Convolutional Neural Network Training with Distributed K-FAC

J. Gregory Pauloski*, Zhao Zhang †, Lei Huang †, Weijia Xu †, Ian T. Foster *‡

*University of Chicago
†Texas Advanced Computing Center
‡Argonne National Laboratory

Large Batch Training

Large batch deep learning training is essential as:

- model parameter counts increase,
- dataset sizes increase,
- overall training times increase.

Large batch training can be difficult:

- Converge to sharp minima leading to poor generalization.
- Requires additional tricks to maintain generalization. E.g. LR scaling and warmup, LARS, batch size warmup, etc.

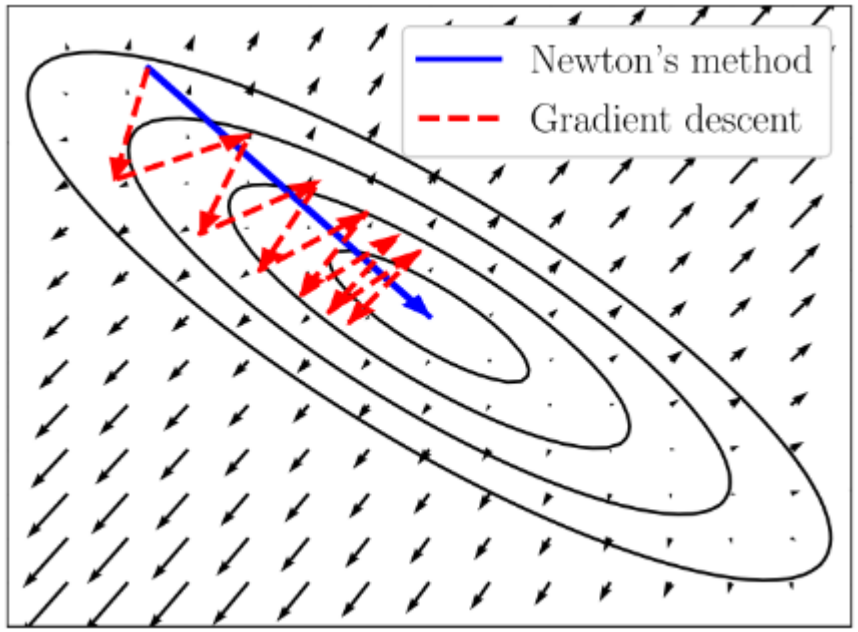
Observations

Large batch training is a good candidate for second-order optimization.

1. Large batches are more representative of the dataset's distribution.
    - Enables second-order update decoupling.
2. Gradient noise limits the maximum effective batch size and increases over the course of training (McCandlish, 2018).
    - Second-order methods optimize noise-independent terms better (Martens, 2014).
3. Second-order methods have higher computation-to-communication ratios.
    - Enables greater benefits from layer-wise distribution schemes.

Second-Order Optimization

- Second-order methods incorporates the curvature of the parameter space.
- Makes more per-iteration progress optimizing the objective function than first-order methods.
- Examples: Gauss-Newton, (L)BFGS, K-FAC
- **Expensive to compute!**



https://www.diva-portal.org/smash/get/diva2:1437676/FULLTEXT01.pdf

Kronecker-Factored Approximate Curvature

- K-FAC is an efficient approximation of the Fisher Information Matrix (FIM) (Martens+, 2015).
  - The FIM is equivalent to the Generalized Gauss-Newton (GGN) matrix, an approximation of the Hessian.
- Generalizes better with large batches and converges in fewer iterations than SGD (Ba+, 2017).
- Scales to extremely large batch sizes, e.g. 131k for ImageNet training (Osawa+, 2019).

