Optimization for Deep Learning

Jack Weitze

March 15, 2021

Optimization Background[1]

[1] Some material from:
https://csc2541-f17.github.io/slides/lec10.pdf

Optimization Background: Gradient Descent (Supervised Setting

Goal: minimize average loss $L(y, f(x, \theta))$ over N training examples,

- Objective $h(\theta) = \frac{1}{N} \sum L(y, f(x, \theta))$
- Neural network f with parameters θ generates predictions.

Optimization Background: Gradient Descent (Supervised Setting

Goal: minimize average loss $L(y, f(x, \theta))$ over N training examples,

- Objective $h(\theta) = \frac{1}{N} \sum L(y, f(x, \theta))$
- Neural network f with parameters θ generates predictions.

For differentiable loss, use gradient descent to update parameters.

- $\theta_{k+1} = \theta_k - \alpha \nabla_\theta h$
- $-\nabla_\theta h$ adjusts parameters in direction of *steepest descent*.
- First order method: locally linear approximation of objective.

Optimization Background: SGD

Compute average loss over a subset of the data,

- Using a smaller batch size results in a noisier update.
- Less progress per optimization step.
- Each step less expensive to compute.

SGD makes more *per-second* progress minimizing the objective than GD.

Optimization Background: SGD Variants

Popular modifications of the first-order SGD update.

- Approximate second order methods
- E.g. Adam, Adagrad, RMSProp
- Little overhead...
- but usually not much better than SGD

Optimization Background: Second Order Methods

Perform steepest-descent under a locally quadratic approximation of the objective.

- Newton's Method: uses Hessian $H = \nabla_\theta^2 h$

$$\theta_{k+1} = \theta_k - \alpha H^{-1}(\theta) \nabla_\theta h$$

- Generalized Gauss-Newton's Method:

$$\theta_{k+1} = \theta_k - \alpha G^{-1}(\theta) \nabla_\theta h$$

Where the GGN matrix $G = \mathbb{E}[\frac{\partial f}{\partial \theta} H \frac{\partial f}{\partial \theta}^\top]$

Optimization Background: First vs. Second Order Methods

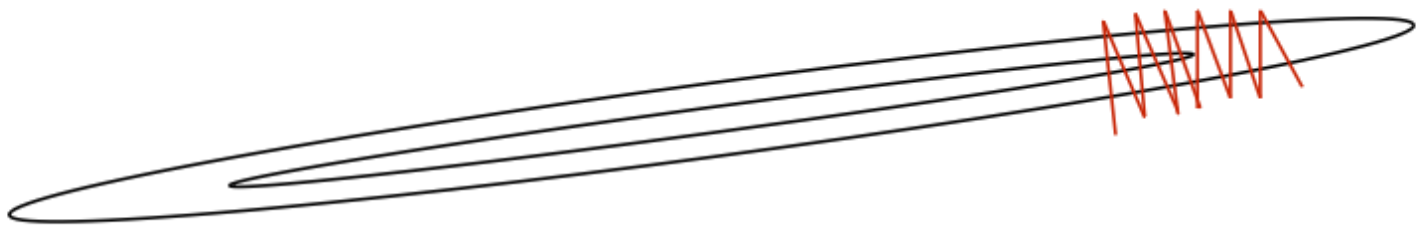
First order optimizers perform badly when curvature is badly conditioned.

[2] Grosse 2017

Optimization Background: First vs. Second Order Methods

First order optimizers perform badly when curvature is badly conditioned.

- Bounce around in high curvature directions.
- Slow progress in low curvature directions.
- Intuition: linear approximations only useful very locally (i.e. for a small step size).



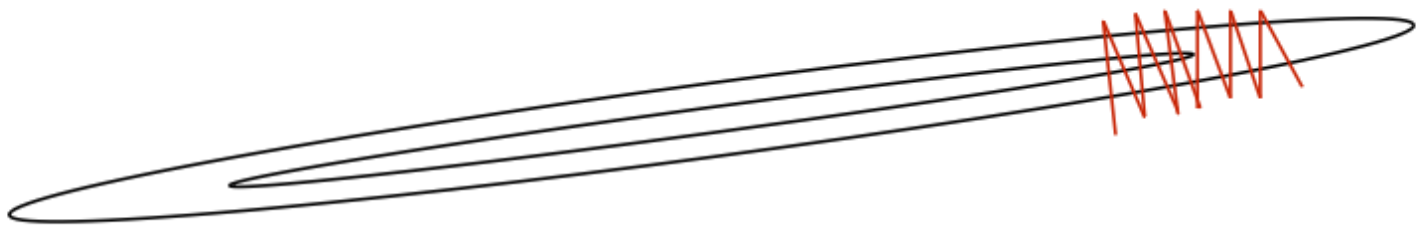
2

[2] Grosse 2017

Optimization Background: First vs. Second Order Methods

First order optimizers perform badly when curvature is badly conditioned.

- Bounce around in high curvature directions.
- Slow progress in low curvature directions.
- Intuition: linear approximations only useful very locally (i.e. for a small step size).



[2]

Second order method is curvature-aware, faster progress.

[2] Grosse 2017

Optimization Background: First vs. Second Order Methods

Second order methods give much more per-step progress.

But are expensive...

- Computing and inverting the Hessian.

Approximations to curvature trade-off efficiency and usefulness.

Questions?

Next: natural gradient descent.

Optimization Background: Steepest Descent

For each optimization step, the direction of *steepest descent* refers to the largest change in objective per unit change of parameters.

- Gradient descent rate$=\frac{\text{change on objective}}{\text{change in parameters}}$
- Change in parameters: measured in Euclidean distance.

Informally:

$$\frac{-\nabla_\theta h}{||\nabla_\theta h||} = \underset{\Delta\theta:||\Delta\theta||\leq 1}{\text{argmin}} \; h(\theta + \Delta\theta)$$

Natural Gradient Descent: KL Divergence, Fisher Matrix

Consider a network representing a predictive conditional distribution $p_\theta(y|x)$.

- Updates $\theta \to \theta' = \theta + \Delta\theta$ result in a new distribution $p_{\theta'}$.

Natural Gradient Descent: KL Divergence, Fisher Matrix

Consider a network representing a predictive conditional distribution $p_\theta(y|x)$.

- Updates $\theta \to \theta' = \theta + \Delta\theta$ result in a new distribution $p_{\theta'}$.
- Measure 'distance' in terms of the average KL divergence between $p_\theta(y|x)$ and $p_{\theta'}(y|x)$.

Natural Gradient Descent: KL Divergence, Fisher Matrix

Consider a network representing a predictive conditional distribution $p_\theta(y|x)$.

- Updates $\theta \to \theta' = \theta + \Delta\theta$ result in a new distribution $p_{\theta'}$.
- Measure 'distance' in terms of the average KL divergence between $p_\theta(y|x)$ and $p_{\theta'}(y|x)$.

The Fisher Information matrix is the second-order Taylor approximation to this average:

$$F_\theta = \mathbb{E}[\nabla^2_{\theta'} D_{KL}(p_{\theta'}||p_\theta)|_{\theta'=\theta}]$$

Natural Gradient Descent: Steepest Descent

Use Fisher matrix as distance constraint.

- Descent rate $= \frac{\text{change in objective}}{\text{change in parameters}}$
- Change in parameters: KL distance between $p_\theta(y|x)$ and $p_{\theta+\Delta\theta}(y|x)$.
- I.e. change in distribution of predictions.

Natural Gradient Descent: Steepest Descent

Use Fisher matrix as distance constraint.

- Descent rate $= \frac{\text{change in objective}}{\text{change in parameters}}$
- Change in parameters: KL distance between $p_\theta(y|x)$ and $p_{\theta+\Delta\theta}(y|x)$.
- I.e. change in distribution of predictions.

