparalleled scalability and efficiency in inference tasks, while simultaneously reducing latency, enhancing throughput, and minimizing cost overhead.

Furthermore, DeepSpeed encompasses a suite of compression techniques aimed at augmenting inference efficiency and reducing model sizes. These techniques, characterized by their ease of use and flexibility, cater to the diverse needs of researchers and practitioners. Notable among these innovations are ZeroQuant and XTC, which represent state-of-the-art advancements in model compression, promising faster inference speeds, smaller model footprints, and significantly reduced compression costs.

3 Features of DeepSpeed

DeepSpeed has a number of sophisticated features, each designed to enhance the efficiency and scalability of deep learning tasks. These features, coupled with DeepSpeed's intuitive interface, facilitate seamless integration into existing workflows, empowering users to tackle complex deep learning challenges with ease.

3.1 Parallelism

One way that DeepSpeed achieves such high speed and scalability is through parallelism. Parallelism aims to distribute the training workload of a neural network across all available processing units to achieve higher efficiency. DeepSpeed offers several different methods of achieving parallelism, each offering different benefits and downsides.

3.1.1 Data Parallelism

Data parallelism splits the training data between the GPUs available. Compute efficiency, or efficiency based on how many computations need to be completed, is high since little communication is needed between the units. However, memory efficiency, or efficiency based on the amount of memory used, is low since the model and optimizer need to be replicated on each unit, taking up large amounts of memory. ZeRO uses data parallelism, but it improves memory efficiency by utilizing three optimization stages that partition the optimizer states to reduce the amount of redundant information taking up memory.

3.1.2 Model Parallelism

Model parallelism splits the layers of the model between the GPUs available. This method has high memory efficiency since each worker unit only has to store the layer it is responsible for in memory. However, the compute efficiency is low since units must constantly communicate with each other about activations. DeepSpeed uses NVIDIA's Megatron-LM for model parallelism, which implements tensor parallelism to reduce the amount of communication necessary, therefore increasing compute efficiency.

3.1.3 Pipeline Parallelism

Pipeline parallelism splits the layers of the model into stages, which can be processed in parallel. When the forward pass for a "micro-batch", or small unit of data, is completed, the activation memory is passed on to the next stage. When a stage finishes its back propagation, the gradients are passed backwards through the pipeline. This method has mixed memory and compute efficiency. Increased pipeline stages proportionally decrease memory, but each processing unit must store the activations for all micro-batches currently being processed. Pipelining has very low communication overhead, but the stages must be exactly load-balanced to achieve good compute efficiency.

3.1.4 3D parallelism

3D parallelism combines multiple subtypes of parallelism, harnessing model and data parallelism along with pipeline parallelism to further accelerate training processes. The key attribute of 3D parallelism is its adaptability, which allows it to handle massive models with over a trillion parameters with high memory and compute efficiency. Memory efficiency is improved by the use of pipeline stages, as in pipeline parallelism, which are then divided up again using model parallelism; this allows the model, optimizer, and activations to use less memory. The compute efficiency is improved by ZeRO data parallelism, which allows for the model to scale to any number of GPUs without excess communication overhead while also improving memory efficiency.

3.2 Compression

DeepSpeed allows for easy compression of deep learning models via the DeepSpeed Compression library, which supports several compression methods for reducing the memory taken up by the model. Benefits of using the library include faster compression time, higher quality and efficiency in less space, and reduced compression cost.

3.2.1 Layer Reduction

The layer reduction method compresses a deep learning model by removing some of the hidden layers without changing the network's width, reducing the inference latency of the hidden layers. This method is best used when the model is deep and/or the knowledge is being transferred from a larger model to a smaller model.

3.2.2 Weight Quantization

Weight quantization reduces the precision of weights by mapping the original, full-precision number to an equivalent low-bit representation, improving execution performance and efficiency but lowering accuracy. This method is best used when high accuracy is not as important as having lower complexity, such as in devices with lower computation resources like smartphones.

