Scalable Second Order Optimization for Deep Learning

Preprint: *https://arxiv.org/abs/2002.09018*

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, Yoram Singer

rohananil @ google dot com

March 12, 2021 at Deep Learning Classics and Trends

Distributed Implementation

@_arohan_

Make neural network training efficient at scale

Make neural network training efficient at scale

This work: making Shampoo a second order method work in practice

Make neural network training efficient at scale

This work: making Shampoo a second order method work in practice

Several other approaches: sparsification, variance reduction, momentum based methods, meta-optimization, exponentiated gradients/multiplicative updates, target prop etc

Why?

Improvements from first order methods have reached a plateau

Both in terms of ***steps to convergence***, and ***implementation performance***

Why is it called Shampoo?

Because it does preconditioning!

Credit: Roy Frostig who suggested the name.

Preconditioning

$$\min_{w \in \mathbb{R}^d} F(w)$$

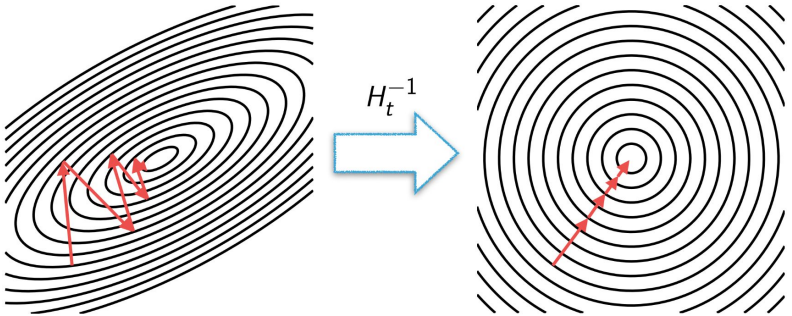
For t=1,2,..T:

$$w_{t+1} = w_t - \eta \, H_t^{-1} g_t$$

H_t:

- Newton
- Natural Gradient
- Full matrix AdaGrad, ...

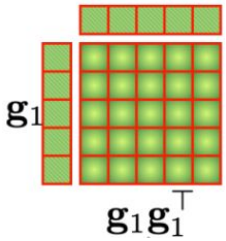
often leads to faster convergence / better "condition number"



Geometrically they **scale and rotate gradients**

Whereas **first order methods** only **scale gradients.**

Full Matrix Adagrad Preconditioner

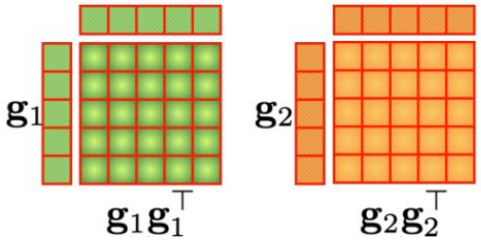


- Time step 1
- Compute outer product of gradient vectors

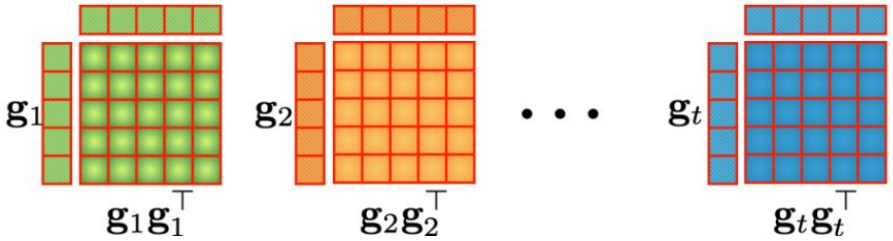
"Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", Duchi, Hazan, Singer'10

"Adaptive Bound Optimization for Online Convex Optimization", H. Brendan McMahan, Matthew Streeter'10