Choose steepest descent direction as:

$$\underset{\Delta\theta : F_\theta = const}{\text{argmin}} \quad h(\theta + \Delta\theta)$$

Natural Gradient Descent: Fisher Matrix

An equivalent formulation of the Fisher matrix is as the covariance of log-likelihood derivatives:

$$F_\theta = \mathbb{E}[\nabla log\ p_\theta(y|x) \nabla log\ p_\theta(y|x)^\top]$$

Expectation over training input (x's) and learned conditional distribution ($p_\theta(y|x)$).

Natural Gradient Descent: Fisher Matrix

An equivalent formulation of the Fisher matrix is as the covariance of log-likelihood derivatives:

$$F_\theta = \mathbb{E}[\nabla log\ p_\theta(y|x) \nabla log\ p_\theta(y|x)^\top]$$

Expectation over training input (x's) and learned conditional distribution ($p_\theta(y|x)$).

- Use Fisher for natural gradient descent:

$$\theta_{k+1} = \theta_k - \alpha F^{-1} \nabla_\theta h$$

Natural Gradient Descent: Fisher Matrix

An equivalent formulation of the Fisher matrix is as the covariance of log-likelihood derivatives:

$$F_\theta = \mathbb{E}[\nabla log\ p_\theta(y|x) \nabla log\ p_\theta(y|x)^\top]$$

Expectation over training input (x's) and learned conditional distribution ($p_\theta(y|x)$).

- Use Fisher for natural gradient descent:

$$\theta_{k+1} = \theta_k - \alpha F^{-1} \nabla_\theta h$$

No second derivatives!

Natural Gradient Descent: Fisher Matrix

Fisher can also be interpreted in terms of Hessian of $log\ p(x, y|\theta)$.

$$F_\theta = -\mathbb{E}[\nabla_\theta^2 log\ p_\theta(x, y)]$$

Thus we can view natural gradient descent as a second-order method.

Natural Gradient Descent: Fisher Matrix

Fisher can also be interpreted in terms of Hessian of $log\ p(x, y|\theta)$.

$$F_\theta = -\mathbb{E}[\nabla^2_\theta log\ p_\theta(x, y)]$$

Thus we can view natural gradient descent as a second-order method.

So long as the loss is a negative log-likelihood.

- E.g. softmax cross entropy, squared error loss.

Natural Gradient Descent: Intuition

Natural gradient descent makes more per-step progress than traditional.

- Can 'jump' over plateaus of likelihood $p_\theta(y|x)$.
- Plateaus of p_θ *usually* match plateaus of the objective.

[2] "Revisting natural gradient for deep networks" Pascanu, Bengio 2014

Natural Gradient Descent: Intuition

Natural gradient descent makes more per-step progress than traditional.

- Can 'jump' over plateaus of likelihood $p_\theta(y|x)$.
- Plateaus of p_θ *usually* match plateaus of the objective.
- 'Jump' over plateaus of objective.

[2] "Revisting natural gradient for deep networks" Pascanu, Bengio 2014

Natural Gradient Descent: Intuition

Natural gradient descent makes more per-step progress than traditional.

- Can 'jump' over plateaus of likelihood $p_\theta(y|x)$.
- Plateaus of p_θ *usually* match plateaus of the objective.
- 'Jump' over plateaus of objective.

KL constraint bounds the amount of per-step progress

- (above and below)

[2] "Revisting natural gradient for deep networks" Pascanu, Bengio 2014

"Optimizing Neural Networks with Kronecker-factored Approximate Curvature"

James martens, Roger Grsosse

JMLR, 2015

Practically Using Natural Gradient Descent in Neural Networks

- Same problems as Hessian.
- Need to efficiently store/compute/invert the Fisher.
- Seek an approximate a Fisher which still captures local curvature.

Practically Using Natural Gradient Descent in Neural Networks

- Same problems as Hessian.
- Need to efficiently store/compute/invert the Fisher.
- Seek an approximate a Fisher which still captures local curvature.

K-FAC approach: use a Kronecker-Factored approximate Fisher.

- Slower per iteration than SGD.
- Better per-second progress (sometimes).

K-FAC: Notation

- $\mathcal{D}Z$ Gradient of $log\ p_\theta(y|x)$ w.r.t Z.
- Neural network parameters:
- $\mathcal{W}$: weight matrix for layer.
- $\mathcal{A}$: input activation vector (output of the previous layer).
- $s = \mathcal{W}\mathcal{A}$: pre-activation inputs.

Fisher Matrix for a particular layer:

$$\begin{aligned} F &= \mathbb{E}[\nabla log\ p_\theta(y|x) \nabla log\ p_\theta(y|x)^\top] \\ &= \mathbb{E}[vec\{\mathcal{D}\mathcal{W}\} vec\{\mathcal{D}\mathcal{W}\}^\top] \\ &= \mathbb{E}[vec\{\mathcal{D}s\mathcal{A}^\top\} vec\{\mathcal{D}s\mathcal{A}^\top\}^\top] \end{aligned}$$

K-FAC: Approximation

Use Kronecker product "vec trick":

$$
\begin{aligned}
F &= \mathbb{E}[vec\{\mathcal{D}sA^\top\}vec\{\mathcal{D}sA^\top\}^\top] \\
&= \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top]
\end{aligned}
$$

K-FAC: Approximation

Use Kronecker product "vec trick":

$$\begin{aligned} F &= \mathbb{E}[vec\{\mathcal{D}sA^\top\}vec\{\mathcal{D}sA^\top\}^\top] \\ &= \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top] \end{aligned}$$

Assume second-order statistics are independent:

$$F \approx \mathbb{E}[\mathcal{A}\mathcal{A}^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

K-FAC: Approximation

Use Kronecker product "vec trick":

$$\begin{aligned} F &= \mathbb{E}[vec\{\mathcal{D}sA^\top\}vec\{\mathcal{D}sA^\top\}^\top] \\ &= \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top] \end{aligned}$$

Assume second-order statistics are independent:

$$F \approx \mathbb{E}[\mathcal{A}\mathcal{A}^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

Finally, assume the Fisher is block-diagonal, i.e. layers are independent.

Just need to invert blocks of Fisher.

K-FAC: Approximation

Defined the per-layer approximation $\hat{F}$:

$$\hat{F} \triangleq \mathbb{E}[\mathcal{A}\mathcal{A}^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

K-FAC: Approximation

Defined the per-layer approximation $\hat{F}$:

$$\hat{F} \triangleq \mathbb{E}[\mathcal{A}\mathcal{A}^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

Using inverse of Kronecker Product $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$

$$\hat{F}^{-1} = \mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1} \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$$

K-FAC: Approximation

Another property of Kronecker Product:

$$(A \otimes B)vec(C) = vec(BCA^\top)$$

(Approximate) Natural gradient descent update:

$$\tilde{\nabla} h \approx \hat{F}^{-1} vec(\nabla_\theta W) = vec\left(\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}(\nabla_\theta W)\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}\right)$$

K-FAC: Approximation

Another property of Kronecker Product:

$$(A \otimes B)vec(C) = vec(BCA^\top)$$

(Approximate) Natural gradient descent update:

$$\tilde{\nabla} h \approx \hat{F}^{-1} vec(\nabla_\theta W) = vec\left(\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}(\nabla_\theta W)\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}\right)$$

Update is reasonable to compute:

- Involves operations on matrices roughly the same size as W.