Kronecker-Factored Approximate Curvature

SGD: $\omega^{(k+1)} = \omega^{(k)} - \frac{\alpha^{(k)}}{n} \sum_{i=1}^{n} \nabla L_i(\omega^{(k)})$

K-FAC: $\omega^{(k+1)} = \omega^{(k)} - \frac{\alpha^{(k)} \hat{F}^{-1}(\omega^{(k)})}{n} \sum_{i=1}^{n} \nabla L_i(\omega^{(k)})$

$\omega^{(k)}$ = weight at iteration *k*

$\alpha^{(k)}$ = learning rate at iteration *k*

n = minibatch size

$\nabla L_i(\omega^{(k)})$ = gradient of the loss function L_i for the i^{th} example with respect to $\omega^{(k)}$

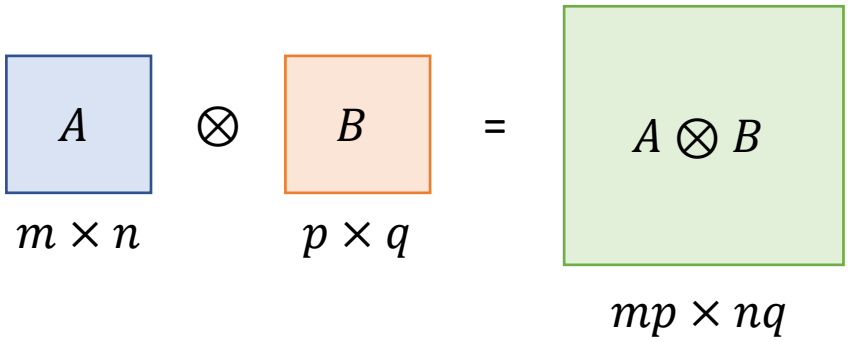
$\hat{F}^{-1}$ = Inverse Fisher Information Matrix approximation, acts as a gradient preconditioner

Kronecker Product

Kronecker Product: $\otimes$

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 1 \times 6 & 2 \times 5 & 2 \times 6 \\ 1 \times 7 & 1 \times 8 & 2 \times 7 & 2 \times 8 \\ 1 \times 9 & 1 \times 0 & 2 \times 9 & 2 \times 0 \\ 3 \times 5 & 3 \times 6 & 4 \times 5 & 4 \times 6 \\ 3 \times 7 & 3 \times 8 & 4 \times 7 & 4 \times 8 \\ 3 \times 9 & 3 \times 0 & 4 \times 9 & 4 \times 0 \end{bmatrix}$$

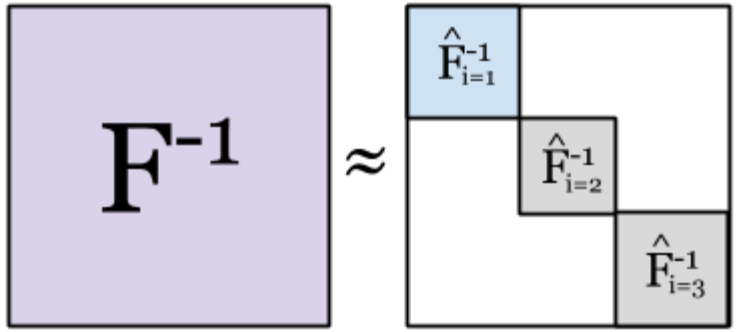


Kronecker-Factored Approximate Curvature

- K-FAC approximates the FIM as a block diagonal matrix.

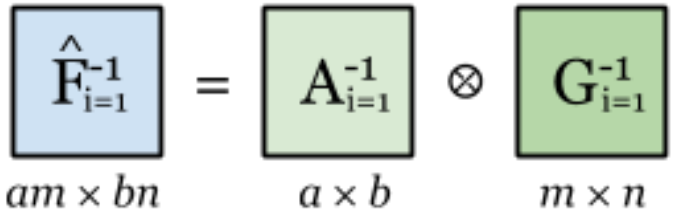
$$F \cong \hat{F} = diag\left(\widehat{F_1}, \ldots, \widehat{F_i}, \ldots, \widehat{F_L}\right)$$

$$\hat{F}^{-1} = diag\left(\widehat{F_1}^{-1}, \ldots, \widehat{F_i}^{-1}, \ldots, \widehat{F_L}^{-1}\right)$$