3.2.3 Activation Quantization

Activation quantization reduces the precision of the activation (input to each layer of the model) by mapping the original number to an equivalent low-precision representation, causing a similar effect to weight quantization (improved performance/efficiency, worsened precision/accuracy). Just like weight quantization, this method is most useful when the accuracy is not as important as being able to run the model using limited computational resources.

3.2.4 Pruning

Pruning reduces the number of parameters and operations used for making predictions by "pruning", or removing, connections in the network. Pruning can be split into four submethods:

1. **Sparse pruning** reduces the number of parameters and operations by setting some elements within each weight matrix to zero, causing these zero-set elements to have no effect on the prediction, improving hardware speedup but reducing accuracy. This submethod of pruning is best used when the ratio of weights kept after pruning vs total weights before pruning is very low.
2. **Row pruning** reduces the number of parameters and operations by setting all of the elements within certain rows of the weight matrix to zero, causing these zero-set rows to have no effect on the prediction, greatly improving hardware speedup but greatly reducing accuracy. This submethod of pruning is designed for models with two back-to-back linear layers, although it also works for models with other kinds of linear layers.
3. **Head pruning** reduces the number of parameters and operations by removing some of the many heads in a model, improving hardware speedup. This submethod of pruning is designed for models that have multiple heads.

4. **Channel pruning** reduces the number of parameters and operations by removing some of the many channels in a model, improving hardware speedup. This submethod of pruning is designed for models with two back-to-back two-dimensional convolutional layers, although it also works for models with other kinds of two-dimensional convolutional layers.

3.2.5 ZeroQuant

ZeroQuant compresses a deep learning model by performing both weight and activation quantization at low or no cost and with minimal quantization error. This method can only be used after training and is best used whenever a transformer-based model needs to be converted to using INT8 weights/activations. It is especially helpful when the model is very time- or computational resource-hungry.

3.2.6 XTC

eXTreme Compression (or XTC) greatly compresses a deep learning model by both reducing the layers of a model and greatly reducing the precision of weights and/or activations via binary/ternary quantization without losing much accuracy. This method is best used when the model needs significant compression as well as strong performance.

3.3 Support for Long Sequence Length

DeepSpeed revolutionizes long sequence processing with its sparse attention kernels, a pivotal technology accommodating extended sequences of model inputs across various data modalities including text, image, and sound. In contrast to conventional dense transformers, these kernels facilitate processing sequences orders of magnitude longer, achieving up to 6 times faster execution while maintaining comparable accuracy. DeepSpeed's sparse attention kernels outperform existing state-of-the-art implementations, delivering 1.5–3 times faster execution. Additionally, they support the efficient execution of diverse sparse formats, empowering users to innovate and customize their sparse structures effectively. This suite of capabilities underscores DeepSpeed's commitment to enabling efficient processing of long sequences and fostering innovation in deep learning research and application.

3.4 Convergence Acceleration

DeepSpeed accelerates convergence through its support for advanced hyperparameter tuning techniques and large batch size optimizers like LAMB. By leveraging these tools, DeepSpeed enhances the effectiveness of model training, enabling faster convergence to the desired accuracy with fewer samples. This combination of advanced tuning methodologies and optimized batch size selection underscores DeepSpeed's commitment to expediting the training process and achieving superior model performance in a more efficient manner.

3.5 Training

Thanks to its many capabilities discussed above, DeepSpeed is capable of training very large, high-density deep learning models, including Turing-NLG, Big Science, and Megatron-Turing NLG 530B. The many options and combinations of features offered by DeepSpeed allow for all types of models to be trained via DeepSpeed technologies.

The two main categories are ZeRO training technologies and 3D parallelism training technologies. ZeRO requires less code refactoring, while 3D parallelism has better throughput efficiency. The two technologies have similar performance when the batch size per GPU is larger, but as the batch size decreases, the gap in performance increases, with the performance of 3D parallelism exceeding that of ZeRO. Therefore, ZeRO training is better for most scenarios, while 3D parallelism training is better for very large models with hundreds of billions or trillions of parameters, similar to GPT-2/GPT-3.