K-FAC: Regularized Approximation

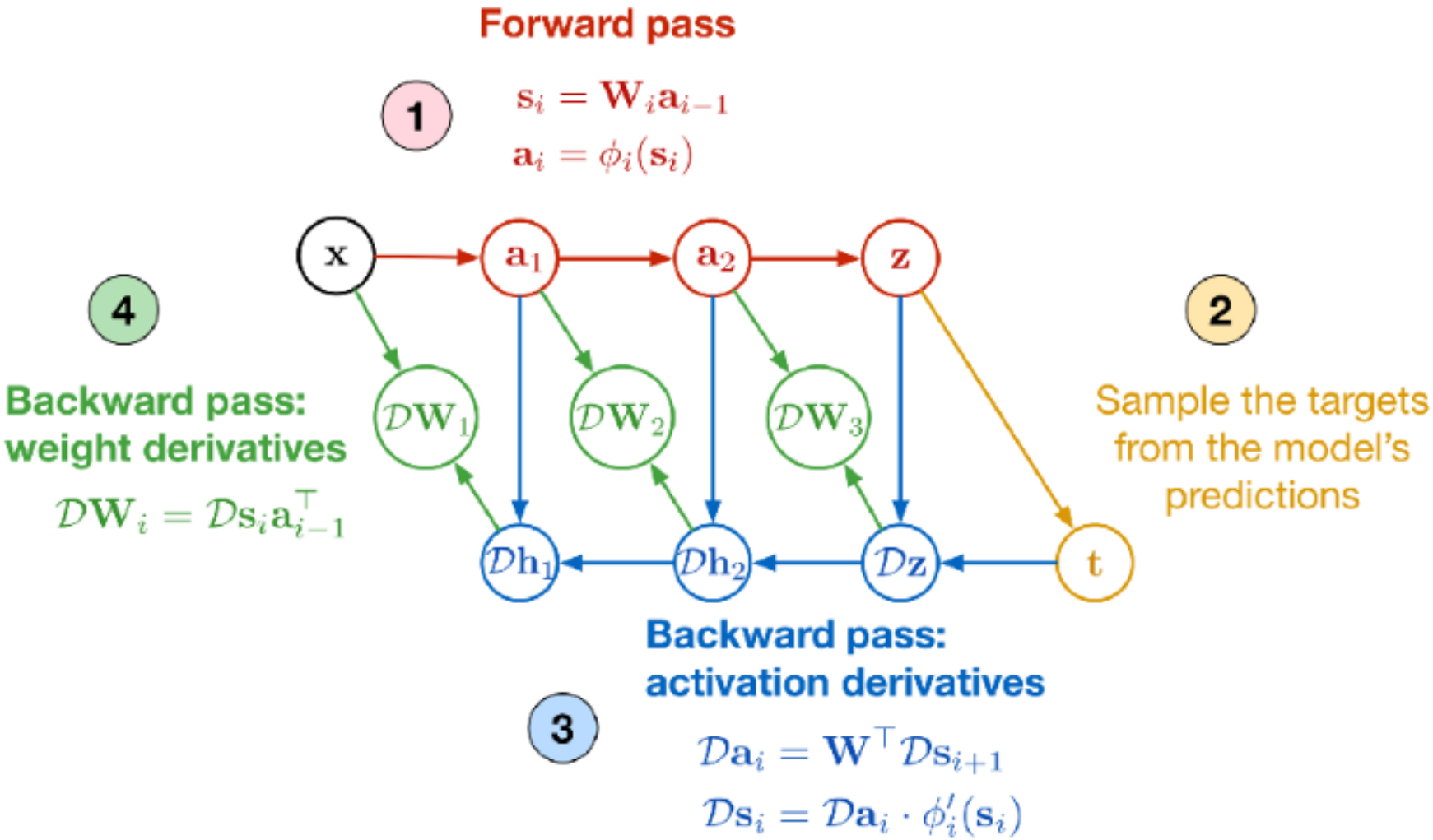
Similar to other second-order methods, K-FAC works better with update regularization.

- Regularized Fisher: $F + \lambda I$
- Approximate by "factored Tikhonov damping":

$$(\hat{F} + \lambda I)^{-1} \approx \left(\mathbb{E}[\mathcal{A}\mathcal{A}^\top] + \pi_{\mathcal{A}} \lambda^{\frac{1}{2}} I\right)^{-1} \otimes \left(\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top] + \pi_{\mathcal{D}s} \lambda^{\frac{1}{2}} I\right)^{-1}$$

Implementing K-FAC

Key point: values in K-FAC are obtained through usual forward/backward propagation.



Implementing K-FAC

Easy to estimate second-order statistics in normal training.

- $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]$ doesn't depend on y, compute $\mathcal{A}$ during forwards pass.

Implementing K-FAC

Easy to estimate second-order statistics in normal training.

- $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]$ doesn't depend on y, compute $\mathcal{A}$ during forwards pass.

- $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$ depends on y.

Approximate $\mathbb{E}_{p_\theta(y|x)}[\mathcal{D}s\mathcal{D}s^\top]$ using Monte-Carlo estimate:

1. Sampling $\hat{y}$ from network's predicted distribution $p_\theta(y|x)$.
2. Use $\hat{y}$ as training target, i.e. $L(\hat{y}, f(x, \theta))$.
3. Back propagate to compute $\mathcal{D}s$.

Implementing K-FAC

Easy to estimate second-order statistics in normal training.

- $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]$ doesn't depend on y, compute $\mathcal{A}$ during forwards pass.

- $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$ depends on y.

Approximate $\mathbb{E}_{p_\theta(y|x)}[\mathcal{D}s\mathcal{D}s^\top]$ using Monte-Carlo estimate:

1. Sampling $\hat{y}$ from network's predicted distribution $p_\theta(y|x)$.
2. Use $\hat{y}$ as training target, i.e. $L(\hat{y}, f(x, \theta))$.
3. Back propagate to compute $\mathcal{D}s$.

Empirical average of $\mathcal{A}$ and $\mathcal{D}s$ over batch.

Implementing K-FAC: Overhead vs SGD

Use forward/backward computations to compute ∇h for a given minibatch.

Implementing K-FAC: Overhead vs SGD

Use forward/backward computations to compute ∇h for a given minibatch.

Additional backprop to compute $\mathcal{D}s$ using sampled target.

Implementing K-FAC: Overhead vs SGD

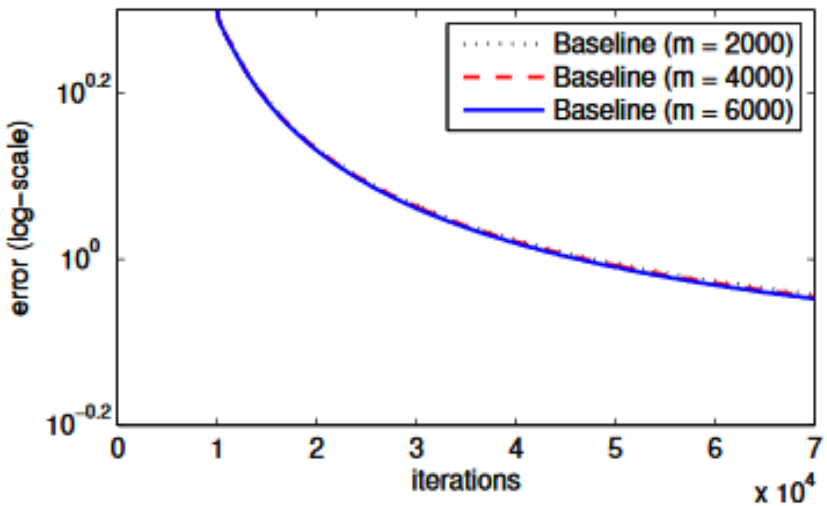
Use forward/backward computations to compute ∇h for a given minibatch.

