The K-FAC method for neural network optimization

James Martens

Thanks to my various collaborators on K-FAC research and engineering:

Roger Grosse, Jimmy Ba, Vikram Tankasali, Matthew Johnson, Daniel Duckworth, Zack Nado, and many more!

Introduction

- Neural networks are everywhere and the need to quickly train them has never been greater
- Main workhorse “diagonal” methods like RMSProp and Adam typically aren’t much faster than *well-tuned* SGD w/ momentum
- New non-diagonal methods like K-FAC and Natural Nets provide much more substantial performance improvements and make better use of larger mini-batch sizes
- In this talk I will introduce the basic K-FAC method, discuss extensions to RNNs and Convnets, and present empirical evidence for its efficacy

Talk outline

- Discussion of second order methods
- Discussion of generalized Gauss-Newton matrix and relationship to Fisher (drawing heavily from this paper)
- Intro to Kronecker-factored approximate curvature (K-FAC) approximation for fully-connected layers (+ results from paper)
- Extension of approximation to RNNs + results (paper)
- Extension of approximation to Convnets + (paper)
- Large batch experiments performed at Google and elsewhere

Notation, loss and objective function

- Neural network function: $f(x, \theta)$
- Loss: $-\log p(y|x, \theta) = -\log r(y|f(x, \theta)) = L(y, f(x, \theta))$
- Loss derivative: $\mathcal{D}V = \frac{\mathrm{d}L(y, f(x, \theta))}{\mathrm{d}V} = -\frac{\mathrm{d}\log p(y|x, \theta)}{\mathrm{d}V}$
- Objective function:

$$h(\theta) = \mathbb{E}_Q[L(y, f(x, \theta))]$$

2nd-order methods

Formulation

- Approximate $h(\theta)$ by its 2nd-order Taylor series around current θ :

$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d$$

- Minimize this local approximation to compute update:

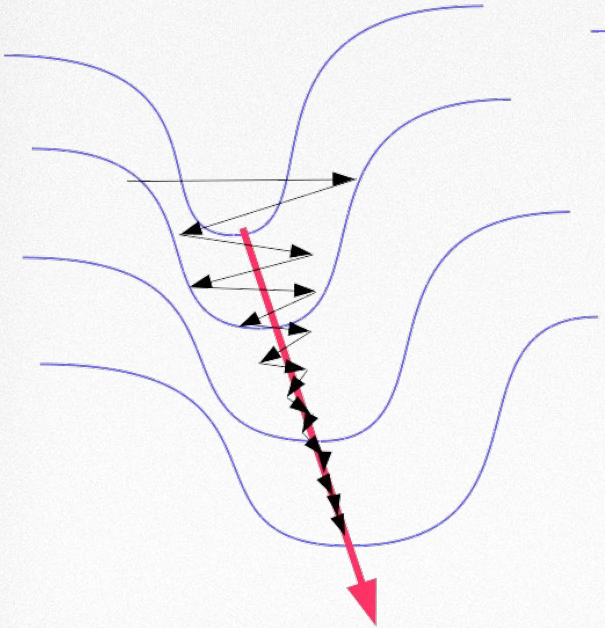
$$-H(\theta)^{-1} \nabla h(\theta) = \arg\min_d \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

- Update current iterate:

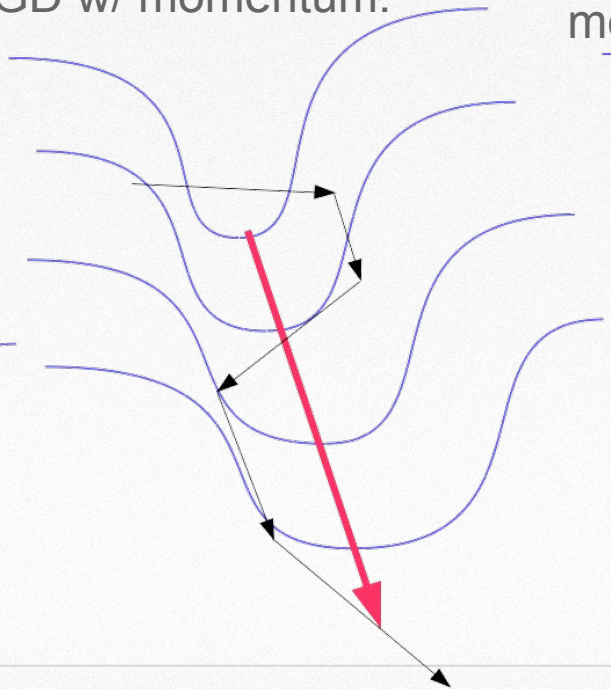
$$\theta_{k+1} = \theta_k - H(\theta)^{-1} \nabla h(\theta_k)$$