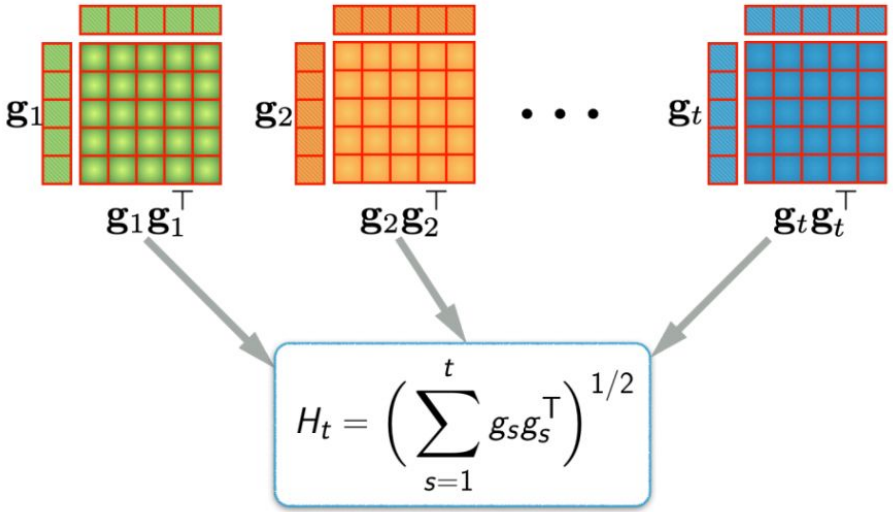
Full Matrix Adagrad Preconditioner



Full Matrix Adagrad Preconditioner



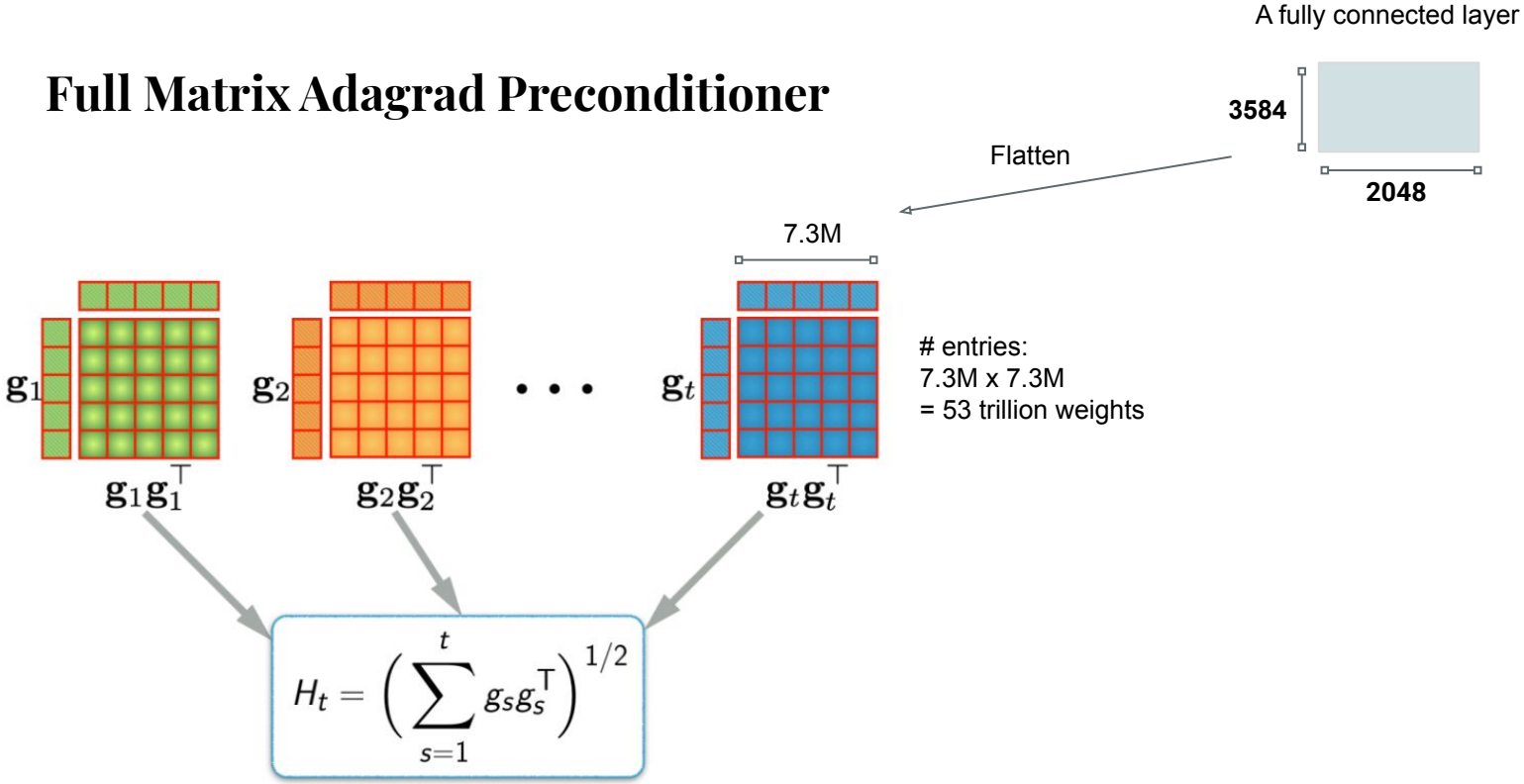
Full Matrix Adagrad Preconditioner



"Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", Duchi, Hazan, Singer'10

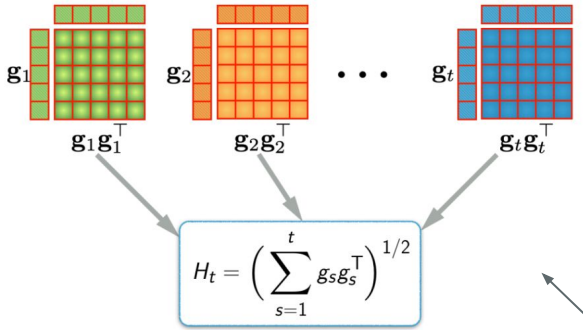
"Adaptive Bound Optimization for Online Convex Optimization", H. Brendan McMahan, Matthew Streeter'10

Full Matrix Adagrad Preconditioner



variants used in practice!

Full Matrix Adagrad Preconditioner



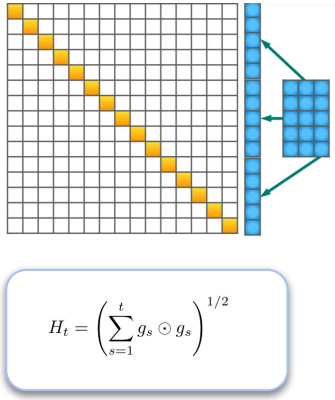
Infeasible!

Computational and Memory costs (for scale of models we train)

Eg: Fully connected layer: [m, n]

Memory: $O((mn)^2)$
Computation: $O((mn)^3)$

Diagonal AdaGrad

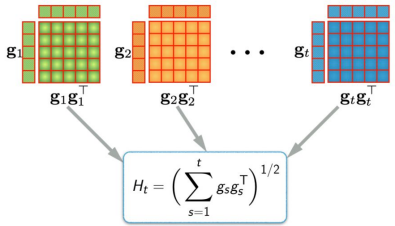


Memory: O(mn)
Computation: O(mn)

Is there something in between?

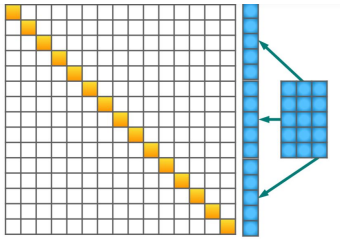
- Brings back correlations between gradients of parameters while being almost as efficient as diagonal AdaGrad?
- Cheaper in memory than diagonal AdaGrad?

Full Matrix Adagrad



?

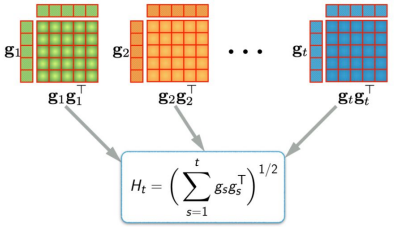
Diagonal AdaGrad



$$H_t = \left(\sum_{s=1}^{t} g_s \odot g_s\right)^{1/2}$$

?

Full Matrix Adagrad

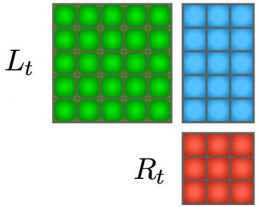


Adaptive Subgradient Methods for Online Learning and Stochastic Optimization
John Duchi, Elad Hazan, Yoram Singer'10

+

Adaptive Bound Optimization for Online Convex Optimization
H. Brendan McMahan, Matthew Streeter'10

Shampoo

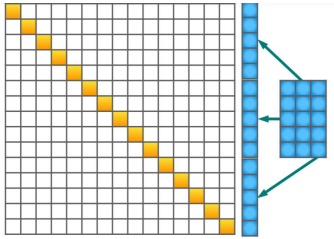


1. Shampoo: Preconditioned Stochastic Tensor Optimization
Vineet Gupta, Tomer Koren, Yoram Singer'18

2. Scalable Second Order Optimization for Deep Learning
Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, Yoram Singer'20

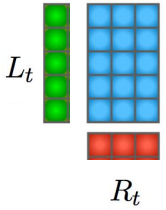
3. Disentangling Adaptive Gradient Methods from Learning Rates
Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, Cyril Zhang'20

Diagonal AdaGrad



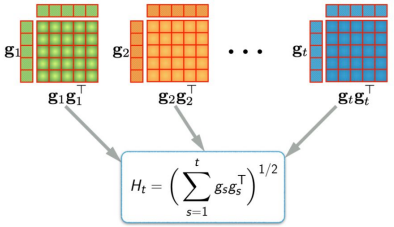
$$H_t = \left(\sum_{s=1}^{t} g_s \odot g_s\right)^{1/2}$$

SM3

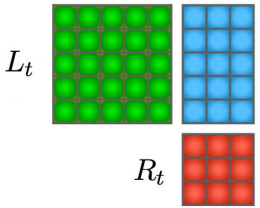


Memory Efficient Adaptive Optimization
Rohan Anil, Vineet Gupta, Tomer Koren, Yoram Singer', 19

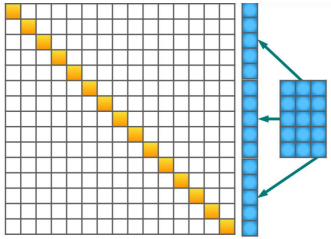
Full Matrix Adagrad



Shampoo

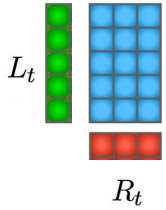


Diagonal AdaGrad



$$H_t = \left(\sum_{s=1}^{t} g_s \odot g_s\right)^{1/2}$$

SM3



Computational and Memory costs

Eg: Fully connected layer: [m, n]

Memory: $O((mn)^2)$
Computation: $O((mn)^3)$

Infeasible!
(for scale of models we train)

Memory: $O(m^2 + n^2)$
Computation: $O(m^3 + n^3)$

Memory: O(mn)
Computation: O(mn)

Sublinear!

Memory: O(m + n)
Computation: O(mn)

Shampoo = approx version of Full Matrix AdaGrad

- First approximation is to treat each layer independently (block diagonal)

- Use smaller matrices (of statistics) whose Kronecker product approximates the full matrix AdaGrad statistics.