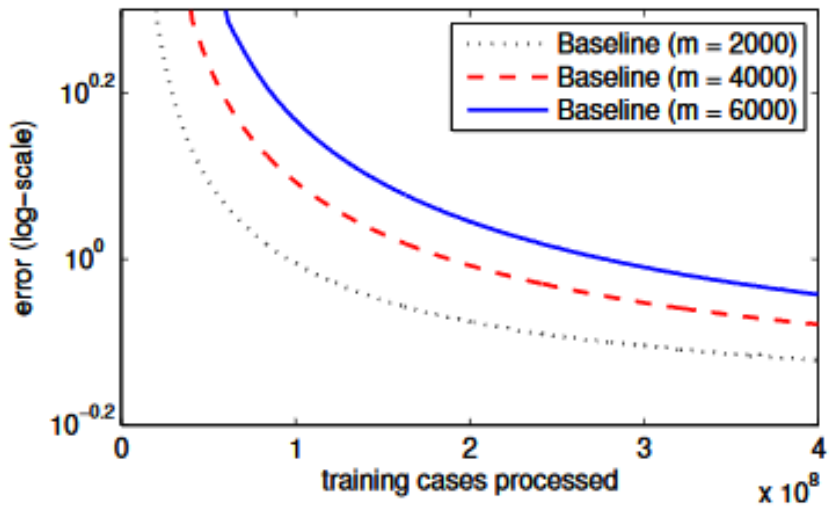
Additional backprop to compute $\mathcal{D}s$ using sampled target.

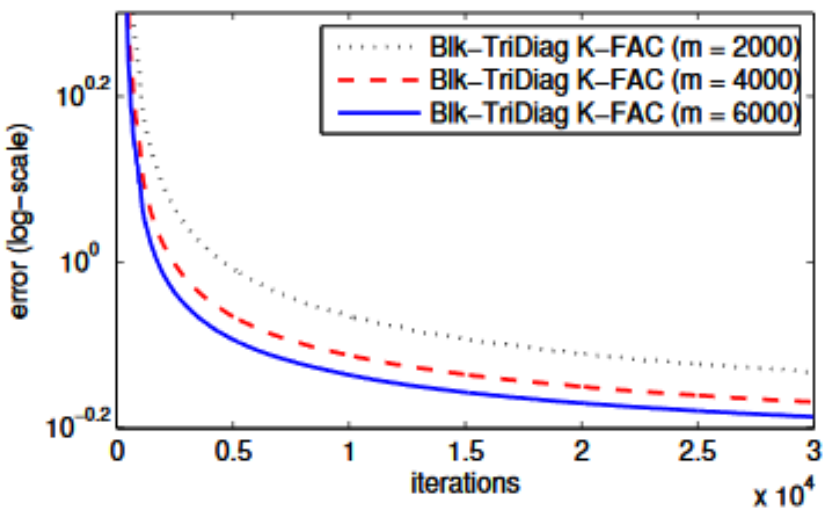
Invert $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$.

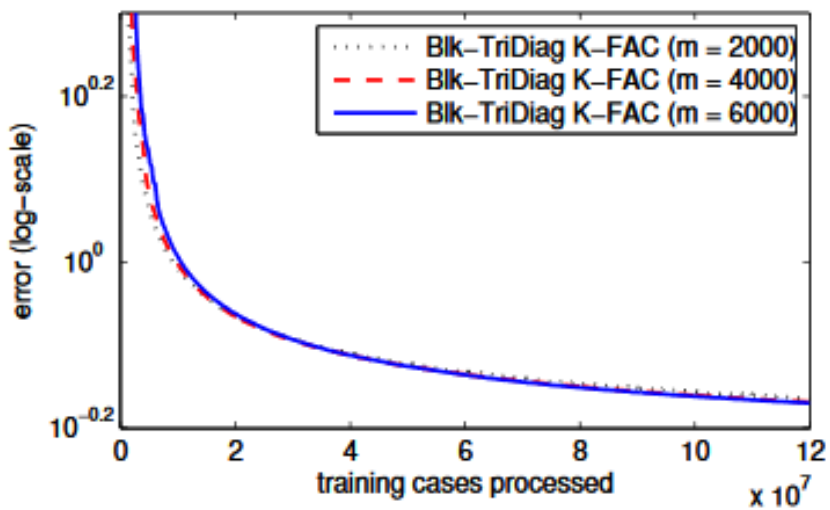
K-FAC Results

Baseline (SGD) vs. K-FAC at various batch-sizes m on MNIST autoencoding.









K-FAC vs. SGD

- Similar per-step cost, SGD is less expensive.
- K-FAC makes more progress per-step.
- K-FAC per-step progress *linear* or better in batch size.
- SGD per-step progress *sublinear* in batch size.

K-FAC thus more suitable than SGD to parallel distributed implementations.

K-FAC vs. SGD: Batch Size

Intuition: gradient descent steps are suboptimal due to

- Gradient noise: stochasticity due to mini-batches.
- Error surface approximations.

K-FAC vs. SGD: Batch Size

Intuition: gradient descent steps are suboptimal due to

- Gradient noise: stochasticity due to mini-batches.
- Error surface approximations.

Reduce noise:

- Gradient noise: increase batch size.
- Error surface: curvature aware approx.

K-FAC vs. SGD: Batch Size

Intuition: gradient descent steps are suboptimal due to

- Gradient noise: stochasticity due to mini-batches.
- Error surface approximations.

Reduce noise:

- Gradient noise: increase batch size.
- Error surface: curvature aware approx.

Gradient noise relevant near local minima, irrelevant in plateaus.

SGD spends time in plateaus, where big batches are wasted.

K-FAC jumps plateaus.

"Distributed Second-Order Optimization Using Kronecker-Factored Approximations"

Jimmy Ba, Roger Grosse, James Martens

ICLR, 2017

Accelerating NN Training

Can parallelize training by using bigger batches with many GPUs.

Accelerating NN Training

Can parallelize training by using bigger batches with many GPUs.

Want a learning method which scales well with bigger batch sizes.

Accelerating NN Training

Can parallelize training by using bigger batches with many GPUs.

Want a learning method which scales well with bigger batch sizes.

SGD has diminishing returns.

Distributed K-FAC: Method

Modifications to K-FAC to take advantage of parallel distributed computation.

Distributed K-FAC: Method

Modifications to K-FAC to take advantage of parallel distributed computation.

Results: faster training on big data sets by using large batches.