A cartoon comparison of different optimizers

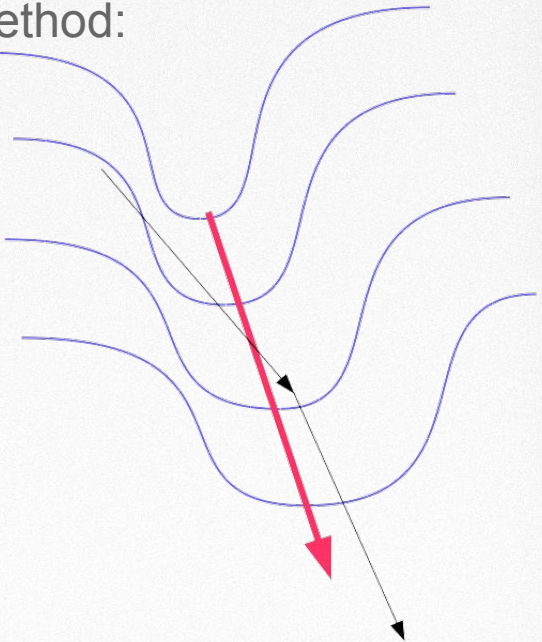
Gradient descent:



GD w/ momentum:



Ideal 2nd-order method:



The model trust problem in 2nd-order methods

- Quadratic approximation of loss is only trustworthy in a local region around current θ
- Unlike gradient descent, which implicitly approximates $LI \approx H(\theta)$ (where L upper-bounds the **global** curvature), the real $H(\theta)$ may underestimate curvature along some directions as we move away from current θ (and curvature may even be *negative!*)
- ***Solution:*** Constrain update d to lie in some local region R around 0 where approximation remains a good one

$$\arg\min_{d \in R} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

Trust-regions and “damping” (aka Tikhonov regularization)

- If we take $R = \{d : \|d\|_2 \leq r\}$ then computing

$$\arg\min_{d \in R} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

is often equivalent to computing

$$-(H(\theta) + \lambda I)^{-1} \nabla h(\theta) = \arg\min_{d} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top (H(\theta) + \lambda I) d \right)$$

for some λ.

- λ is a complicated function of r, but fortunately we can just work with λ directly. There are effective heuristics for adapting λ such as the “Levenberg-Marquardt” method.

Alternative curvature matrices

A complementary solution to the model trust problem

- In place of the Hessian we can use a matrix with more forgiving properties that tends to upper-bound the curvature over larger regions (without being too pessimistic!)
- Very important effective technique in practice if used alongside previously discussed trust-region / damping techniques
- Some important examples
    - Generalized Gauss-Newton matrix (GGN)
    - Fisher information matrix (often equivalent to the GGN)
    - Empirical Fisher information matrix (a type of approximation to the Fisher)

Generalized Gauss-Newton

Definition

- To define the GGN matrix we require that

  where $$h(\theta) = \frac{1}{m}\sum_{i=1}^{m} h_i(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(y_i, f(x_i, \theta))$$

  $\ell(y, z)$ is a loss that is convex in z, and

  $f(x, \theta)$ is some high-dimensional function (e.g. neural network w/ input x)

- The GGN is then given by

$$G = \frac{1}{m}\sum_{i=1}^{m} J_i^\top H_i J_i$$

where J_i is Jacobian of $f(x_i, \theta)$ w.r.t. θ

and H_i is the Hessian of $\ell(y_i, z_i)$

w.r.t. $z_i = f(x_i, \theta)$

Generalized Gauss-Newton

- G is equal to the Hessian of $h(\theta)$ if we replace each $f(x_i, \theta)$ with its local 1st-order approximation centered at current θ:

$$f(x_i, \theta') \approx f(\theta) + J_i \cdot (\theta' - \theta)$$

- When $\ell(y, z) = \|y - z\|^2/2$ we have $H_i = I$ and so

$$G = \frac{1}{m} \sum_{i=1}^{m} J_i^\top J_i$$

which is the matrix used in the well-known Gauss-Newton approach for optimizing nonlinear least squares