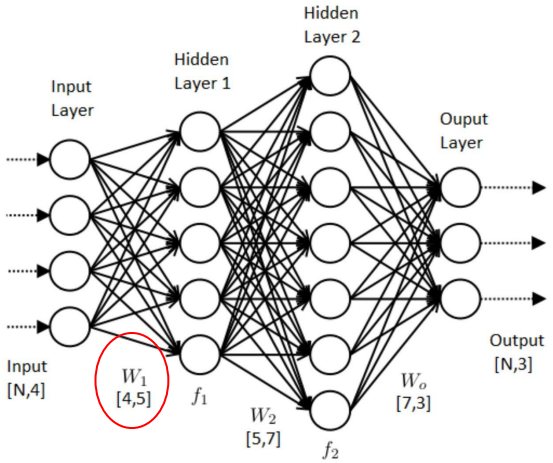
Simple example: 2D Tensors (fully connected)

Parameters of layer 1: $W_1 \in \mathbb{R}^{4\times 5}$

Gradient tensor: $G_t = \nabla_W \ell(f(W, x_t), y_t) \quad G_t \in \mathbb{R}^{4\times 5}$

Shampoo statistics: $L_t = \sum_{s=1}^{t} G_s G_s^\mathsf{T} \; ; \qquad R_t = \sum_{s=1}^{t} G_s^\mathsf{T} G_s$

$$L_t \in \mathbb{R}^{4\times 4}, R_t \in \mathbb{R}^{5\times 5}$$



Full matrix AdaGrad statistics: $H_t = \sum_{s=1}^{t} g_s g_s^\mathsf{T} \; , H \in \mathbb{R}^{20\times 20} \qquad g_t = \overline{\mathrm{vec}}(G_t), g_t \in \mathbb{R}^{20}$

$$\epsilon I_{mn} + \frac{1}{r}\sum_{s=1}^{t} g_s g_s^\mathsf{T} \preceq \left(\epsilon I_m + L_t\right)^{1/2} \otimes \left(\epsilon I_n + R_t\right)^{1/2}$$

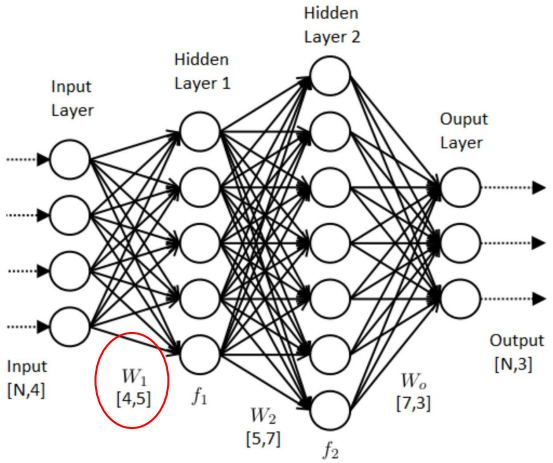
Simple example: 2D Tensors (fully connected)

Parameters of layer 1: $W_1 \in \mathbb{R}^{4\times 5}$

Gradient tensor: $G_t = \nabla_W \ell(f(W, x_t), y_t) \quad G_t \in \mathbb{R}^{4\times 5}$

Shampoo statistics: $L_t = \sum_{s=1}^{t} G_s G_s^\mathsf{T} \; ; \qquad R_t = \sum_{s=1}^{t} G_s^\mathsf{T} G_s$

$$L_t \in \mathbb{R}^{4\times 4}, R_t \in \mathbb{R}^{5\times 5}$$



Full matrix AdaGrad statistics: $H_t = \sum_{s=1}^{t} g_s g_s^\mathsf{T} \; , H \in \mathbb{R}^{20\times 20} \qquad g_t = \overline{\text{vec}}(G_t), g_t \in \mathbb{R}^{20}$

$$\frac{1}{} \sum_{s=1}^{t} g_s g_s^\mathsf{T} \preceq \left(\epsilon I_m + L_t\right)^{1/2} \otimes \left(\epsilon I_n + R_t\right)^{1/2}$$

```
jax.grad()
tf.gradients()
```

Simple example: 2D Tensors (fully connected)

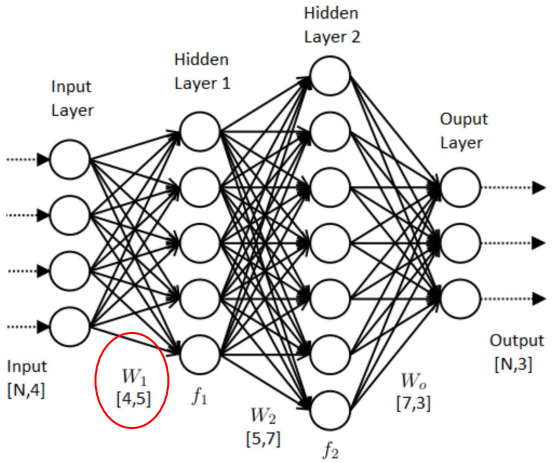
Parameters of layer 1: $W_1 \in \mathbb{R}^{4\times 5}$

Gradient tensor: $G_t = \nabla_W \ell(f(W, x_t), y_t) \quad G_t \in \mathbb{R}^{4\times 5}$

Shampoo statistics:

$$L_t = \sum_{s=1}^{t} G_s G_s^\mathsf{T} \; ; \qquad R_t = \sum_{s=1}^{t} G_s^\mathsf{T} G_s$$

$$L_t \in \mathbb{R}^{4\times 4}, R_t \in \mathbb{R}^{5\times 5}$$



Full matrix AdaGrad statistics: $H_t = \sum_{s=1}^{t} g_s g_s^\mathsf{T} \; , H \in \mathbb{R}^{20\times 20}$ $\quad g_t = \overline{\text{vec}}(G_t), g_t \in \mathbb{R}^{20}$

```
Let G = gradient.


L = L + G @ G.T
R = R + G.T @ G
```

$$\sum_{s=1}^{t} g_s g_s^\mathsf{T} \preceq \left(\epsilon I_m + L_t\right)^{1/2} \otimes \left(\epsilon I_n + R_t\right)^{1/2}$$

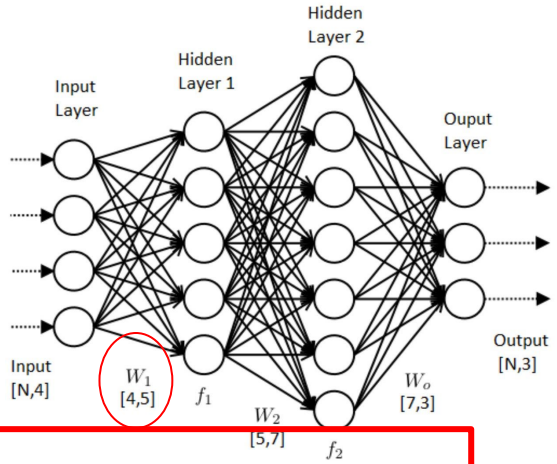
Simple example: 2D Tensors (fully connected)

Parameters of layer 1: $W_1 \in \mathbb{R}^{4\times 5}$

Gradient tensor: $G_t = \nabla_W \ell(f(W, x_t), y_t) \quad G_t \in \mathbb{R}^{4\times 5}$

Shampoo statistics: $L_t = \sum_{s=1}^{t} G_s G_s^\mathsf{T} \; ; \qquad R_t = \sum_{s=1}^{t} G_s^\mathsf{T} G_s$

$$L_t \in \mathbb{R}^{4\times 4}, R_t \in \mathbb{R}^{5\times 5}$$



Full matrix AdaGrad statistics: $H_t = \sum_{s=1}^{t} g_s g_s^\mathsf{T} \; , H \in \mathbb{R}^{20\times 20} \qquad g_t = \overline{\text{vec}}(G_t), g_t \in \mathbb{R}^{20}$

```
Let G be gradient
g = reshape(G, [-1])
H = H + g @ g.T
```

$$g_s^\mathsf{T} \preceq \left(\epsilon I_m + L_t\right)^{1/2} \otimes \left(\epsilon I_n + R_t\right)^{1/2}$$

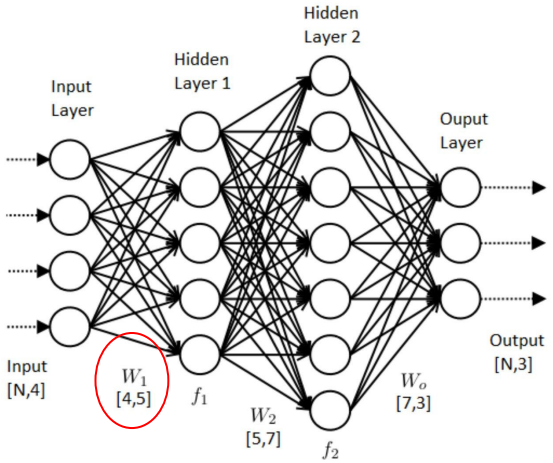
Simple example: 2D Tensors (fully connected)

Parameters of layer 1: $W_1 \in \mathbb{R}^{4\times 5}$

Gradient tensor: $G_t = \nabla_W \ell(f(W, x_t), y_t) \quad G_t \in \mathbb{R}^{4\times 5}$