- $\widehat{F_i}$ is a Kronecker product of the activations of the $(i-1)^{th}$-layer and the gradient w.r.t output of i^{th}-layer.

$$\widehat{F_i} = a_{i-1}a_{i-1}^T \otimes g_i g_i^T = A_{i-1} \otimes G_i$$



Kronecker-Factored Approximate Curvature

Kronecker Product Properties:

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

$$(A \otimes B)\vec{c} = B^T \vec{c} A$$

KFAC update step for layer i:

$$\omega_i^{(k+1)} = \omega_i^{(k)} - \alpha^{(k)} \boxed{\hat{F}_i^{-1} \nabla L_i(\omega_i^{(k)})}$$

where*:

$$\hat{F}_i^{-1} \nabla L_i\left(\omega_i^{(k)}\right) = (A_{i-1} \otimes G_i)^{-1} \nabla L_i(\omega_i^{(k)})$$

$$\hat{F}_i^{-1} \nabla L_i\left(\omega_i^{(k)}\right) = (A_{i-1}^{-1} \otimes G_i^{-1}) \nabla L_i(\omega_i^{(k)})$$

$$\hat{F}_i^{-1} \nabla L_i\left(\omega_i^{(k)}\right) = \boxed{G_i^{-1} \nabla L_i(\omega_i^{(k)})\, A_{i-1}^{-1}}$$

A^{-1} and G^{-1} are symmetric.

Preconditioned Gradient

***Note:** In practice, $(\hat{F}_i + \gamma I)^{-1} \nabla L_i\left(\omega_i^{(k)}\right)$ is computed to prevent ill-conditioned matrix inversion. γ is a damping constant.

Large-Batch Distributed K-FAC

Ba, Grosse, and Martens (2017)

- TensorFlow Parameter Server
- Reports 2x training time improvements over SGD for ResNet-50 training on ImageNet with 4 GPUs.
- Baseline SGD implementation only achieved ~70% validation accuracy.

Osawa et al. (2019)

- Synchronous, data-parallel scheme in Chainer
- Layer-wise distribution scheme. Worse scaling when n_workers > n_layers as workers are left idle.
- 75% validation accuracy for ResNet-50 training on ImageNet with 131k batch size in 978 iterations.
- However, SGD still achieves better validation accuracy is less time.

Goals

Distributed K-FAC optimization strategy that:

1. Maintains validation accuracy comparable to SGD at large batch sizes.
2. Converges in faster time and fewer iterations than SGD.
3. Open source and easy to incorporate into existing training scripts.

Design: Matrix Inversion

- K-FAC update step requires computing A^{-1} and G^{-1} for every layer to compute the preconditioned gradient.
- Preconditioned gradient can alternatively be computed using an eigen-decomposition of A and G.
- Q_A, v_A: eigenvectors and eigenvalues of A.
- Q_G, v_G: eigenvectors and eigenvalues of G.
- Empirically, a more stable approximation for the FIM.