Relationship of GGN to the Fisher

- When $\ell(y, z) = -\log p(y|z)$ with z the "natural parameter" of some exponential family conditional density $p(y|z)$, G becomes **equivalent** to the Fisher information matrix:

$$F = \mathbb{E}[\mathcal{D}\theta \mathcal{D}\theta^\top] = \mathrm{cov}(\mathcal{D}\theta, \mathcal{D}\theta)$$

Recall notation:

$$\mathcal{D}V = \frac{\mathrm{d}L(y, f(x, \theta))}{\mathrm{d}V} = -\frac{\mathrm{d}\log p(y|x, \theta)}{\mathrm{d}V}$$

- In this case $G^{-1}\nabla h(\theta)$ is equal to the well-known "natural gradient", although has the additional interpretation as a second-order update
- This relationship justifies the common use of methods like damping/trust regions with natural gradient based optimizers

GGN Properties

The GGN matrix has the following nice properties:

- it always PSD
- it is often more “conservative” than the Hessian (but isn’t guaranteed to be larger in *all* directions)
- optimizer using update $d = -\alpha G^{-1} \nabla h(\theta)$ will be invariant to any smooth reparameterization in limit as $\alpha \to 0$
- for RELU networks the GGN is equal to the Hessian on diagonal blocks
- *and most importantly…* works much better than the Hessian in practice for neural networks!

Updates computed using the GGN can sometimes make *orders of magnitude* more progress than gradient updates for neural nets. But there is a catch...

The problem of high dimensional objectives

The main issue with 2nd-order methods

- For neural networks, $\theta \in \mathbb{R}^n$ can have 10s of millions of dimensions
- We simply cannot compute and store an $n \times n$ matrix for such an n, let alone invert it! $(\mathcal{O}(n^3))$
- Thus we must approximate the curvature matrix using one of a number of techniques that simplify its structure to allow for efficient...
  - computation,
  - storage,
  - and inversion

Curvature matrix approximations

- Well known curvature matrix approximations include:
  - diagonal (e.g. RMSprop, Adam)
  - block-diagonal (e.g. TONGA)
  - low-rank + diagonal (e.g. L-BFGS)
  - Krylov subspace (e.g. HF)
- The K-FAC approximation of the Fisher/GGN uses a more sophisticated approximation that exploits the special structure present of neural networks

The amazing Kronecker product

- The Kronecker product is defined by:

$$B \otimes C \equiv \begin{bmatrix} [B]_{1,1}C & \cdots & [B]_{1,n}C \\ \vdots & \ddots & \vdots \\ [B]_{m,1}C & \cdots & [B]_{m,n}C \end{bmatrix}$$

- And has many nice properties, such as:
  - $(B \otimes C)(E \otimes F) = BE \otimes CF$
  - $(B \otimes C)^\top = (B^\top \otimes C^\top)$
  - $(B \otimes C)^{-1} = B^{-1} \otimes C^{-1}$

Kronecker-factored approximation

- Consider a weight matrix W in network which computes the mapping:

$$s = Wa$$

(i.e. a “fully connected layer” or “linear layer”)

Here, and going forward F will refer just to the ***block*** of the Fisher corresponding to W

- Define $g = \mathcal{D}s$ and observe that $\mathcal{D}W = ga^\top$. If we approximate g and a as *statistically independent*, we can write F as:

$$\begin{aligned} F &= \operatorname{cov}(\operatorname{vec}(\mathcal{D}W), \operatorname{vec}(\mathcal{D}W)) = \mathbb{E}[\operatorname{vec}(ga^\top)\operatorname{vec}(ga^\top)^\top] = \mathbb{E}[(a \otimes g)(a \otimes g)^\top] \\ &= \mathbb{E}[(aa^\top) \otimes (gg^\top)] = \mathbb{E}[aa^\top] \otimes \mathbb{E}[gg^\top] = A \otimes G, \end{aligned}$$

Recall notation:

$$\mathcal{D}V = \frac{\mathrm{d}L(y, f(x,\theta))}{\mathrm{d}V} = -\frac{\mathrm{d}\log p(y|x,\theta)}{\mathrm{d}V}$$

Kronecker-factored approximation (cont.)

- Approximating $F = A \otimes G$ allows us to easily invert F and multiply the result by a vector, due to the following identities for Kronecker products:

$$(B \otimes C)^{-1} = B^{-1} \otimes C^{-1} \quad \text{and} \quad (B \otimes C)\,\mathrm{vec}(X) = \mathrm{vec}(CXB^\top)$$

- We can easily estimate the matrices

$$A = \mathbb{E}[aa^\top] \text{ and } G = \mathbb{E}[gg^\top] = \mathrm{cov}(g, g)$$

  using simple Monte-Carlo and exp-decayed moving averages.