Shampoo statistics: $L_t = \sum_{s=1}^{t} G_s G_s^\mathsf{T} \; ; \qquad R_t = \sum_{s=1}^{t} G_s^\mathsf{T} G_s$

$$L_t \in \mathbb{R}^{4\times 4}, R_t \in \mathbb{R}^{5\times 5}$$



Full matrix AdaGrad statistics: $H_t = \sum_{s=1}^{t} g_s g_s^\mathsf{T}$, $H \in \mathbb{R}^{20\times 20}$ $\qquad g_t = \overline{\mathrm{vec}}(G_t), g_t \in \mathbb{R}^{20}$

$$\epsilon I_{mn} + \frac{1}{r}\sum_{s=1}^{t} g_s g_s^\mathsf{T} \preceq \left(\epsilon I_m + L_t\right)^{1/2} \otimes \left(\epsilon I_n + R_t\right)^{1/2}$$

A B

Sketch of proof (Gupta, Koren, Singer, 2018)

SVD: $G = \sum_{i=1}^{r} \sigma_i u_i v_i^{\mathsf{T}}$ vectorized form: $g = \sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)$

Sketch of proof (Gupta, Koren, Singer, 2018)

SVD: $G = \sum_{i=1}^{r} \sigma_i u_i v_i^\mathsf{T}$ vectorized form: $g = \sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)$

$$
\begin{aligned}
gg^\mathsf{T} &= \Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)\Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)^\mathsf{T} \\
&\preceq r \sum_{i=1}^{r} \sigma_i^2 (u_i \otimes v_i)(u_i \otimes v_i)^\mathsf{T} \\
&= r \sum_{i=1}^{r} \sigma_i^2 (u_i u_i^\mathsf{T}) \otimes (v_i v_i^\mathsf{T}) \ .
\end{aligned}
$$

Sketch of proof (Gupta, Koren, Singer, 2018)

SVD: $G = \sum_{i=1}^{r} \sigma_i u_i v_i^\mathsf{T}$ vectorized form: $g = \sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)$

$$
\begin{aligned}
gg^\mathsf{T} &= \Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)\Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)^\mathsf{T} \\
&\preceq r \sum_{i=1}^{r} \sigma_i^2 (u_i \otimes v_i)(u_i \otimes v_i)^\mathsf{T} \\
&= r \sum_{i=1}^{r} \sigma_i^2 (u_i u_i^\mathsf{T}) \otimes (v_i v_i^\mathsf{T}) \ .
\end{aligned}
$$

$$
\Big(\sum_{i=1}^{r} w_i\Big)\Big(\sum_{i=1}^{r} w_i\Big)^\mathsf{T} \preceq r \sum_{i=1}^{r} w_i w_i^\mathsf{T}
$$

Sum of vectors

Sketch of proof (Gupta, Koren, Singer, 2018)

SVD: $G = \sum_{i=1}^{r} \sigma_i u_i v_i^\mathsf{T}$ vectorized form: $g = \sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)$

$$
\begin{aligned}
gg^\mathsf{T} &= \Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)\Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)^\mathsf{T} \\
&\preceq r \sum_{i=1}^{r} \sigma_i^2 (u_i \otimes v_i)(u_i \otimes v_i)^\mathsf{T} \\
&= r \sum_{i=1}^{r} \sigma_i^2 (u_i u_i^\mathsf{T}) \otimes (v_i v_i^\mathsf{T}) \ .
\end{aligned}
$$

$$
\Big(\sum_{i=1}^{r} w_i\Big)\Big(\sum_{i=1}^{r} w_i\Big)^\mathsf{T} \preceq r \sum_{i=1}^{r} w_i w_i^\mathsf{T}
$$

Sum of vectors

Scalars: $(a + b + c)(a + b + c) <= 3(a^2 + b^2 + c^2)$

Sketch of proof (Gupta, Koren, Singer, 2018)

SVD: $G = \sum_{i=1}^{r} \sigma_i u_i v_i^\mathsf{T}$ vectorized form: $g = \sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)$

$$
\begin{aligned}
gg^\mathsf{T} &= \Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)\Big(\sum_{i=1}^{r} \sigma_i (u_i \otimes v_i)\Big)^\mathsf{T} \\
&\le r \sum_{i=1}^{r} \sigma_i^2 (u_i \otimes v_i)(u_i \otimes v_i)^\mathsf{T} \\
&= r \sum_{i=1}^{r} \sigma_i^2 (u_i u_i^\mathsf{T}) \otimes (v_i v_i^\mathsf{T}) \ .
\end{aligned}
$$

$$\Big(\sum_{i=1}^{r} w_i\Big)\Big(\sum_{i=1}^{r} w_i\Big)^\mathsf{T} \le r \sum_{i=1}^{r} w_i w_i^\mathsf{T}$$

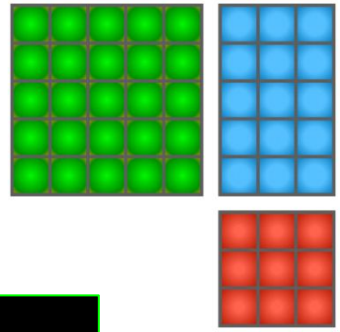
Sum of vectors

Since, $GG^\mathsf{T} = \sum_{i=1}^{r} \sigma_i^2 u_i u_i^\mathsf{T}$ and $v_i v_i^\mathsf{T} \le I_n$

$$\frac{1}{r} gg^\mathsf{T} \le \sum_{i=1}^{r} \sigma_i^2 (u_i u_i^\mathsf{T}) \otimes I_n = (GG^\mathsf{T}) \otimes I_n$$

Similarly, for R_t

Sketch of Update Rule for 2D Tensors



Update statistics:

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

```
Let G = gradient.


L = L + G @ G.T
R = R + G.T @ G
```

Compute update:

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

```
W = W - lr * invp(L, 4) @ G @ invp(R, 4)


invp(matrix, p) = inverse pth root of matrix
```

A comparison with K-FAC

(**Heskes**, 2000; **Martens** & **Grosse**, 2015; **Grosse** & **Martens**, 2016)

K-FAC is a natural gradient algorithm, and approximates the curvature of the loss using the Fisher Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)} \left[\nabla \log p(x|\theta) \, \nabla \log p(x|\theta)^\mathsf{T} \right] = \mathop{\mathbb{E}}_{p(x|\theta)} \left[g_{p(x|\theta)} \, g_{p(x|\theta)}^\mathsf{T} \right].$$

For a fully connected layer with $W \in \mathbb{R}^{m \times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m \times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t) x^\mathsf{T}$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

$$\begin{aligned} \mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \otimes x) \, (\nabla_s \ell(s_t, y_t) \otimes x)^\mathsf{T} \right] \\ &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \otimes (x_t x_t^\mathsf{T}) \right]. \end{aligned}$$

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and x, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right] \otimes \mathbb{E} \left[x_t x_t^\mathsf{T} \right].$$

If we let $D = \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right]$ and $X = \mathbb{E} \left[x_t x_t^\mathsf{T} \right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta \, D^{-1} G_t X^{-1}.$$

A comparison with K-FAC

(**Heskes**, 2000; **Martens** & **Grosse**, 2015; **Grosse** & **Martens**, 2016)

K-FAC is a natural gradient algorithm, and approximates the curvature of the loss using the Fisher Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)} \left[\nabla \log p(x|\theta) \, \nabla \log p(x|\theta)^\mathsf{T} \right] = \mathop{\mathbb{E}}_{p(x|\theta)} \left[g_{p(x|\theta)} \, g_{p(x|\theta)}^\mathsf{T} \right].$$

Gradient outer product!