$$V_1 = Q_G^T \nabla L_i \left(\omega_i^{(k)}\right) Q_A$$

$$V_2 = V_1 / (v_G v_A^T + \gamma)$$

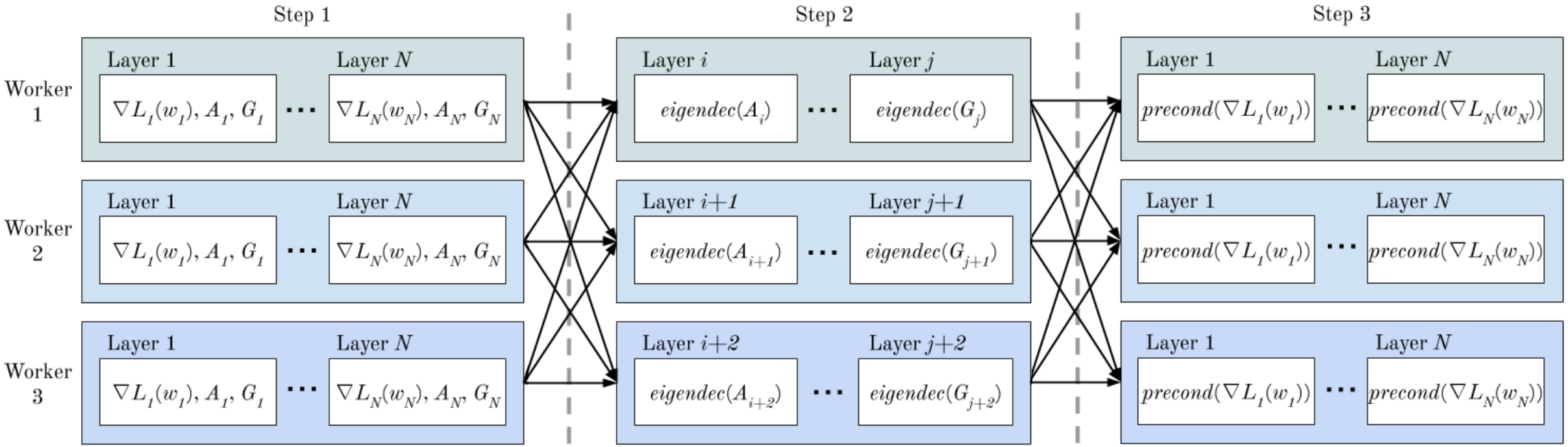
$$(\widehat{F}_i + \gamma I)^{-1} \nabla L_i \left(\omega_i^{(k)}\right) = Q_G V_2 Q_A^T$$

Design: Parallelism

Idea: Assign each factor A_i and G_i to a different worker to be eigen decomposed then broadcast the results to all workers for gradient preconditioning.

Standard **data parallel** training. I.e. each worker maintains a local copy of the model and computes the forward and backward pass for a local mini-batch then the gradients are reduced across workers.

Before the SGD optimization step: distribute and compute the factors, broadcast the results to all workers, and precondition the gradients in-place.



Step 1:

- Compute forward/backward pass for local batch.
- Compute A and G for local batch.

Step 2:

- Each worker eigen decomposes the factors it was assigned.

Step 3:

- Each worker computes the preconditioned gradient for all layers.

reduce()

broadcast()

Design: Communication Optimizations

- Communication in three places:
  a) Average gradients ($\nabla L_i(\omega_i^{(k)})$)
  b) Average Kronecker factors (A_i, G_i)
  c) Broadcast eigen decompositions ($Q_{A_i}, Q_{G_i}, v_{A_i}, v_{G_i}$)
- A common second-order optimization strategy is to only update the second-order information every n-iterations.
- Thus, in non-KFAC update steps, we **reuse the eigen decompositions** from previous iterations and **avoid the communication in (b) and (c).**

Design: Communication Optimizations

Tradeoffs:

- Eigendecompositions become stale over time.
- KFAC update steps require more communication (factors + eigendecompositions).
- Non-KFAC update steps require **only** communicating the gradients, the **same amount of communication as SGD**.
- Communication decreases and update interval increases.
- Less communication than previous distributed K-FAC methods.