Distributed K-FAC: Method

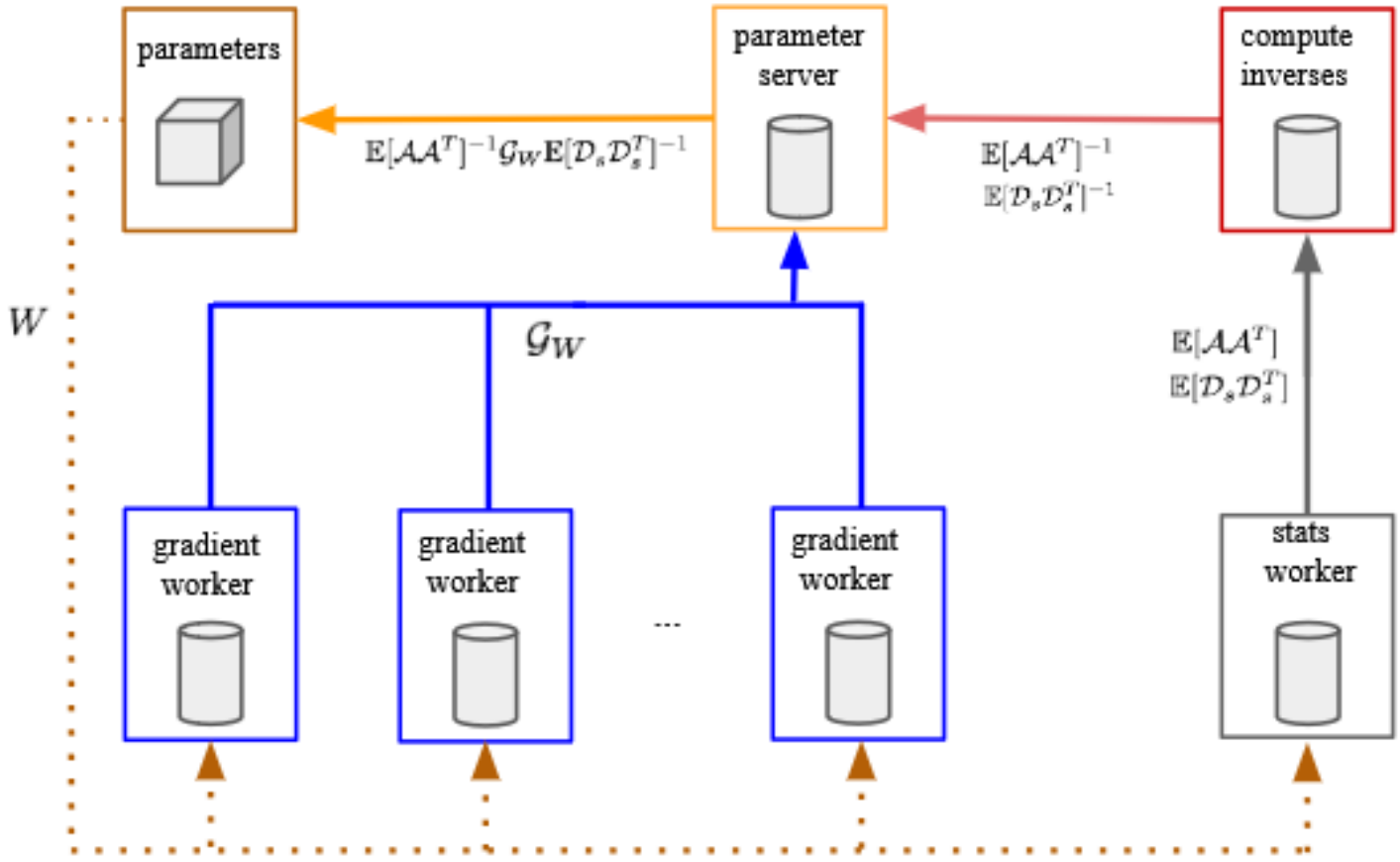
Modifications to K-FAC to take advantage of parallel distributed computation.

Results: faster training on big data sets by using large batches.

Main ideas:

- Asynchronous second order statistic estimation.
- Asynchronous Fisher block inversion.
- Doubly-Factored K-FAC for large layers.

Distributed K-FAC: Method



Distributed K-FAC: Asynchronous Statistics

Rather than reuse a mini-batch to compute statistics, use separate mini-batches.

Distributed K-FAC: Asynchronous Statistics

Rather than reuse a mini-batch to compute statistics, use separate mini-batches.

- Slightly more total computation...

Distributed K-FAC: Asynchronous Statistics

Rather than reuse a mini-batch to compute statistics, use separate mini-batches.

- Slightly more total computation...
- But enables greater degree of parallelism.
- Some statistics computed locally, e.g. for convolution layer approximations.

Distributed K-FAC: Asynchronous Statistics

Rather than reuse a mini-batch to compute statistics, use separate mini-batches.

- Slightly more total computation...
- But enables greater degree of parallelism.
- Some statistics computed locally, e.g. for convolution layer approximations.

Stats worker asynchronously computes $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$.

Distributed K-FAC: Asynchronous Fisher

K-FAC update requires inverted Kronecker factors $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$.

Distributed K-FAC: Asynchronous Fisher

K-FAC update requires inverted Kronecker factors $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$.

- Matrix inversion and eigendecomposition are GPU inefficient, compared to CPU.
- Curvature changes relatively slowly.

Distributed K-FAC: Asynchronous Fisher

K-FAC update requires inverted Kronecker factors $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$.

- Matrix inversion and eigendecomposition are GPU inefficient, compared to CPU.
- Curvature changes relatively slowly.

Compute inverted factors asynchronously on a CPU.

Distributed K-FAC: Asynchronous Fisher

K-FAC update requires inverted Kronecker factors $\mathbb{E}[\mathcal{A}\mathcal{A}^\top]^{-1}$, $\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}$.

- Matrix inversion and eigendecomposition are GPU inefficient, compared to CPU.
- Curvature changes relatively slowly.

Compute inverted factors asynchronously on a CPU.

Refresh Fisher only occasionally.

Result: 40-50% speed-up in wall-clock time compared to K-FAC.

Distributed K-FAC: Doubly Factored Layers

Layers with large inputs too expensive to invert, e.g. in large convolutional networks.

Double-factoring: further approximate inputs $\mathcal{A}$

Distributed K-FAC: Doubly Factored Layers

Layers with large inputs too expensive to invert, e.g. in large convolutional networks.

Double-factoring: further approximate inputs $\mathcal{A}$

- Recall: $F = \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top]$

Distributed K-FAC: Doubly Factored Layers

Layers with large inputs too expensive to invert, e.g. in large convolutional networks.

Double-factoring: further approximate inputs $\mathcal{A}$

- Recall: $F = \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top]$
- Rank one approx: $\mathcal{A} \approx \mathcal{K}\Psi^\top$

Distributed K-FAC: Doubly Factored Layers

Layers with large inputs too expensive to invert, e.g. in large convolutional networks.

Double-factoring: further approximate inputs $\mathcal{A}$

- Recall: $F = \mathbb{E}[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top]$
- Rank one approx: $\mathcal{A} \approx \mathcal{K}\Psi^\top$
- Use SVD of $\mathcal{A}$ for optimal rank-one approximation:
- $\mathcal{K} = \sqrt{\sigma_1}u_1$, $\Psi = \sqrt{\sigma_1}v_1$.

Distributed K-FAC: Doubly Factored Layers

Using rank-one approximation of $\mathcal{A}$:

$$\tilde{F} \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top \otimes \Psi\Psi^\top \otimes \mathcal{D}s\mathcal{D}s^\top]$$

Next assumption: $\mathcal{D}s$, Ψ and $\mathcal{K}$ independent.

$$\tilde{F} \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top] \otimes \mathbb{E}[\Psi\Psi^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

Distributed K-FAC: Doubly Factored Layers

Using rank-one approximation of $\mathcal{A}$:

$$\tilde{F} \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top \otimes \Psi\Psi^\top \otimes \mathcal{D}s\mathcal{D}s^\top]$$