3.6 Inference

If a deep learning model was trained in DeepSpeed, then the DeepSpeed Inference library can be used to apply the trained model to the desired scenario. The DeepSpeed Inference library seeks to

address limitations in existing inference tools with three main features: multi-GPU inference utilizing adaptive parallelism, kernels that are optimized for inference and tuned for smaller batch sizes, and support for quantization via quantize-aware training and inference kernels adapted to quantized models. Together, these three features greatly increase DeepSpeed Inference's throughput per GPU in comparison to PyTorch. Due to quantization, the number of GPUs needed to run DeepSpeed Inference for massive models is at least half that of full-precision PyTorch.

3.6.1 Multi-GPU inference utilizing adaptive parallelism

As in training, parallelism can be used in inference to fit large models and evenly spread memory consumption among the processing devices, but the choices made for training do not necessarily work for inference. Inference tends to require less memory and needs a stronger focus on latency optimization or meeting the requirements for latency rather than throughput, as in training.

To this end, the DeepSpeed Inference library first uses model parallelism to reach the latency target, then adds in pipeline parallelism to optimize the throughput. This adaptive approach meets the goals of model inference while reducing the cost of deployment.

3.6.2 Small-batch custom inference kernels

DeepSpeed Inference uses inference kernels tailored for model parallelism for multiple GPUs via operator fusion, fusing general matrix multiply operations as well as element-wise operations using efficient vector-matrix and skinny matrix-matrix multiplication. Fused operations must keep the access-pattern of inputs/outputs intact throughout the entire sequence of fused operations (preventing different thread-blocks from encountering data being transferred between the streaming-multiprocessors). Additionally, the fusion must occur at every all-reduce boundary so that the execution can continue since the execution is paused until partial results are reduced.

To run inference, the DeepSpeed inference kernels need the location of the model checkpoints and the degree of model parallelism and pipeline parallelism. Kernels can be customized for standard model architectures (ex. HuggingFace or Megatron-GPT) via a provided policy map, which maps the original parameters to those in the inference kernels. Other models can provide their own policy map to the kernels to run inference mode.

3.6.3 Quantization support

The DeepSpeed Quantization Toolkit is designed to reduce the inference cost for large models via quantization, creating support for flexible quantize-aware training and high-performance quantized inference kernels. The toolkit does not require client-side code changes, allowing for ease of use.

In the training, mixed-precision training occurs in tandem with the application of quantization in a process called Mixture of Quantization (MoQ), allowing for control over the model's precision via simulation of the impact of quantization on the parameters at each step of training. This also supports flexible quantization schedules and policies, since the adjustment of the quantity of quantization bits during training results in higher accuracy using the same compression ratio in the final quantized model. MoQ can also adapt to various tasks by using models' second-order information to determine how much precision is required, tweaking the quantization target and schedule to compensate.

In order to optimize the benefits of quantization, the DeepSpeed Quantization Toolkit creates inference kernels custom-made for quantized models: these kernels reduce latency by optimizing the movement of data, but they don't require any specialized hardware.

4 Benefits of DeepSpeed

4.1 Speed

DeepSpeed achieves remarkable training times for large-scale models, exemplified by its ability to train BERT-large to parity in a mere 44 minutes utilizing 1024 V100 GPUs or in 2.4 hours with 256 GPUs. Such expedited training is further underscored by DeepSpeed's capacity to outpace state-of-the-art solutions, exemplified by its 3.75x faster training of GPT2 (1.5 billion parameters) compared to NVIDIA Megatron on Azure GPUs. Central to DeepSpeed's efficacy is its enhanced

memory efficiency, facilitating higher throughput and faster convergence rates. This is exemplified by ZeRO-2's capability to train 100-billion-parameter models on a 400 NVIDIA V100 GPU cluster with over 38 teraflops per GPU and aggregated performance exceeding 15 petaflops. Compared to using Megatron-LM alone, ZeRO-2 achieves a remarkable 10x speedup in training speed for models of equivalent size, underscoring its prowess in accelerating deep learning workflows.

4.2 Scalability

One of DeepSpeed's standout offerings is its ability to efficiently run large-scale models, delivering up to 10x faster training speeds across a spectrum of model sizes ranging from 1.5 billion to hundreds of billions of parameters. This efficiency is particularly pronounced in configurations leveraging ZeRO-powered data parallelism, which can be seamlessly combined with various types of model parallelism. By optimizing memory usage and batch sizes, DeepSpeed achieves significant performance gains compared to relying solely on model parallelism.