Implementation

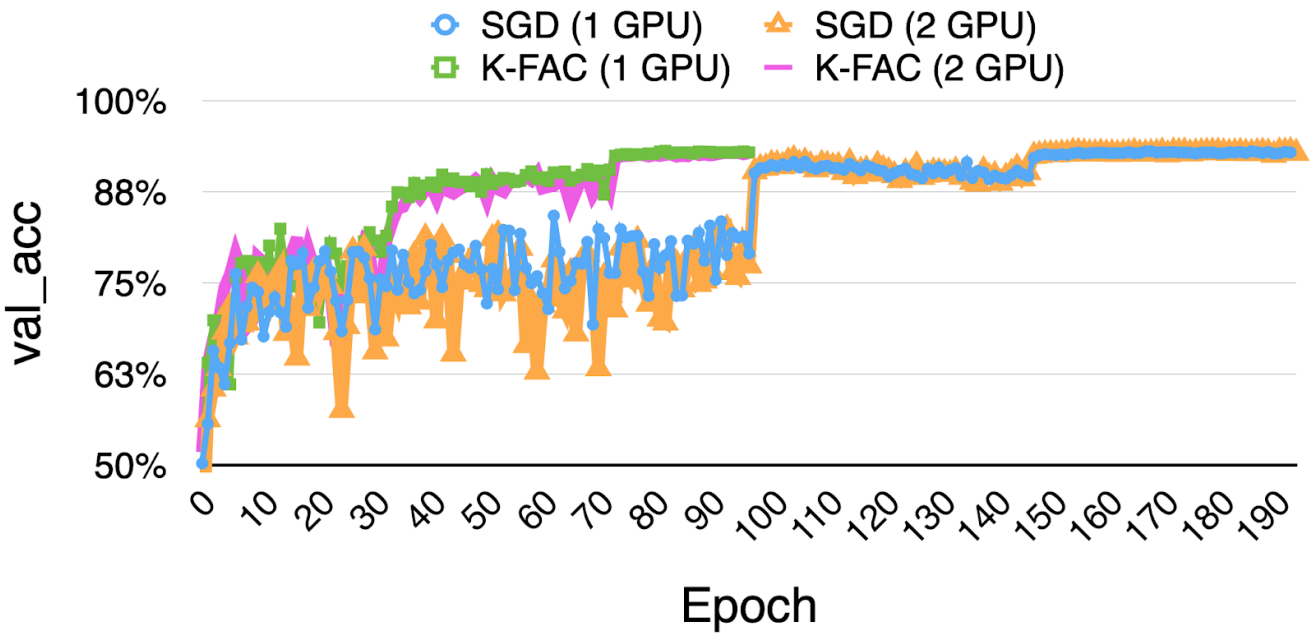
- K-FAC modifies gradients in-place after the backward pass and before the optimization step.
- Allows for compatibility with any PyTorch optimizer or preconditioner (e.g. SGD, Adam, LARS, LAMB, etc.).
- Linear and Conv2D layers.
- Horovod and torch.distributed data-parallel training backends.

Horovod Example Usage

```
...

optimizer = optim.SGD(model.parameters(), ...)
optimizer = hvd.DistributedOptimizer(optimizer, ...)
preconditioner = KFAC(model, ...)

...

for i, (data, target) in enumerate(train_loader):
  optimizer.zero_grad()
  output = model(data)
  loss = criterion(output, target)
  loss.backward()

  optimizer.synchronize()
  preconditioner.step()
  with optimizer.skip_synchronize():
    optimizer.step()

  ...
```

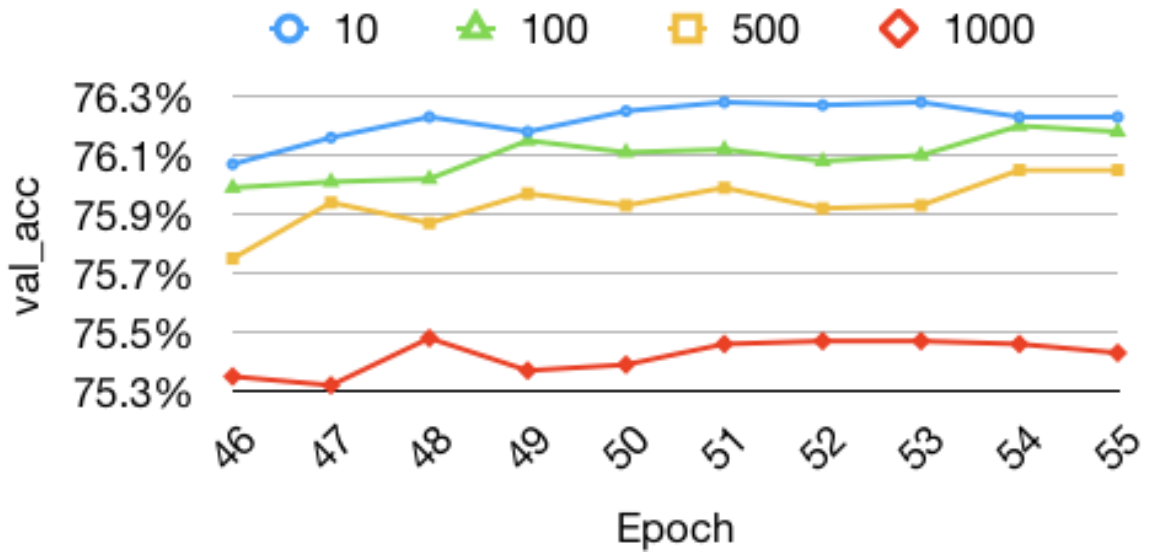
https://github.com/gpauloski/kfac_pytorch

