DSA4212:
Shampoo Optimizer

November 15, 2024

Group Composition:

1. Yap Vit Chun, A0225837X, e0596609@u.nus.edu

Main References: The main reference used throughout this assignment is the paper by Gupta et al. (2018) titled 'Shampoo: Preconditioned Stochastic Tensor Optimization'. Some reference was also taken from a YouTube video of the authors' sharing titled 'JAX Meetup: Scalable second order optimization for deep learning [ft. Rohan Anil]'

1 Introduction

Optimization algorithms are foundational in machine learning and deep learning, as they adjust model parameters to minimize a loss function. In our course DSA4212, the first algorithm (and the simplest) introduced to us was **Gradient Descent**, which simply computes the gradient of the loss function with respect to each parameter and updates parameters in the direction of the steepest descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

where θ represents the parameters, η is the learning rate, and $\nabla_\theta L(\theta_t)$ is the gradient of the loss function with respect to θ at iteration t.

However, as mentioned in class, **Gradient Descent** can be computationally expensive, especially in large datasets, as it requires calculating the full gradient across all samples.

Hence, **Stochastic Gradient Descent (SGD)** was introduced. Unlike Gradient Descent, SGD computes the gradient on a randomly selected subset of data (a mini-batch or even a single sample), making it computationally efficient:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_i)$$

where $L(\theta_t; x_i)$ is the loss function evaluated at a single sample x_i or a mini-batch of samples, which introduces noise but allows for faster updates. Moreover, we also learned that the stochastic gradient estimate is an unbiased estimator, meaning, on average, the stochastic gradient points exactly in the direction of the true gradient.

However, with a constant step size (learning rate), the SGD algorithm may not converge towards a minimum of the objective function. Hence, we have optimizers that adapt the learning rate based on some statistics of the gradient estimates, such as AdaGrad, RMSprop, and Adam optimizers.

AdaGrad adapts the learning rate by dividing it by the square root of the accumulated outer product of the gradients, g_t:

$$G_t = \sum_{i=1}^{t} \nabla_\theta L(\theta_i) \nabla_\theta L(\theta_i)^T$$

and the update rule:

$$\theta_{t+1} = \theta_t - \eta G_t^{-\frac{1}{2}} \nabla_\theta L(\theta_t)$$

However, computing and storing the full matrix G_t becomes infeasible as the size of the dataset or number of parameters increases, due to significant memory and computation costs. In practice, the commonly used version of AdaGrad focuses only on the diagonal elements of G_t which is just the squared gradients and can be computed efficiently in linear time, making it far more scalable.

The update rule for this diagonal AdaGrad is:

$$G_t = G_{t-1} + \nabla_\theta L(\theta_i)^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\epsilon + \sqrt{G_t}} \nabla_\theta L(\theta_t)$$

where ϵ is a small constant to prevent division by zero and $\nabla_\theta L(\theta_i)^2$ means that each coordinate of the gradient is squared. Note that G_t represents an approximation of the gradient covariance matrix, with its diagonal elements capturing the variance of the gradients over time. Informally and intuitively, this means that when the variance of the gradient is high, AdaGrad automatically decreases the learning rate for that parameter, thus reducing the impact of the update and promoting stability in high-variance directions.

Building on AdaGrad, we have the **RMSprop** which introduces exponentially decaying average of past squared gradients - the diagonal elements of :

$$G_t = \beta G_{t-1} + (1 - \beta) \nabla_\theta L(\theta_t)^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\epsilon + \sqrt{G_t}} \nabla_\theta L(\theta_t)$$

where β is the decay rate, helping to smooth the adaptive learning rate over time.

similarly, the **Adam optimizer** introduced in our class further improves upon RMSprop by introducing momentum, a mechanism that takes the moving average of past gradients. In particular, Adam combines momentum with adaptive learning rates:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta_t) \\
G_t &= \beta_2 G_{t-1} + (1-\beta_2)\nabla_\theta L(\theta_t)^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{G}_t = \frac{G_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\epsilon + \sqrt{\hat{G}_t}}\hat{m}_t
\end{aligned}
$$

where m_t and G_t are the biased first and second moments of the gradient, and $\hat{m}_t$ and $\hat{G}_t$ are bias-corrected estimates, resulting in smoother convergence. As of now, Adam is one of the most used optimizers in the industry for deep learning due to its robustness.

The shampoo optimizer that we will be exploring in this assignment is another variation that builds upon the AdaGrad optimizer. Before we move on to the next section that elaborates Shampoo, it is also worth pointing out that the AdaGrad, RMSprop and Adam optimizers can also be viewed as **second-order-like** method. Respectively, their $G_t^{-\frac{1}{2}}$ acts as a "preconditioning matrix" that aims to approximate the Hessian of the objective function.