For a fully connected layer with $W \in \mathbb{R}^{m \times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m \times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t) x^\mathsf{T}$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

$$\begin{aligned} \mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \otimes x) \, (\nabla_s \ell(s_t, y_t) \otimes x)^\mathsf{T} \right] \\ &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \otimes (x_t x_t^\mathsf{T}) \right]. \end{aligned}$$

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and x, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right] \otimes \mathbb{E} \left[x_t x_t^\mathsf{T} \right].$$

If we let $D = \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right]$ and $X = \mathbb{E} \left[x_t x_t^\mathsf{T} \right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta \, D^{-1} G_t X^{-1}.$$

A comparison with K-FAC

(**Heskes**, 2000; **Martens** & **Grosse**, 2015; **Grosse** & **Martens**, 2016)

```
// x = [1, M], s = [1, N]
// W = [M, N], G = [M, N]

x @ W = s

Loss = f(s)

G = x.T @ grad(Loss, s)

// Express the following as:
//  x ⊗ grad(Loss, s)
g = reshape(G, [-1])
```

K-FAC is a natural gradient algorithm, and approximates the curvature of the l
Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)} \left[\nabla \log p(x|\theta) \, \nabla \log p(x|\theta)^\mathsf{T} \right] = \mathop{\mathbb{E}}_{p(x|\theta)} \left[g_{p(x|\theta)} \, g_{p(x|\theta)}^\mathsf{T} \right]$$

For a fully connected layer with $W \in \mathbb{R}^{m \times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m \times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t) x^\mathsf{T}$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

Chain rule

$$\begin{aligned} \mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \otimes x) \, (\nabla_s \ell(s_t, y_t) \otimes x)^\mathsf{T} \right] \\ &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \otimes (x_t x_t^\mathsf{T}) \right]. \end{aligned}$$

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and x, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right] \otimes \mathbb{E} \left[x_t x_t^\mathsf{T} \right].$$

If we let $D = \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right]$ and $X = \mathbb{E} \left[x_t x_t^\mathsf{T} \right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta \, D^{-1} G_t X^{-1}.$$

A comparison with K-FAC

(**Heskes**, 2000; **Martens** & **Grosse**, 2015; **Grosse** & **Martens**, 2016)

```
// Express the following as:
//  x ⊗ grad(Loss, s)
g = reshape(G, [-1])


Sum over np.outer(g, g)
```

K-FAC is a natural gradient algorithm, and approximates the curvature of the loss using the Fisher Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)} \left[\nabla \log p(x|\theta) \, \nabla \log p(x|\theta)^\mathsf{T} \right] = \mathop{\mathbb{E}}_{p(x|\theta)} \left[g_{p(x|\theta)} \, g_{p(x|\theta)}^\mathsf{T} \right].$$

For a fully connected layer with $W \in \mathbb{R}^{m \times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m \times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t) x^\mathsf{T}$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

$$\begin{aligned} \mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \otimes x) \, (\nabla_s \ell(s_t, y_t) \otimes x)^\mathsf{T} \right] \\ &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \otimes (x_t x_t^\mathsf{T}) \right] \end{aligned}$$

Rearrange

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and x, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right] \otimes \mathbb{E} \left[x_t x_t^\mathsf{T} \right].$$

If we let $D = \mathbb{E} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \right]$ and $X = \mathbb{E} \left[x_t x_t^\mathsf{T} \right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta \, D^{-1} G_t X^{-1}.$$

A comparison with K-FAC

(**Heskes**, 2000; **Martens** & **Grosse**, 2015; **Grosse** & **Martens**, 2016)

```
// x = [1, M], s = [1, N]
// W = [M, N], G = [M, N]

x @ W = s

Loss = f(s)

G = x.T @ grad(Loss, s)

// K-FAC Statistics `D` and `X`
D = grad(Loss, s).T @ grad(Loss, s)
X = x.T @ x
```

K-FAC is a natural gradient algorithm, and approximates the curvature of the l
Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)} \left[\nabla \log p(x|\theta)\, \nabla \log p(x|\theta)^\mathsf{T}\right] = \mathop{\mathbb{E}}_{p(x|\theta)} \left[g_{p(x|\theta)}\, g_{p(x|\theta)}^\mathsf{T}\right]$$

For a fully connected layer with $W \in \mathbb{R}^{m \times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m \times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t) x^\mathsf{T}$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

$$\begin{aligned}\mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \otimes x)\, (\nabla_s \ell(s_t, y_t) \otimes x)^\mathsf{T}\right] \\ &= \mathop{\mathbb{E}}_{p(x|\theta)} \left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T}) \otimes (x_t x_t^\mathsf{T})\right].\end{aligned}$$

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and x, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E}\left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T})\right] \otimes \mathbb{E}\left[x_t x_t^\mathsf{T}\right].$$

If we let $D = \mathbb{E}\left[(\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\mathsf{T})\right]$ and $X = \mathbb{E}\left[x_t x_t^\mathsf{T}\right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta\, D^{-1} G_t X^{-1}.$$

Shampoo update rule

Update statistics:

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

Compute update:

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

Convert one to another

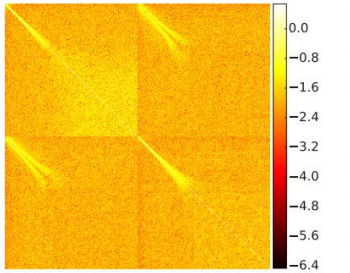
$$G_t G_t^\intercal = \nabla_s \ell(s_t, y_t) x_t^\intercal x_t \nabla_s \ell(s_t, y_t)^\intercal = \|x_t\|_2^2 \nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\intercal;$$
$$G_t^\intercal G_t = x_t \nabla_s \ell(s_t, y_t)^\intercal \nabla_s \ell(s_t, y_t) x_t^\intercal = \|\nabla_s \ell(s_t, y_t)\|_2^2 x_t x_t^\intercal.$$

Dividing by the scale, and taking expectations on both sides:

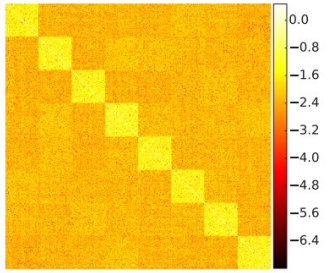
$$\mathbb{E}\left[\frac{G_t G_t^\intercal}{\|x_t\|_2^2}\right] = \mathbb{E}\left[\nabla_s \ell(s_t, y_t) \nabla_s \ell(s_t, y_t)^\intercal\right] = D;$$
$$\mathbb{E}\left[\frac{G_t^\intercal G_t}{\|\nabla_s \ell(s_t, y_t)\|_2^2}\right] = \mathbb{E}\left[x_t x_t^\intercal\right] = X.$$

- This exposition is to show their similarity in construction (when batch =1)
- Differences based on choices such as
  - empirical fisher or fisher
  - moving averages vs sum
  - inverse exponents (1/2 for Full Matrix AdaGrad, 1 for Online Newton Step)
- Shampoo always uses the mini-batch gradient in our experiments.
- Another key difference is that Shampoo construction is agnostic to layer types.

What do preconditioners look like?

Preconditioned Gradient = 

(a) $L_t^{-1/4}$ of attention layer

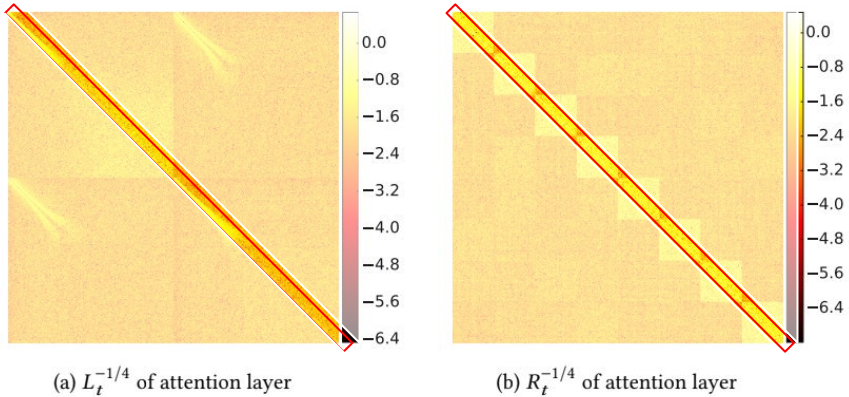
@ Gradient @ 

(b) $R_t^{-1/4}$ of attention layer

- There appears to be structure in the preconditioners. Snapshot of the preconditioner from the Transformers for language translation task.
- We notice **~30% the preconditioned gradient** changes sign.

Related work: What is SM3?