Correctness: Cifar10 + ResNet-32



Validation Accuracy				
GPUs	**1**	**2**	**4**	**8**
SGD	92.76%	92.77%	92.58%	92.69%
K-FAC	92.93%	92.76%	92.90%	92.92%

Performance: K-FAC Update Interval



Model		SGD	K-FAC Update Freq. 100	500	1000
ResNet-50	Val Accu	76.2%	76.2%	76.1%	75.5%
	Train. Time (min)	178	152	128	124
ResNet-101	Val Accu	78.0%	77.7%	77.7%	77.3%
	Train. Time (min)	244	227	197	195
ResNet-152	Val Accu	78.2%	78.0%	78.0%	77.8%
	Train. Time (min)	345	369	310	300

Performance: Scaling

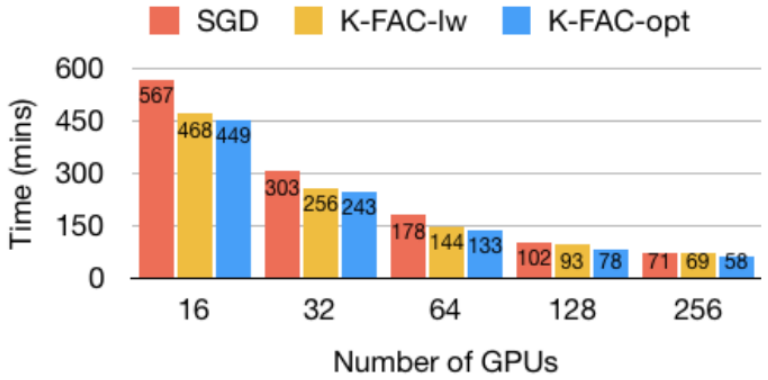


Fig. 7: Time-to-solution comparison of ResNet-50 using K-FAC with SGD.

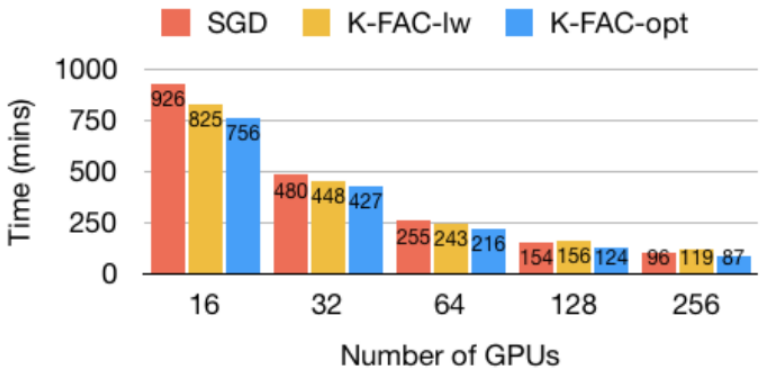


Fig. 8: Time-to-solution comparison of ResNet-101 using K-FAC and SGD.