In benchmarking against the GPT-3 model architecture with over 175 billion parameters, DeepSpeed's 3D parallelism configurations demonstrate remarkable efficiency. While 2D configurations struggle with low throughput due to suboptimal parallelism strategies, the 3D configurations, arranged by increasing degrees of pipeline parallelism, achieve superior performance by striking a balance between memory, compute, and communication efficiency. The best 3D approaches attain an impressive 49 teraflops per GPU, representing over 40% of the theoretical hardware peak.

4.3 Efficiency

DeepSpeed offers impressive efficiency in memory, data, and computation.

4.3.1 Memory Efficiency

DeepSpeed's prowess in memory efficiency revolutionizes deep learning training by offering memory-efficient data parallelism. Notably, DeepSpeed excels in training models with up to 13 billion parameters on a single GPU, a feat unattainable by existing frameworks like PyTorch's Distributed Data Parallel, which struggles with models exceeding 1.4 billion parameters due to memory limitations.

Central to DeepSpeed's memory optimization strategy is its innovative algorithm, the Zero Redundancy Optimizer (ZeRO), which partitions model states and gradients to significantly reduce memory consumption. Unlike conventional data parallelism approaches that replicate memory states across processes, ZeRO mitigates memory overhead by strategically partitioning data. ZeRO minimizes activation memory and fragmented memory, further enhancing memory efficiency. The current version, ZeRO-2, achieves up to an 8x reduction in memory usage compared to prevailing methodologies, as detailed in our research paper and related blog posts.

The profound impact of DeepSpeed's memory optimization capabilities is evidenced by the accomplishments of early adopters, who have successfully trained the Turing-NLG language model boasting over 17 billion parameters, thus setting a new standard in the language model domain.

Additionally, DeepSpeed extends its memory optimization capabilities through ZeRO-Offload, which leverages both CPU and GPU memory for training large models. This innovative approach enables users to train models with up to 13 billion parameters on a single GPU, a tenfold increase compared to existing methodologies while maintaining competitive throughput. By democratizing multi-billion-parameter model training, ZeRO-Offload empowers deep learning practitioners to explore larger and more sophisticated models, thereby advancing the frontiers of deep learning research and application.

4.3.2 Data Efficiency

The Data Efficiency Library spearheads advancements in data efficiency by offering efficient data sampling through curriculum learning and streamlined data routing via random layerwise token dropping. This integrated solution delivers remarkable benefits, including up to 2 times data and time savings during pretraining of models like GPT-3 and BERT, as well as during finetuning tasks such as GPT and ViT. Alternatively, it enables users to enhance model quality within the same data and time constraints.

4.3.3 Compute Efficiency

DeepSpeed showcases exceptional compute efficiency through its utilization of pipeline parallelism, which effectively reduces communication volume during distributed training. This reduction in communication overhead enables users to train multi-billion-parameter models 2–7 times faster on clusters with limited network bandwidth, enhancing training efficiency and scalability. Furthermore, DeepSpeed introduces optimized communication techniques such as 1-bit Adam, 0/1 Adam, and 1-bit LAMB, which significantly reduce communication volume by up to 26 times while maintaining comparable convergence efficiency to traditional Adam optimization. This reduction in communication overhead not only facilitates seamless scaling across different types of GPU clusters and networks but also enhances the overall efficiency of distributed training workflows.

4.4 Usability

DeepSpeed offers exceptional usability, requiring minimal code changes to enable a PyTorch model to utilize its capabilities, including ZeRO. Unlike most current model parallelism libraries, DeepSpeed doesn't necessitate code redesign or model refactoring, ensuring a seamless integration process. It also imposes no restrictions on model dimensions, batch size, or other training parameters. For models of up to 13 billion parameters, ZeRO-powered data parallelism can be conveniently employed without requiring model parallelism, while standard data parallelism typically encounters memory limitations for models exceeding 1.4 billion parameters. Additionally, DeepSpeed facilitates flexible combinations of ZeRO-powered data parallelism with custom model parallelisms, such as tensor slicing of NVIDIA's Megatron-LM, which further enhances its usability and adaptability to diverse training scenarios.