1. A sub-linear memory optimizer. Useful training models under a memory constraint (say larger models) Paper
2. Think of it as the diagonals of Shampoo
3. SM3 is a tighter estimate than what can be found via Kronecker product of diagonals of Shampoo, for estimating the diagonal entries of Full Matrix AdaGrad.

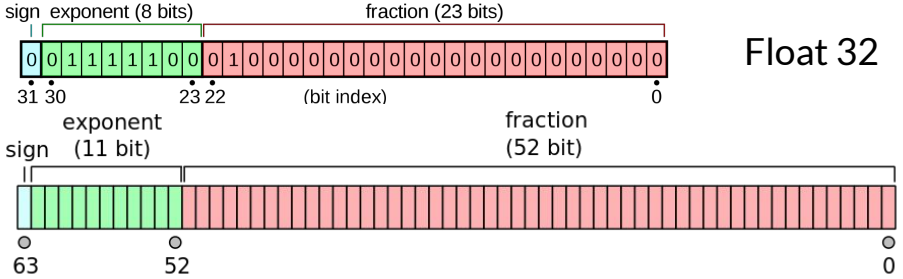


(a) $L_t^{-1/4}$ of attention layer (b) $R_t^{-1/4}$ of attention layer

Inverse Pth Roots of ill-conditioned matrices (M)

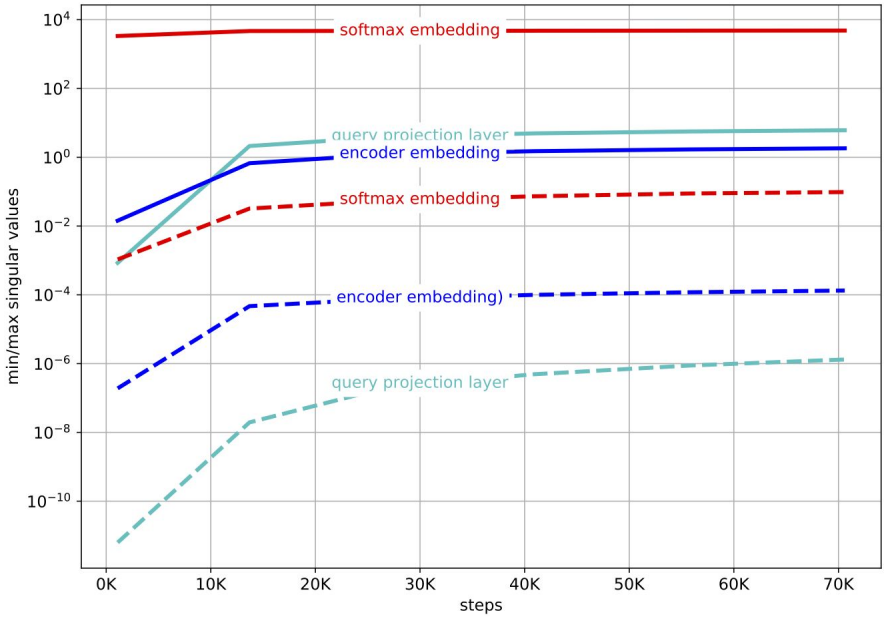
Rule of thumb (Overton, 2001): Computing inverse pth root loses $\log_2(1/p\ \kappa(M))$ bits of precision.

$$\kappa(M) = |\lambda_{max}| / |\lambda_{min}|$$



- A condition number of 10^6 loses 19 bits of precision; left with 4 bits in single precision fp32 for inverse 4th root

Inverse Pth Roots of ill-conditioned matrices (M)



Fast matrix inverse pth roots

$$X^p A - I = 0$$

Option a: Use classical Newton's method:

$$X_{k+1} = \frac{1}{p} X_k [(p+1)I - X_k^p A]$$

However, *numerically unstable* and requires certain assumptions on eigenvalues of **A**

Simple **modification** to form a stable iteration (Iannazzo et al): $M_k = X_k^p A$

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

} Coupled iterations

code

Neural Network accelerators prefer *lower precision*

- For good reasons - higher precision acceleration is expensive.
- Making a second order method work at scale for neural network training is **precisely** (no pun intended) a major part of our work, other part was the details around what we now call grafting, numerics..

Preconditioning at Large Scale Settings

- Heterogeneous compute: Make use of the CPUs attached to the accelerator to compute inverse pth roots
- Pipeline the computation with the training steps.

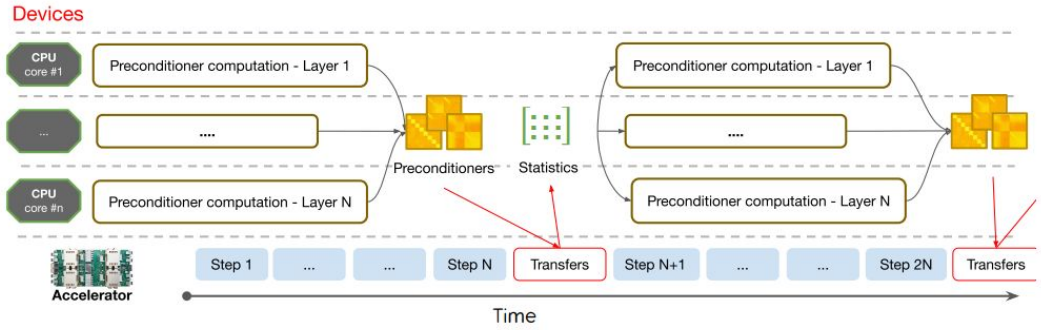
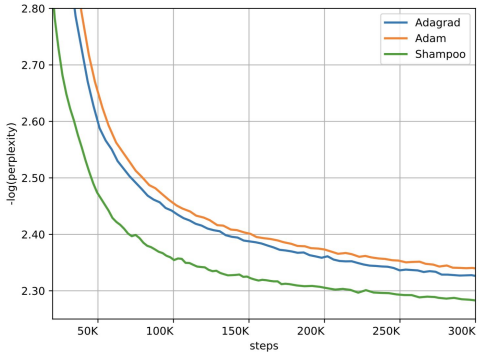


Figure 1: Timeline illustrating the design of the optimization algorithm. Preconditioner statistics (L_t and R_t) are computed at each step by the accelerators. Preconditioners ($L_t^{1/4}$, $R_t^{1/4}$) are computed every N steps and this computation is distributed to all available CPU cores.

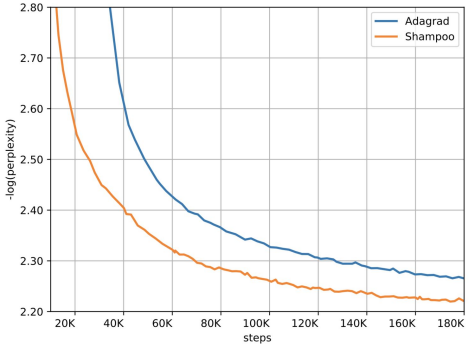
Results

Translation with a Transformer: English to French

- Standard WMT'14 translation dataset
  - 36.3M sentence pairs
- Transformer: 93.3M parameters
- 32 cores of TPU-v3
  - Batch size: 1536
- **1.95x fewer steps**
- **40% less wallclock time**

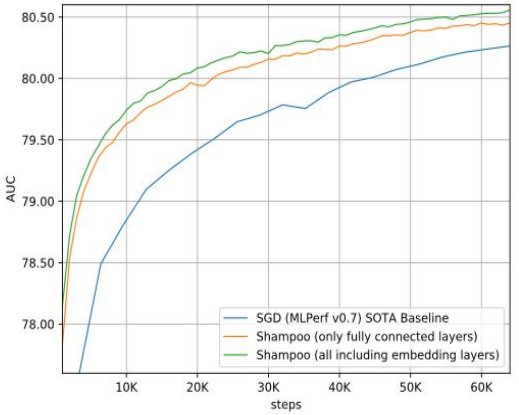


- Standard WMT'14 translation dataset
  - 36.3M sentence pairs
- Transformer Big: 340M parameters
- 32 cores of TPU-v3
  - Batch size: 1536
- **2x fewer steps**
- **41% less wallclock time**