2 Shampoo Optimizer

The **Shampoo optimizer** was first introduced in 2018, motivated by the full AdaGrad optimizer. Unlike the diagonal AdaGrad optimizer, which only considers the diagonal elements of

$$
G_t = \sum_{i=1}^{t} \nabla_\theta L(\theta_i),
$$

Shampoo maintains two preconditioning matrices, denoted as L_t and R_t. Here, L_t (left) preconditions the rows of $\nabla_\theta L(\theta_t)$, while R_t (right) preconditions the columns of $\nabla_\theta L(\theta_t)$. Suppose the dimension of $\nabla_\theta L(\theta_i)$ is (m, n); then L_t has dimensions (m, m) and R_t has dimensions (n, n). The matrices L_t and R_t are defined as follows:

$$
\begin{aligned}
L_t &= L_{t-1} + \nabla_\theta L(\theta_t)\nabla_\theta L(\theta_t)^\top, \\
R_t &= R_{t-1} + \nabla_\theta L(\theta_t)^\top \nabla_\theta L(\theta_t),
\end{aligned}
$$

and the update rule is given by:

$$
\theta_{t+1} = \theta_t - \eta L_t^{-\frac{1}{4}} \nabla_\theta L(\theta_t) R_t^{-\frac{1}{4}}.
$$

The $\frac{1}{4}$ exponent was chosen based on the analysis presented in the paper. Intuitively, this choice induces an overall step-size decay rate of $O\left(\frac{1}{\sqrt{t}}\right)$, which is desirable for convergence.

Note that even though the Shampoo optimizer may look very similar to the full AdaGrad, the key difference lies in the dimension of the preconditioning matrix. In simple neural network such as multiple layer perceptron (MLP), the $\nabla_\theta(\theta_t)$ typically has dimension (m, n), depending on the network structure. The full Adagrad would first flatten the gradient into vectors of dimension mn, and subsequently the G will be of dimension (mn, mn), whereas for Shampoo, the two preconditioners have dimensions (m, m) and (n, n).

With that, we have some notable observations about the Shampoo optimizer:

1. The space requirement for the preconditioning matrices L_t and R_t is $O(m^2+n^2)$, as opposed to the $O(m^2n^2)$ space needed for full-matrix AdaGrad.
2. The computational requirement for updating the preconditioning matrices L_t and R_t is $O(m^3+n^3)$ (due to matrix inversion through spectral decomposition), which is significantly less than the $O(m^3n^3)$ for full-matrix AdaGrad.
3. The update rule is relatively simple to implement.

These characteristics make Shampoo significantly more memory- and computation-efficient than full AdaGrad, especially as the parameter space grows, a common scenario in deep learning. Shampoo achieves a balance between the simplicity of diagonal methods (e.g., Adam, RMSProp) and the expressiveness of full-matrix AdaGrad by effectively leveraging preconditioners.

In particular, unlike methods such as Adam, RMSProp, or SGD, which only scale the gradient, Shampoo is also designed to "rotate" the gradient using second-order information. By transforming the gradient into a more optimal direction, Shampoo can converge faster and more effectively, particularly in ill-conditioned optimization problems. This makes it especially advantageous for deep learning tasks with large and complex parameter spaces, where improved convergence rates are critical.

3 Implementing Shampoo in Python

3.1 Problem statement

In this assignment, we will attempt to implement the Shampoo optimizer using Python and JAX. In particular, we have chosen to apply the optimizer to a classification problem on the MNIST dataset, a standard benchmark dataset widely used in machine learning.

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits (0-9) that are grayscale and has a resolution of 28x28 pixels, resulting in 784 features when flattened. We will build a neural network that takes in the 784 flattened features as input and classify the images.

3.2 Problem choice

We have chosen this problem as it strikes a balance between simplicity and complexity, making it an appropriate benchmark for testing the Shampoo optimizer. For simpler problems like the Rosenbrock function, the advantages of Shampoo are less pronounced because such problems involve smooth optimization landscapes with relatively few parameters. In contrast, the chosen problem, with a more complex neural network structure featuring more nodes in each layer, aligns better with Shampoo's design philosophy. Shampoo is specifically optimized for tasks where the parameter space is large and the optimization landscape is ill-conditioned.

While it would be ideal to compare Shampoo's performance on a more complex problem, such as training a convolutional neural network (CNN) on the CIFAR-10 data set, we have decided not to do so due to the increased complexity, especially in handling tensor operations and the challenges of developing an optimizer for such cases. By focusing on the MNIST classification task, we hope to uncover some meaningful insights into Shampoo's behavior and its potential benefits, even in a moderately complex setting.