- They are of size ***d*** by ***d*** where ***d*** is the number of units in the incoming or outgoing layer. Thus inverting them is relatively cheap, and can be amortized over many iterations.

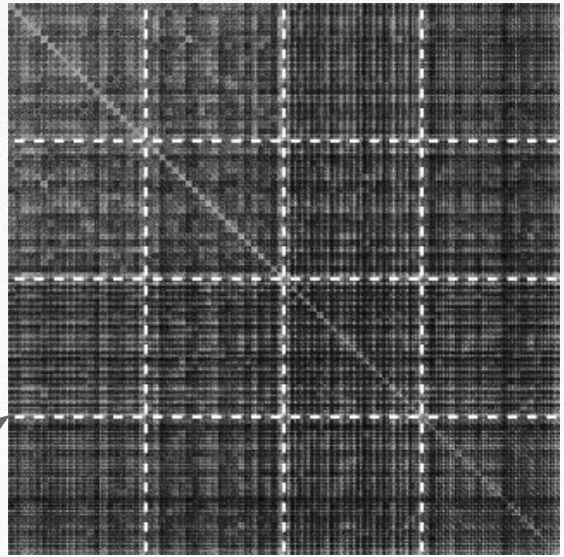
Further remarks about the K-FAC approximation

- Originally appeared in a 2000 paper by Tom Heskes!
- Can be seen as discarding order 3+ cumulants from the joint distribution of the a's and g's
  - (And thus is exact if the a's and g's are jointly Gaussian-distributed)
- For linear neural networks with a squared error loss:
  - is exact on the diagonal blocks
  - approximate natural gradient differs from exact one by a constant factor (Bernacchia et al., 2018)
- Can also be derived purely from the GGN perspective without invoking the Fisher (Botev et al., 2017)

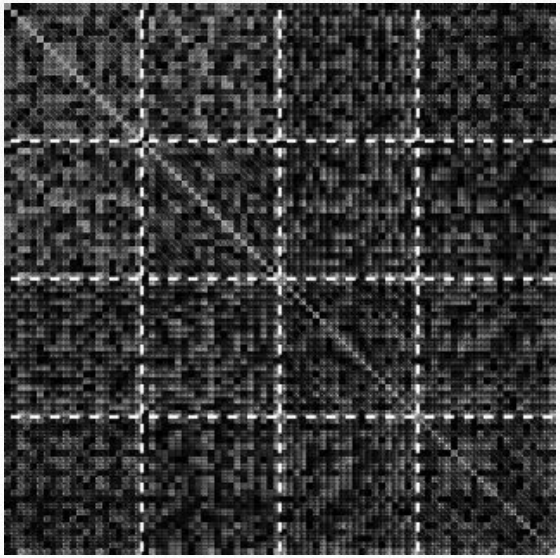
Visual inspection of approximation quality