DLRM:
Criteo pCTR prediction task

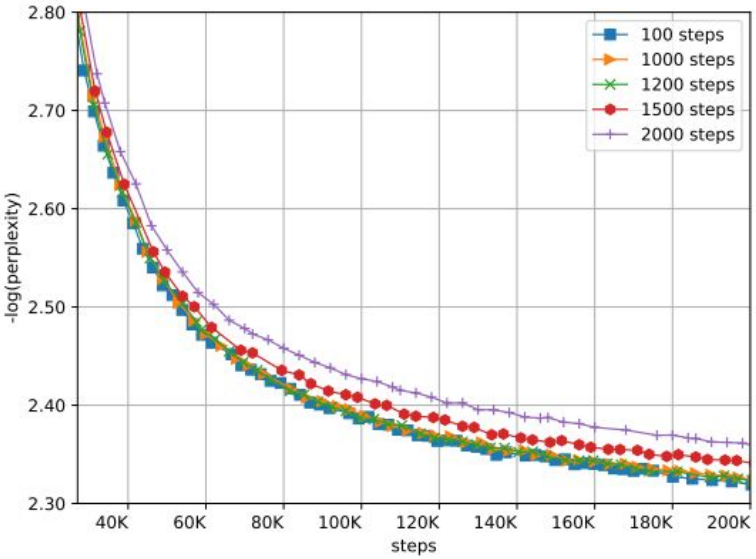
- Shampoo reaches a target AUC of 80.25% in **half as many steps** with preconditioning embedding layers improving the results, and achieves a new state-of-the-art AUC of **80.56%;**



How often to run preconditioning?

It depends

- For a ***relatively large batch size***, each gradient step makes much larger progress which seem to require computing preconditioners to be computed more frequently.
- For **smaller batch sizes**, which is ***true for majority of NLP pipelines***, we expect we can tolerate large delays. This is what we see in one example (being tolerant to delays upto 1200 steps)



Preconditioner computation run every N steps for a Transformer for Machine Translation

Step time for a Transformer model

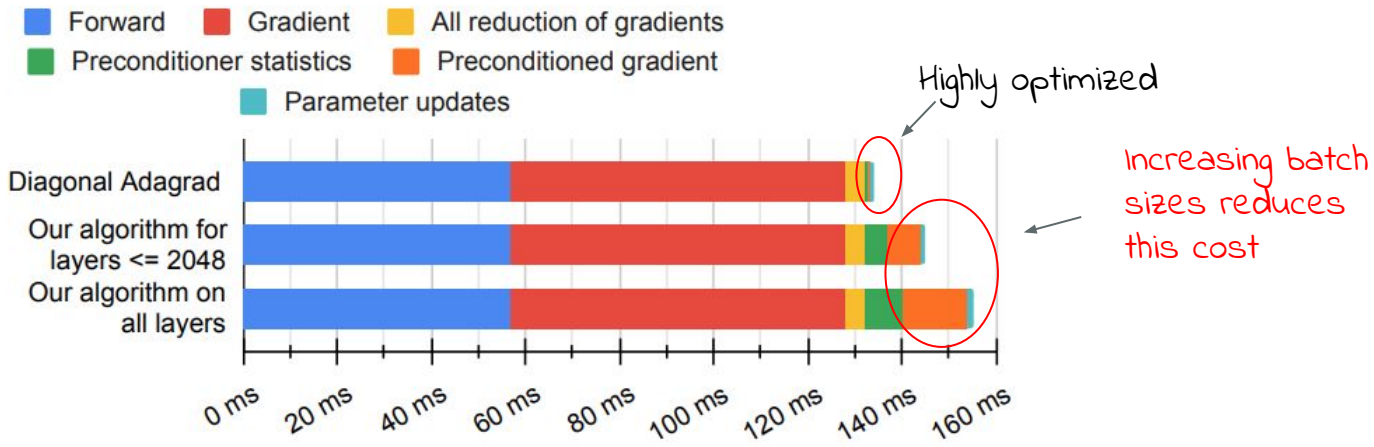
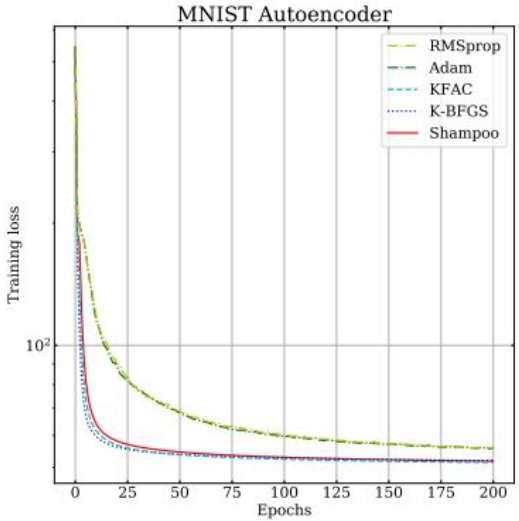
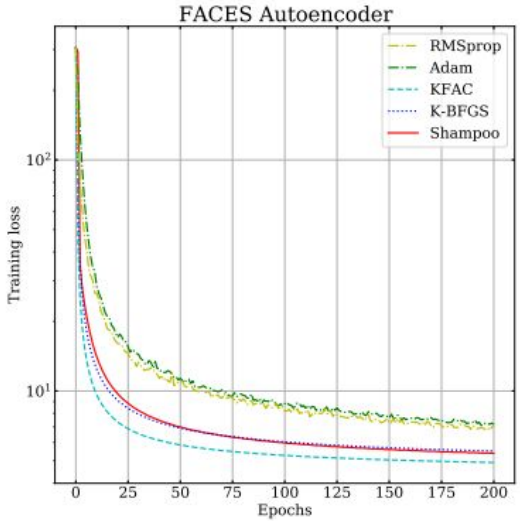
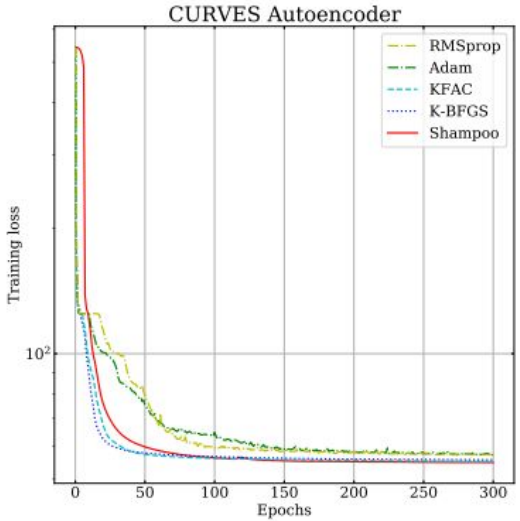


Figure 5: Detailed breakdown of latency of a single step. Diagonal AdaGrad optimizer: 134ms, our implementation of Shampoo: 145ms (for all layers except embedding and softmax layers) and 155ms (for all layers). As preconditioner computation is pipelined and distributed over CPUs it does not add any overhead, and transfer latency is minimal (≈100ms) and is amortized over hundreds of steps.

Second order methods: Deep Autoencoder Task







- Based on code from K-BFGS Pytorch implementation.
- Shampoo seems to work just as well others, except in FACES task where K-FAC is better.
- Shampoo only relies on the ***gradient information***, and the ***shape of the layer***
  - No per example gradients required and agnostic to layer types (batch norm, convolution, ..)

Growth of model size in NLP

Model	Number of parameters
Transformer (translation) Chen et al 2018	375.4M
BERT (language model) Devlin et al 2018	340M
GPT-2 Radford et al, 2019	1.5B
GPT-3 Brown et al, 2020	175B

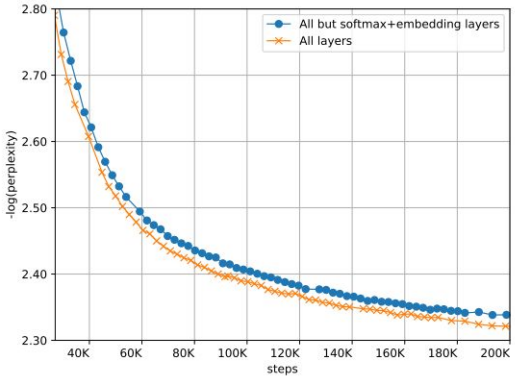
- Model size increase from:
  - Increasing the number of layers (stacking)
  - Or **increasing layer width**

Memory: $O(m^2 + n^2)$
Computation: $O(m^3 + n^3)$

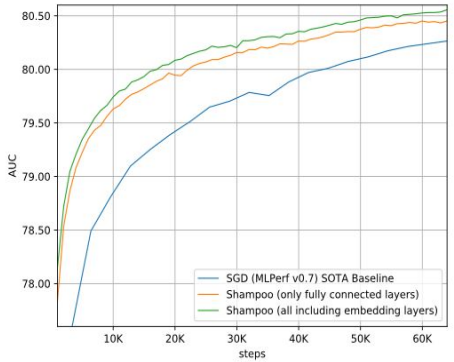
Preconditioning extremely large layers

Embedding layers (a very large rectangular layer)

1. Medium sized embedding layers, make use of only the smaller preconditioner
2. Very large embedding layer, exploit sparsity, compute gradient with respect to the lookup, and use that to compute the preconditioner.