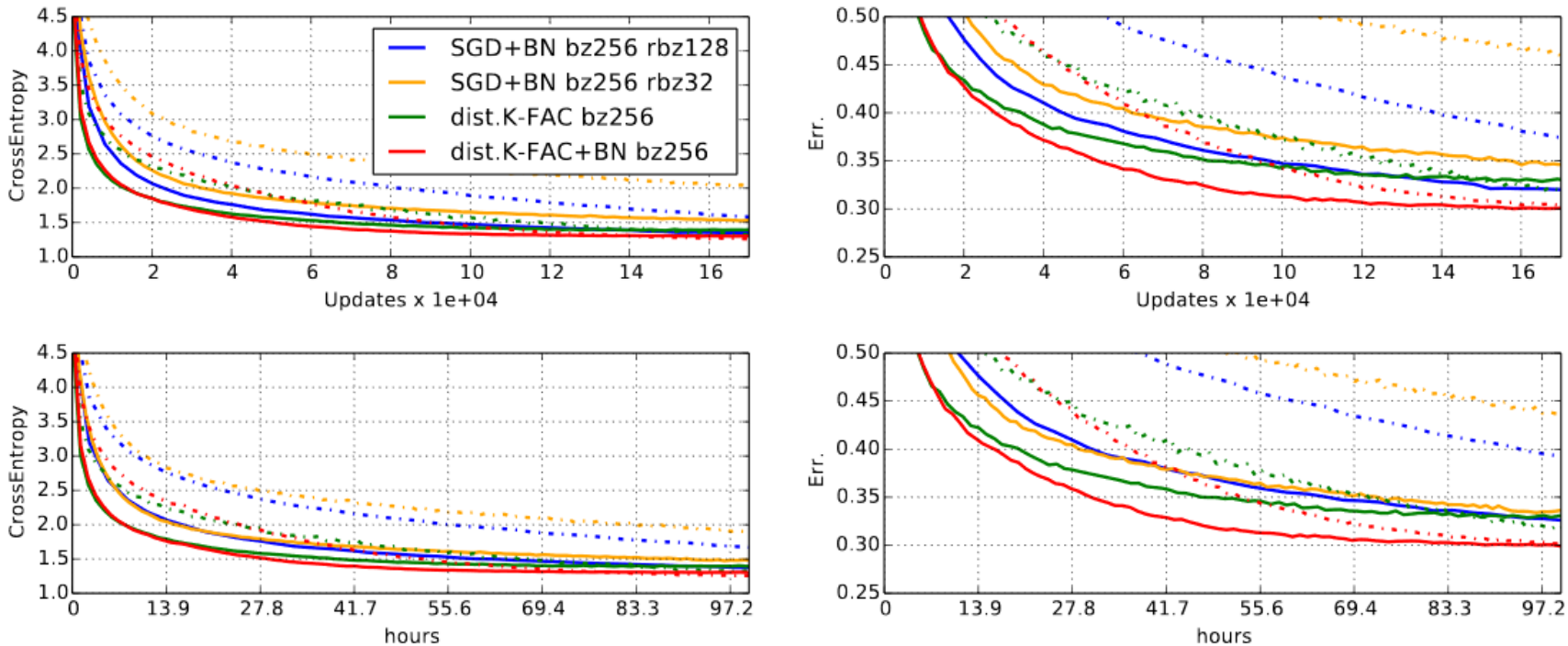
Next assumption: $\mathcal{D}s$, Ψ and $\mathcal{K}$ independent.

$$\tilde{F} \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top] \otimes \mathbb{E}[\Psi\Psi^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]$$

Results: can invert a 9216x4096 linear layer (first FC in AlexNet) in 15 seconds on a CPU.

Distributed K-FAC: Results

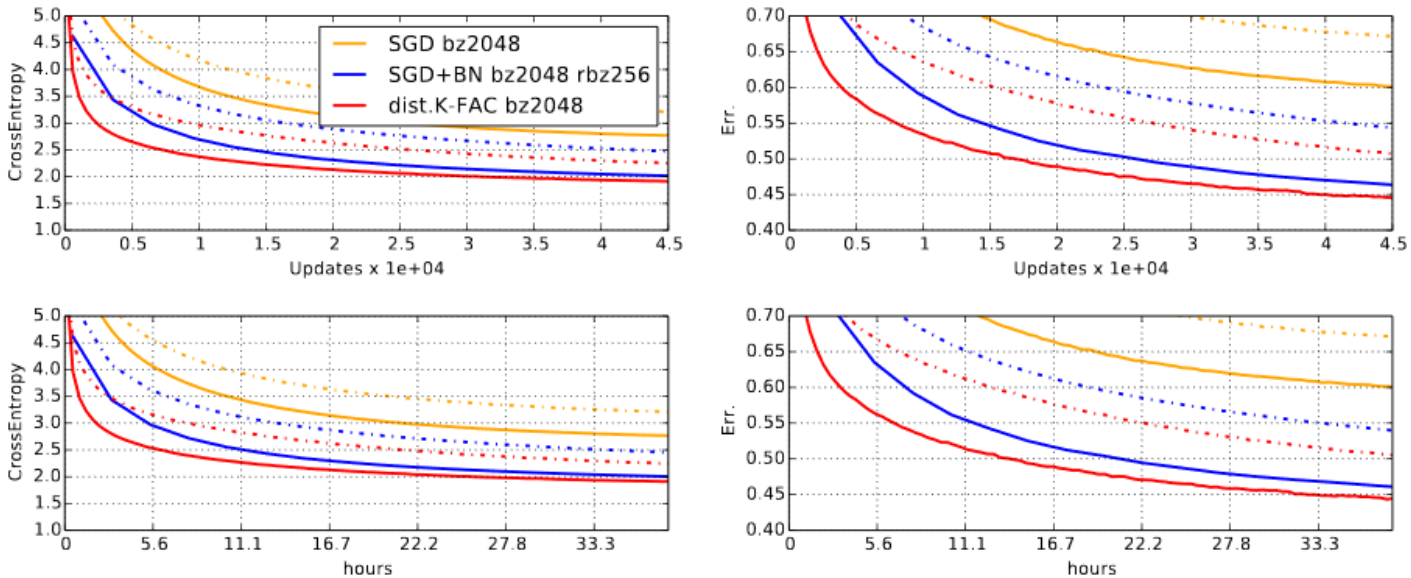
Training GoogLeNet on ImageNet: distributed K-FAC trains faster than SGD.



Solid lines are training loss, dashed lines are validation loss.