3.3 Neural network structure

The following table summarizes the structure of our chosen MLP:

Layer	**Units**	**Weight Initialization**	**Activation**
Input Layer	784 (Flattened)	N/A	N/A
Hidden Layer 1	256	Glorot Uniform	ReLU
Hidden Layer 2	128	Glorot Uniform	ReLU
Hidden Layer 3	64	Glorot Uniform	ReLU
Output Layer	10	Glorot Uniform	None (Logits)

In our implementation, we will initialize the parameters with a fixed key, fix the learning rate of 0.001 and batch size of 256, and train the MLP for 10 epochs by minimizing the entropy loss function using:

1. Shampoo optimizer (our own implementation)
2. Adam
3. RMSprop
4. SGD

whereas for Adam, RMSprop and SGD, we will be using the implementation by Optax. We will then compare the performance of the optimizers in terms of convergence speed and accuracy.

3.4 Implementing Shampoo optimizer

We will be following the Algorithm 1 provided by Gupta et al. (2018), which seems simple to implement. However, one major challenge we encountered was using JAX, as the parameters of the neural network in JAX are stored in a dictionary, making it necessary to carefully manage and handle the dimensions of each parameter during the optimization process.

Initialize $W_1 = \mathbf{0}_{m\times n}$; $L_0 = \epsilon I_m$; $R_0 = \epsilon I_n$
for $t = 1, \ldots, T$ **do**
  Receive loss function $f_t : \mathbb{R}^{m\times n} \mapsto \mathbb{R}$
  Compute gradient $G_t = \nabla f_t(W_t)$ $\{G_t \in \mathbb{R}^{m\times n}\}$
  Update preconditioners:

$$L_t = L_{t-1} + G_t G_t^\intercal$$
$$R_t = R_{t-1} + G_t^\intercal G_t$$

  Update parameters:

$$W_{t+1} = W_t - \eta L_t^{-1/4} G_t R_t^{-1/4}$$

Algorithm 1: Shampoo, matrix case.

A more significant and interesting issue we ran into was performing the operations $L_t^{-\frac{1}{4}}$ and $R_t^{-\frac{1}{4}}$. Initially, we relied on the built-in function fractional_matrix_power from the SciPy library to compute these fractional matrix powers and invert it using solve. However, when we ran the optimizer, we encountered errors almost immediately.

Upon further reading, we found that the primary issue was that the preconditioners L_t and R_t, derived from second-moment accumulations, were sometimes not strictly positive semidefinite due to numerical instability or ill-conditioning. As a result, their eigenvalues could include negative or very small values. This caused 'fractional_matrix_power' to produce complex outputs, which are incompatible with our optimizer's requirements. Ensuring the preconditioners remained positive semidefinite was critical to resolving this issue.

Reading more recent papers on Shampoo optimizers (Anil et al., 2020; Morwani et al., 2024) prompted us to try the Schur-Newton method to compute the inverse 4th root of the matrices. Briefly, suppose we want to find the p^{th} of A, then we can attempt to solve $X^{-p} - A = 0$ and the Newton method will take the form

$$X_{k+1} = \frac{1}{p}\{(p+1)X_k + X_k^{p+1}A\}, \quad X_0 = \frac{1}{c}I$$

where $X_k \to A^{-frac1p}$ as $k \to \infty$. The above method may not be numerically stable if we have large eigenvalues magnitude or if A is ill-conditioned, which is the case for our Shampoo implementation. The paper by Guo and Higham (2006) has shown that a modification to the

Schur-Newton method can achieve numerical stability, with the updates given as

$$X_{k+1} = X_k \left(\frac{(p+1)I - M_k}{p} \right), \quad X_0 = \frac{1}{c} I$$

$$M_{k+1} = X_{k+1}^p A = \left(\frac{(p+1)I - M_k}{p} \right)^p X_k^p A$$

$$= \left(\frac{(p+1)I - M_k}{p} \right)^p M_k, \quad M_0 = \frac{1}{c^p} A$$

We have adopted the above algorithm in our implementation. In particular, we have also set $p = 2$ to obtain the inverse square root of the matrix, X, then we multiply X by X to obtain our desired inverse fourth root. With this algorithm at hand, our optimizer finally converged without errors.

4 Results and Discussion

Figures 1 and 2 summarize the convergence performance and accuracy of the four optimizers discussed in Section 3.3. From Figure 1, it is evident that the Shampoo optimizer outperforms the others in minimizing cross-entropy loss for both the training and test datasets, with SGD showing the poorest performance. This aligns with our expectations, as Shampoo employs more sophisticated preconditioners compared to the other methods.