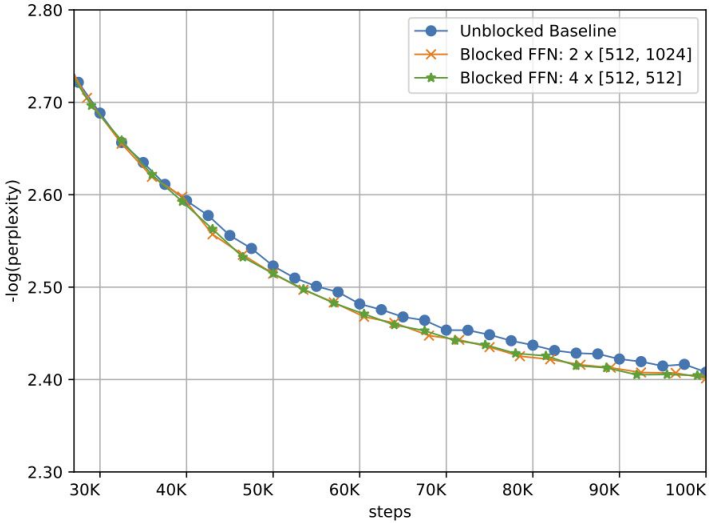
Shampoo on all layers vs excluding embedding/softmax layers on a Transformer for Machine Translation



Shampoo on all layers vs exclude embedding layers on a Transformer for DLRM Recommendation Model

Preconditioning extremely large layers

- **W: [24K, 24K]** fully connected layer, compute preconditioners for: [1024, 1024]. Reduce **computational costs**!
- We use a block size of 128x128 for **ResNet-50 training** (shown later)
- We **also** reshape gradients.
  - [1, 3, 3, 1024, 1024] -> [9, 1024, 1024]



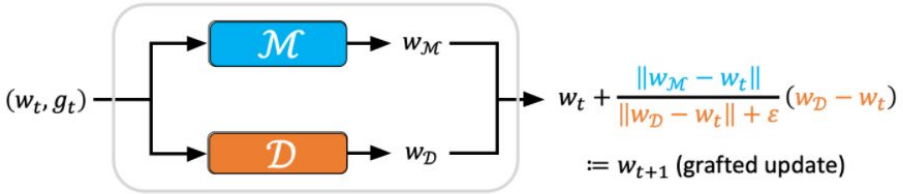
Shampoo with different blocking configuration on Transformer for Machine Translation

Key: Learning rate schedules

- Single most important factor with first order optimization methods
- Confounding variable
  - some provide implicit decay 1/sqrt(T)
  - others have constant step size and requires external schedule
- We studied this on wide range of Direction/Magnitude combinations:
  - *"Disentangling Adaptive Gradient Methods from Learning Rates", Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, Cyril Zhang, https://arxiv.org/pdf/2002.11803.pdf*

Idea:

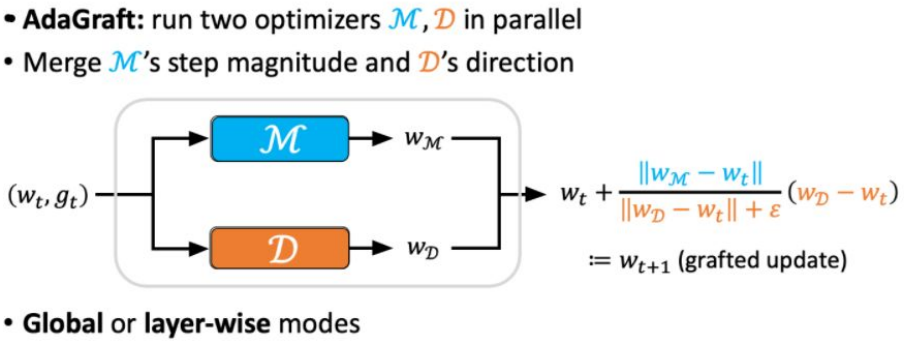
- **AdaGraft:** run two optimizers $\mathcal{M}, \mathcal{D}$ in parallel
- Merge $\mathcal{M}$'s step magnitude and $\mathcal{D}$'s direction



- **Global** or **layer-wise** modes

Is your Optimizer 1 better than Optimizer 2?

- Try grafting Optimizer 1's ***layerwise update magnitude*** onto Optimizer 2 and retune.
- Generally we see following
  - Optimizer that didn't work on a problem, magically works now
  - Allows us to ***bootstrap*** on a new problem that is heavily hyperparameter tuned.
- Shampoo chooses to graft the magnitude from SGD or AdaGrad. Both are cheap to compute. Thus, Shampoo is only used for computing the direction of the update.



code

Alternate design choice: Emulating higher precision on accelerators

- Higher precision can be emulated using bfloat16 numerics.
  a. G. Henry, P. T. P. Tang, and A. Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pages 69–76. IEEE, 2019.
     - https://arxiv.org/abs/1904.06376

- **Which architecture to use? Tradeoffs!**
  a. Communication overhead between CPU <-> Accelerator
  b. Number of preconditioners: Parallelism available on CPU vs Accelerator
  c. Staleness tolerance of the preconditioner (large batch vs small batch)
  d. How long does the training step take (without including the preconditioner computation)

Fastest

ResNet-50 training at large batch sizes

Optimizer	Steps to reach 75.9 validation accuracy	Wall clock time
LARS	2512	~309-311 seconds
Nesterov	2512	~309-311 seconds
Distributed Shampoo (this work)	**1729 (31.17 % reduction)**	**~267-269 seconds**

Code is available (with more details)

- Batch size: **32,768**
- Same benchmarking hardware
- Blocked preconditioning (128x128 blocks)
- Runs inverse computation **every step!**

Concluding remarks

- Innovations in compiler or runtime stack that can make it easy to write efficient heterogeneous pipelined computations.
- Ways to exploit parallelism thats available in the optimizer that doesn't add too much code complexity, making it easy to integrate to rest of the training pipeline.
- Second order methods discussed here all rely on using **Symmetric** Matrix Multiplies in many of the operations. (a) **save half the memory** by storing upper triangular matrix efficiently (b) matrix multiplies of symmetric matrices can be optimized
- ML libraries with linear algebra routines that can run on accelerators.
- Mixed precision algorithms for inverse roots, faster variants of higher precision emulation can all reduce the computational complexity of inverse roots.

Thank you!

https://arxiv.org/abs/2002.09018

```
@misc{anil2021scalable,
    title={Scalable Second Order Optimization for Deep Learning},
    author={Rohan Anil and Vineet Gupta and Tomer Koren and Kevin Regan and Yoram Singer},
    year={2021},
    eprint={2002.09018},
    archivePrefix={arXiv},
    primaryClass={cs.LG}
}
```

Feels like we are just getting started with this stuff!

Please email me (**rohananil** at **google** dot **com**) as for further questions or collaborations.

Google Research, Brain Team

Kronecker product rules: from Matrix Cookbook

1. Inverse of Kronecker Product

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \quad (512)$$

2. Mixed product property

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD} \quad (511)$$

3. Matrix Multiply:

$$\mathrm{vec}(\mathbf{AXB}) = (\mathbf{B}^T \otimes \mathbf{A})\mathrm{vec}(\mathbf{X}) \quad (520)$$