4.5 Accessibility

DeepSpeed makes deep learning more accessible by offering algorithm designers a suite of benefits that streamline the development process, allowing them to focus on conceptualizing and refining cutting-edge algorithms rather than getting bogged down in implementation intricacies.

Firstly, by tackling complexity and scale. With DeepSpeed, algorithm designers can delve into more intricate tasks and handle vast datasets with ease. The framework's optimizations enable efficient processing of large-scale data, empowering designers to tackle complex problems without being hindered by scalability concerns.

Additionally, DeepSpeed accelerates experimentation and iteration cycles, significantly reducing the time required to test and refine algorithms. This rapid feedback loop enables designers to explore various approaches swiftly, facilitating quicker progress and innovation.

Thirdly, by abstracting away implementation details, DeepSpeed allows algorithm designers to concentrate on refining the core concepts of their algorithms. Freed from the burden of low-level coding tasks, designers can devote their energy to pushing the boundaries of algorithmic innovation.

Allowances for memory-intensive algorithm exploration also benefit algorithm designers. DeepSpeed's efficient memory management capabilities remove the constraints typically associated with memory-intensive algorithms. Algorithm designers can explore and develop sophisticated models without worrying about memory limitations, opening up new avenues for experimentation and advancement.

DeepSpeed also provides a flexible environment for experimenting with different training designs and performance optimizations. Designers can easily tweak parameters, explore alternative architectures, and fine-tune training strategies to achieve optimal results for their algorithms. In essence, DeepSpeed empowers algorithm designers to unlock their creativity and push the boundaries of what's possible in algorithmic research and development, ushering in a new era of innovation and discovery.

5 Conclusion

DeepSpeed has emerged as a pivotal force in the advancement of deep learning methodologies. It offers a comprehensive arsenal of tools and techniques to expedite model development, training, and inference processes. Simultaneously, DeepSpeed pushes the boundaries of scalability, efficiency,

and cost-effectiveness in the domain of deep learning research and application. With these tools and benefits, DeepSpeed helps to democratize deep learning and make it accessible to more and more people.

References

[1] "DeepSpeed Inference: Multi-GPU inference with customized inference kernels and quantization support." DeepSpeed, https://www.deepspeed.ai/2021/03/15/inference-kernel-optimization.html. Accessed 15 Apr. 2024.

[2] "Latest News." DeepSpeed, www.deepspeed.ai. Accessed 15 Apr. 2024.

[3] Patel, Hitesh, et al. "Zero Redundancy Optimizers: A Method for Training Machine Learning Models with Billion Parameters." Medium, Medium, 10 Aug. 2021, oracle-oci-ocas.medium.com/zero-redundancy-optimizers-a-method-for-training-machine-learning-models-with-billion-parameter-472e8f4e7a5b. Accessed 15 Apr. 2024.

[4] "Pipeline Parallelism." DeepSpeed, www.deepspeed.ai/tutorials/pipeline. Accessed 15 Apr. 2024.

[5] Rajbhandari, Samyam, et al. "ZeRO: Memory optimizations Toward Training Trillion Parameter Models," arXiv.org (2020): n. pag. Web.

[6] Ruwase, Olatunji, et al. "DeepSpeed: Extreme-Scale Model Training for Everyone." Microsoft Research, Microsoft, 10 Sept. 2020, www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[7] Shoeybi, Mohammad et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." arXiv.org (2020): n. pag. Web.

[8] Qi, Xiandong. "Intro Distributed Deep Learning." Xiandong Qi, 13 May 2017, xiandong79.github.io/Intro-Distributed-Deep-Learning. Accessed 15 Apr. 2024.

[9] "Training Overview and Features." DeepSpeed, www.deepspeed.ai/training/. Accessed 15 Apr. 2024.

[10] "Training your large model with DeepSpeed." DeepSpeed, www.deepspeed.ai/tutorials/large-models-w-deepspeed. Accessed 15 Apr. 2024.