4 middles layers of partially trained MNIST classifier

Exact



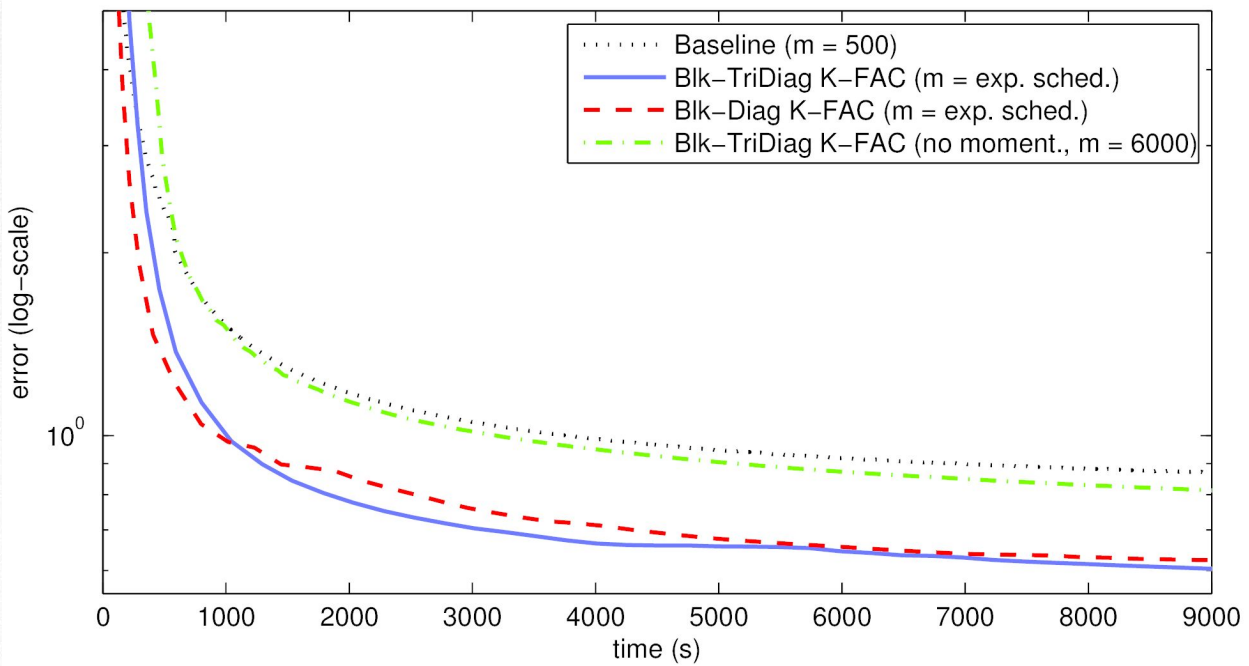
Approx



Dashed lines delineate the blocks

(plotting absolute value of entries, dark means small)

MNIST deep autoencoder - single GPU wall clock



Baseline = **highly optimized** SGD w/ momentum

Some stochastic convergence theory

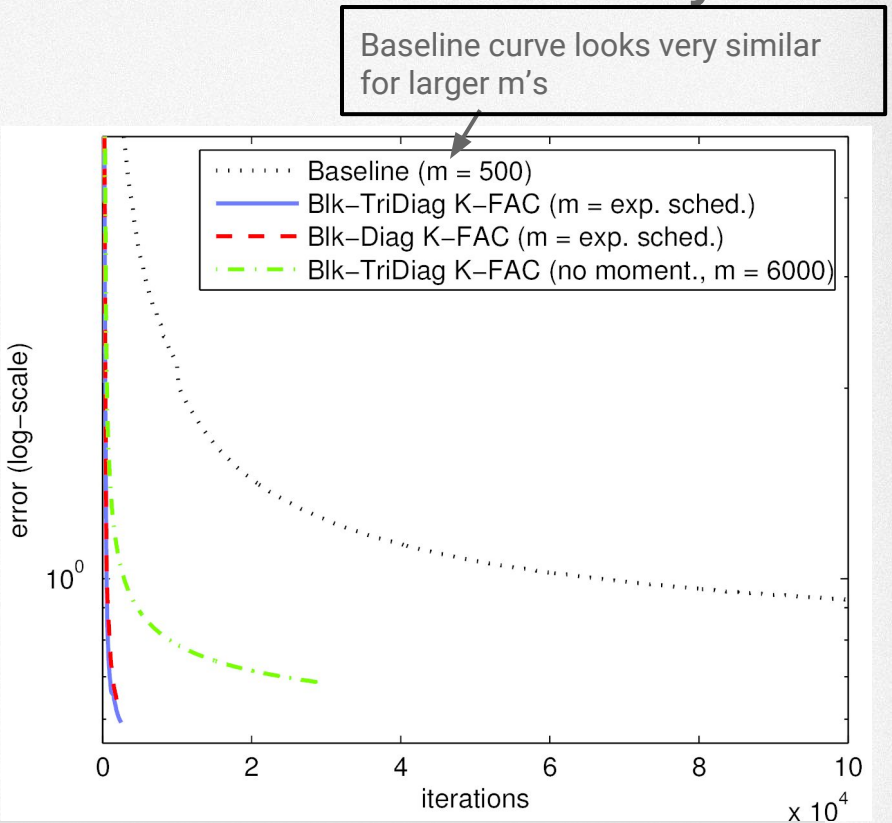
- There is no ***asymptotic*** advantage to using 2nd-order methods or momentum over plain SGD w/ *Polyak averaging*
- Actually, SGD w/ *Polyak averaging* is ***asymptotically optimal*** among any estimator that sees k training cases, obtaining the optimal asymptotic rate:

$$E[h(\theta_k)] - h(\theta^*) \in \mathcal{O}\left(\frac{1}{k}\,\mathrm{tr}\left(H(\theta^*)^{-1}\Sigma\right)\right)$$

  where θ^* is the optimum, and Σ is the (the limiting value of) the per-case gradient covariance
- *However*, ***pre-asymptotically*** there can still be an advantage to using 2nd-order updates and/or momentum. (Asymptotics kick in when signal-to-noise ratio in stochastic gradient becomes small.)