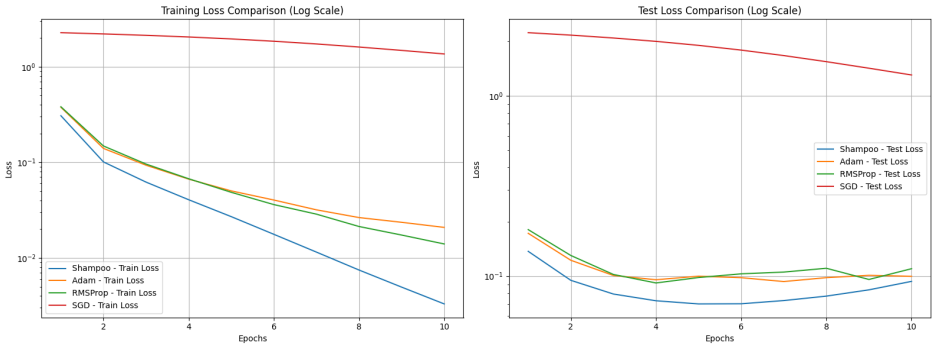

Figure 1: Training loss and validation loss for each epoch

Interestingly, Adam and RMSprop demonstrate similar performance, supporting what we learned in this course that the choice of optimizer often usually does not lead to significant differences in outcomes, except with something new like Shampoo.

One interesting (though less relevant) observation is that overfitting begins to appear around epochs 5 or 6 where the validation loss starts increasing while training loss continues to decrease (except for SGD), suggesting that employing early stopping could be a viable strategy to prevent

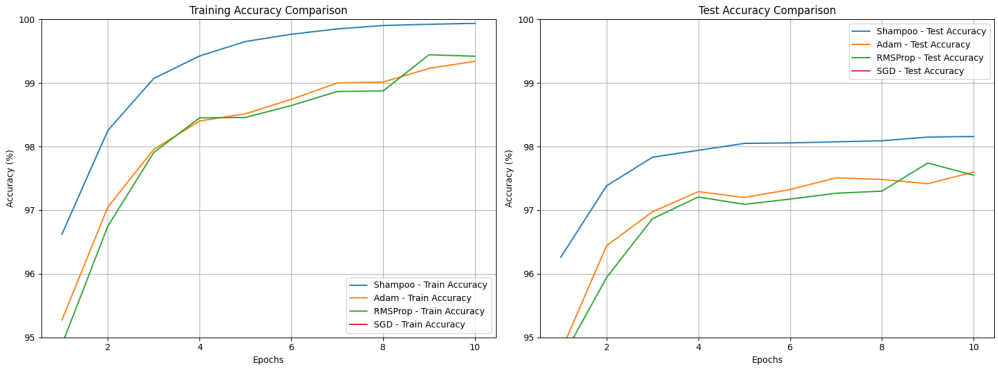

Figure 2: Training accuracy and validation accuracy for each epoch

overfitting.

Figure 2 presents similar findings. Namely, Shampoo demonstrates significantly higher training and test accuracy compared to Adam and RMSprop, which perform similarly to each other. Notably, the accuracy trajectory for SGD is not shown in the plot as its values are too low. To better highlight the differences between the other three optimizers, the y-axis limits have been adjusted.

Although Shampoo demonstrated superior performance in terms of convergence speed in our assignment, a notable drawback of this optimizer is its high memory and computational cost mentioned in Section 2. While directly measuring memory usage across optimizers is challenging, we have recorded the computation time required per epoch for each optimizer to provide a comparative assessment.

Optimizer	**Computation Time per Epoch (s)**
Shampoo	222
Adam	4-5
RMSprop	4-5
SGD	2

We observe that Shampoo takes significantly longer to run per epoch, which is expected due to the more sophisticated steps involved in the algorithm, particularly the computation of the inverse fourth root. A key takeaway and conclusion is that while Shampoo can achieve convergence in a **smaller number of steps**, each **step requires significantly more computation and time**. However, it is worth pointing out that in some scenarios, the reduction in the number of steps may offset the longer per-step computation time, potentially resulting in a shorter overall training duration. Some studies have demonstrated such cases, particularly with much more optimized implementations of Shampoo, and when applied to much more complicated problems such as Large Language Model and CNN with complex architecture such as ResNet. (Anil et al., 2020; Morwani et al., 2024; Gupta et al., 2018)

References

[AGK+20] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv (Cornell University)*, 1 2020.

[GH06] Chun-Hua Guo and Nicholas J. Higham. A Schur–Newton Method for the Matrix th Root and its Inverse. *SIAM Journal on Matrix Analysis and Applications*, 28(3):788–804, 1 2006.

[GKS18] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. *International Conference on Machine Learning*, pages 1842–1850, 7 2018.

[MSV+24] Depen Morwani, Itai Shapira, Nikhil Vyas, Eran Malach, Sham Kakade, and Lucas Janson. A new perspective on shampoo's preconditioner. *arXiv (Cornell University)*, 6 2024.