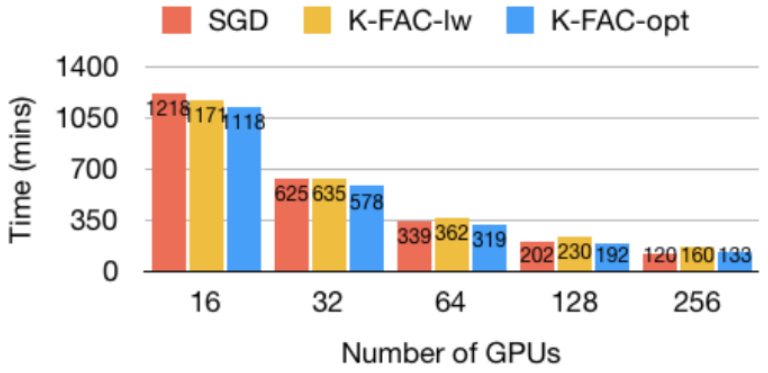


Fig. 9: Time-to-solution comparison of ResNet-152 using K-FAC and SGD.

TABLE IV: Summary of *K-FAC-opt* improvement over SGD

Scale	16	32	64	128	256
ResNet-50	20.9%	19.7%	25.2%	23.5%	17.7%
ResNet-101	18.4%	11.1%	15.1%	19.5%	9.7%
ResNet-152	8.2%	7.6%	6.0%	4.9%	-11.1%

- *KFAC-lw*: layer-wise distribution scheme by Osawa et al. (2019)
- *KFAC-opt*: this work

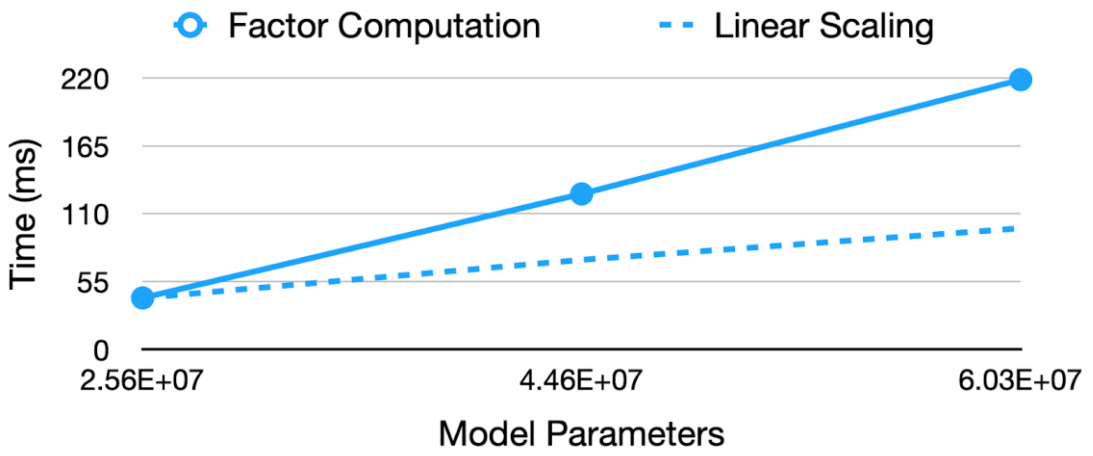
Limitations

Performance degrades as model complexity and worker counts increase.

- E.g. super-linear increase in factor computation time as model parameters increase.

Large imbalance in factor sizes

- Speedup limited by slowest worker
- ResNet-50: largest factor ~5000x5000, smallest factor 128x128
- ResNet-50, 16->64 GPUS:
  - Faster worker: 6.61x speedup
  - Slowest worker: 1.55x speedup



Relative Factor Computation Speedup

	ResNet-50		ResNet-101		ResNet-152	
GPUs	min	max	min	max	min	max
16	1.00	1.00	1.00	1.00	1.00	1.00
32	1.34	2.88	1.41	3.33	1.51	2.03
64	1.55	6.61	1.26	6.18	1.85	8.27

Conclusions

- We introduce an open source, distributed K-FAC preconditioner that is correct, efficient, and scalable.
- Converges to the 75.9% MLPerf ResNet-50 ImageNet baseline 18-25% faster than SGD.

Questions?

Feel free to contact me: jgpauloski@uchicago.edu