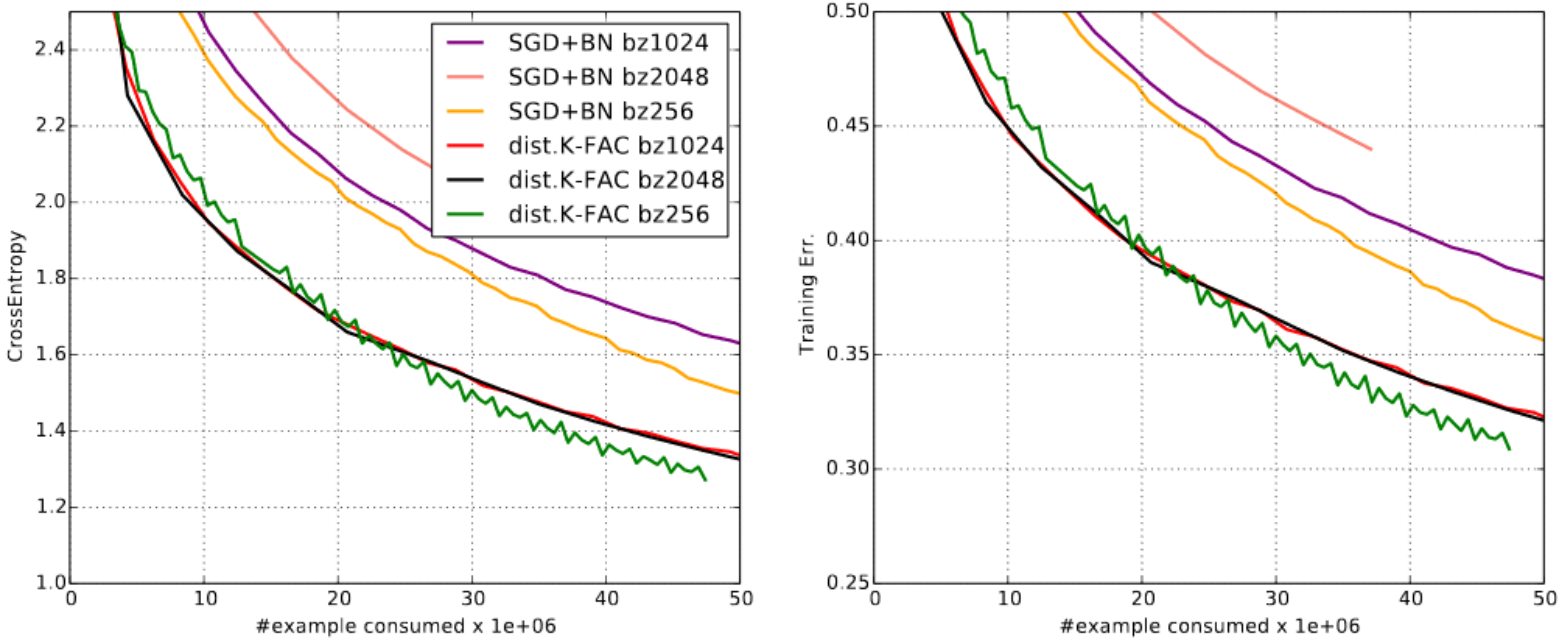
Distributed K-FAC: Results

Training AlexNet using doubly-factored approx: converges faster than SGD by wall-clock time.



Distributed K-FAC: Results

GoogLeNet varying batch size: K-FAC has constant per-example progress with increasing batch size.
SGD has *decreasing* per example progress, i.e. it is data inefficient.



Note x-axis is number of examples.

Distributed K-FAC: Conclusions

- Speedup training by increasing batch-size.

Many drawbacks:

- Not as useful if constrained to small batch sizes.

Distributed K-FAC: Conclusions

- Speedup training by increasing batch-size.

Many drawbacks:

- Not as useful if constrained to small batch sizes.
- Doesn't scale for large network layers.

Distributed K-FAC: Conclusions

- Speedup training by increasing batch-size.

Many drawbacks:

- Not as useful if constrained to small batch sizes.
- Doesn't scale for large network layers.
- Difficulty to implement; special approximations for different layers.

"On Empirical Comparisons of Optimizers for Deep Learning"

Daniel Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, George E. Dahl

Hyperparameter Search

Usual approach:

- Choose an optimizer.
- Choose a hyperparameter search space.
- Random or grid search, measure validation loss.
- Choose best configuration.

Hyperparameter Search: Comparing Optimizers

SGD depends only on learning rate λ_t:

$$\theta_{t+1} = \theta_t - \lambda_t \nabla h(\theta_t)$$

Hyperparameter Search: Comparing Optimizers

SGD depends only on learning rate λ_t:

$$\theta_{t+1} = \theta_t - \lambda_t \nabla h(\theta_t)$$

Momentum generalizes SGD with γ:

$$\begin{aligned} v_0 &= 0 \\ v_{t+1} &= \gamma v_t + \nabla h(\theta_t) \\ \theta_{t+1} &= \theta_t - \lambda_t v_{t+1} \end{aligned}$$

Hyperparameter Search: Comparing Optimizers

SGD depends only on learning rate λ_t:

$$\theta_{t+1} = \theta_t - \lambda_t \nabla h(\theta_t)$$

Momentum generalizes SGD with γ:

$$\begin{aligned} v_0 &= 0 \\ v_{t+1} &= \gamma v_t + \nabla h(\theta_t) \\ \theta_{t+1} &= \theta_t - \lambda_t v_{t+1} \end{aligned}$$

Nesterov adds history after gradient (changes meaning of γ):

$$\begin{aligned} v_0 &= 0 \\ v_{t+1} &= \gamma v_t + \nabla h(\theta_t) \\ \theta_{t+1} &= \theta_t - (\gamma \lambda_t v_{t+1} + \nabla h(\theta_t)) \end{aligned}$$

Hyperparameter Search: Comparing Optimizers

ADAM and RMSProp approximate second order methods.

ADAM parameters (λ_t, β_1, β_2, ϵ):

$$\theta_{t+1} = \theta_t - \lambda_t \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

Hyperparameter Search: Comparing Optimizers

ADAM and RMSProp approximate second order methods.

ADAM parameters (λ_t, β_1, β_2, ϵ):

$$\theta_{t+1} = \theta_t - \lambda_t \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

RMSProp parameters (λ_t, γ, ρ, ϵ):

$$m_{t+1} = \gamma m_t + \frac{\lambda_t}{\sqrt{v_{t+1} + \epsilon}} \nabla h(\theta_t)$$

$$\theta_{t+1} = \theta_t - m_{t+1}$$

Hyperparameter Search: Optimizer Tuning

Compare use “fair” search spaces.
Many ways to interpret this for optimizers.

Hyperparameter Search: Optimizer Tuning

Compare use "fair" search spaces.
Many ways to interpret this for optimizers.

"Similarly named parameters should take similar values"

- Momentum and Nesterov both have γ with entirely different meanings.

Hyperparameter Search: Optimizer Tuning

Compare use "fair" search spaces.
Many ways to interpret this for optimizers.

"Similarly named parameters should take similar values"

- Momentum and Nesterov both have γ with entirely different meanings.

"Tune only an important subset of parameters"

- Leave ϵ in RMSProp and ADAM at its default value.

Tuning ϵ

Why don't we tune ϵ?

Tuning ϵ

Why don't we tune ϵ?

- TF documentation: ϵ is "a small constant for numerical stability".
- If everyone leaves it fixed, comparisons might still be fair.
- Growing the search space is expensive.

Meaning of ϵ in ADAM

Consider ADAM as an empirical, diagonal approximation to natural gradient descent (NGD).

Meaning of ϵ in ADAM

Consider ADAM as an empirical, diagonal approximation to natural gradient descent (NGD).

- ϵ acts as a damping term, improves conditioning of Fisher.

Meaning of ϵ in ADAM

Consider ADAM as an empirical, diagonal approximation to natural gradient descent (NGD).

- ϵ acts as a damping term, improves conditioning of Fisher.
- ϵ interpolates momentum and diagonal NGD.

Meaning of ϵ in ADAM

Consider ADAM as an empirical, diagonal approximation to natural gradient descent (NGD).

- ϵ acts as a damping term, improves conditioning of Fisher.
- ϵ interpolates momentum and diagonal NGD.

Argument: best ϵ might be problem dependent, and should be tuned.

ADAM or SGD?

Conventional wisdom: ADAM might be easier to tune, but SGD tends to outperform.

ADAM or SGD?

Conventional wisdom: ADAM might be easier to tune, but SGD tends to outperform.

This wisdom made *without* tuning ϵ.

ADAM or SGD?

Conventional wisdom: ADAM might be easier to tune, but SGD tends to outperform.

This wisdom made *without* tuning ϵ.

Benchmarks don't reveal a single-best optimizer.

Optimizer A or Optimizer B?

Difficult to decide which optimizer to use.

Optimizer A or Optimizer B?

Difficult to decide which optimizer to use.

Easy to rely on conventional wisdom.

Optimizer A or Optimizer B?

Difficult to decide which optimizer to use.

Easy to rely on conventional wisdom.

What would be a better way to structurally choose an optimizer?

(Approximate) Hierarchy of Optimizers

If optimizer A generalizes optimizer B, then for an appropriate search space, A outperforms B

(Approximate) Hierarchy of Optimizers

If optimizer A generalizes optimizer B, then for an appropriate search space, A outperforms B

Momentum generalizes SGD so long as search space includes $\gamma = 0$

(Approximate) Hierarchy of Optimizers

If optimizer A generalizes optimizer B, then for an appropriate search space, A outperforms B

Momentum generalizes SGD so long as search space includes $\gamma = 0$

ADAM generalizes momentum so long as...

- ϵ can grow very large,
- and with a specialized learning rate schedule.

Hierarchy of Optimizers

Hierarchy only holds under specialized conditions on search space.

Hierarchy of Optimizers

Hierarchy only holds under specialized conditions on search space.

- Choose most general optimizer.

Hierarchy of Optimizers

Hierarchy only holds under specialized conditions on search space.

- Choose most general optimizer.

May not be practically relevant.

Hierarchy of Optimizers

Hierarchy only holds under specialized conditions on search space.

- Choose most general optimizer.

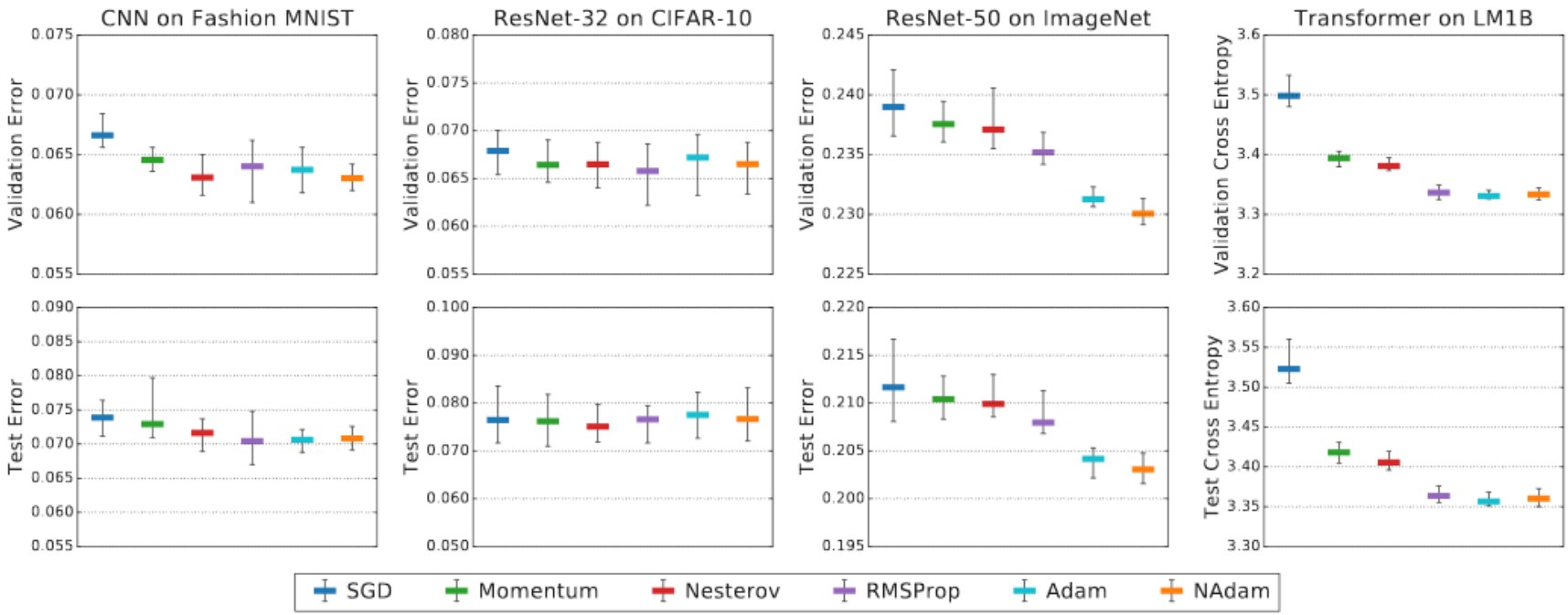
May not be practically relevant.

Unless, results using the special search space are competitive.

- Then hierarchy is worthwhile.

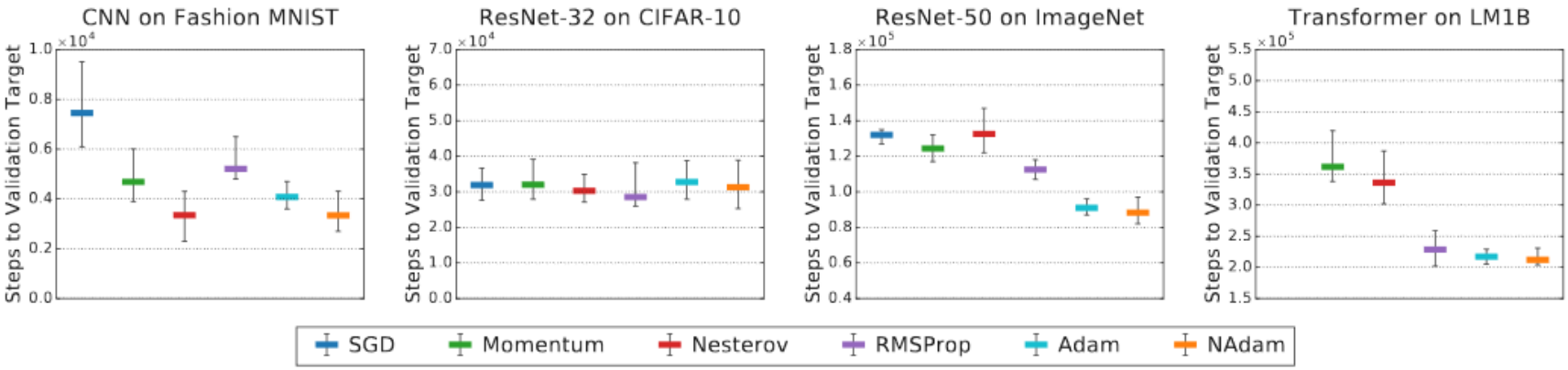
Experiments

Notice: ADAM error almost always lower than SGD.



Experiments

Notice: ADAM training time almost always lower than SGD.



Takeaways, Limitations of Study

Interesting challenge to conventional wisdom.

Takeaways, Limitations of Study

Interesting challenge to conventional wisdom.

Usefulness of hierarchy hinges on practical relevance.

Takeaways, Limitations of Study

Interesting challenge to conventional wisdom.

Usefulness of hierarchy hinges on practical relevance.

Only considered 16 experiments, might wait for more.

Takeaways, Limitations of Study

Interesting challenge to conventional wisdom.

Usefulness of hierarchy hinges on practical relevance.

Only considered 16 experiments, might wait for more.

Experiments done on toy examples, not current state-of-the-art.

Future Work

Found that tuning ϵ *does* matter.

Shown that framework for inclusion relationships has potential.

More experiments showing state of the art results would encourage use of their search method.

Questions?

References

- "Scalable natural gradient using probabilistic models of backpropr" Roger Grosse, 2017 (slides). https://csc2541-f17.github.io/slides/lec10.pdf
- "Optimizing Neural Networks with Kronecker-factored Approximate Curvature" Martens, Grosse, 2020. https://arxiv.org/pdf/1503.05671.pdf
- "Revisting natural gradient for deep networks" Pascanu, Bengio, 2014. https://arxiv.org/pdf/1301.3584.pdf
- "New Insights and Perspectives on the Natural Gradient Method" Martens, 2020. https://www.jmlr.org/papers/volume21/17-678/17-678.pdf
- "Distributed Second-Order Optimization Using Kronecker-Factored Approximates" Ba, Grosse, Martens, 2017. https://jimmylba.github.io/papers/nsync.pdf
- "On Empirical Comparisons of Optimizers for Deep learning" Choi, Shallue, Nado, Lee, Maddison, Dahl, 2020. https://arxiv.org/pdf/1910.05446.pdf