MNIST deep autoencoder - iteration efficiency

- K-FAC uses far fewer total iterations than a well-tuned baseline when given a **very large** mini-batch size
  - This makes it ideal for large distributed systems
- *Intuition:* the asymptotics of stochastic convergence kick in sooner with more powerful optimizers since “optimization” stops being the bottleneck sooner



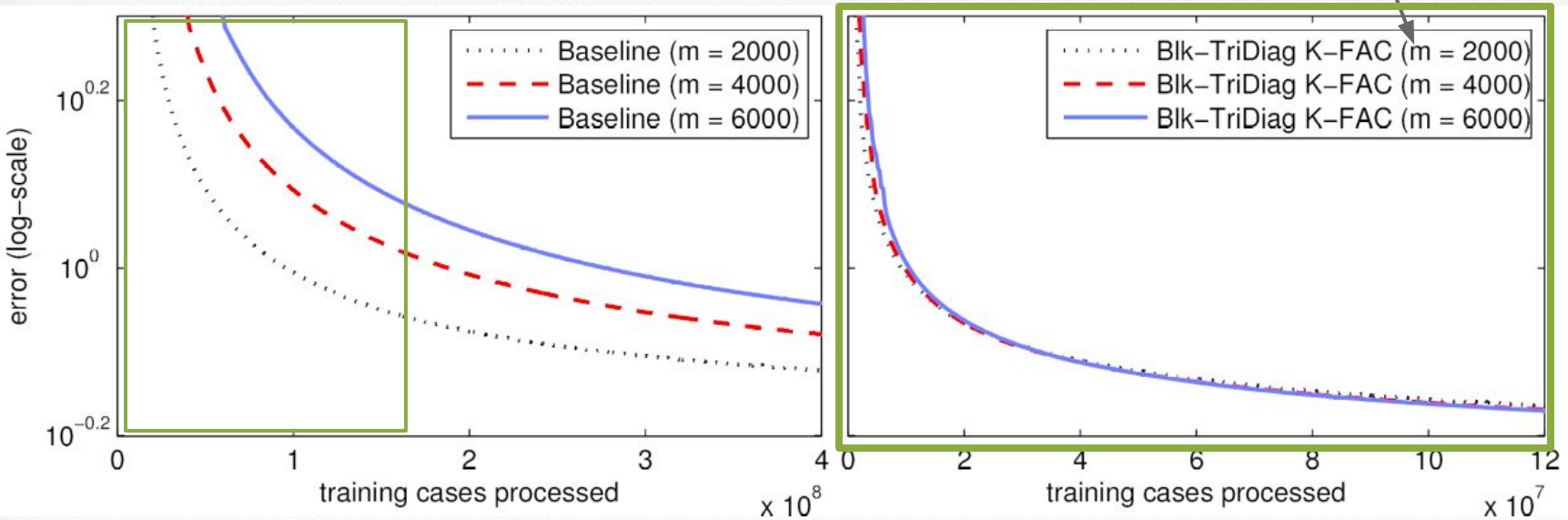
MNIST deep autoencoder - data efficiency

Baselines spends much longer in pre-asymptotic phase

Exact

Approx

m = mini-batch size



Baseline = highly optimized SGD w/ momentum + Polyak averaging

K-FAC approximation for recurrent layers

- The situation for RNNs is somewhat more complicated. We have

$$s_t = W a_t,$$

where t indexes the time-step from 1 to $\mathcal{T}$.

- Defining $g_t = \mathcal{D}s_t$ we have that

$$\mathcal{D}W = \sum_{t=1}^{\mathcal{T}} g_t a_t^\top$$

Recall notation:

$$\mathcal{D}V = \frac{\mathrm{d}L(y, f(x,\theta))}{\mathrm{d}V} = -\frac{\mathrm{d}\log p(y|x,\theta)}{\mathrm{d}V}$$

- Define $w_t = \mathrm{vec}(g_t a_t^\top)$ so that $\mathrm{vec}(\mathcal{D}W) = \sum_{t=1}^{\mathcal{T}} w_t$. Then we have

$$F = \mathbb{E}_{\mathcal{T}}[F_{\mathcal{T}}], \quad \text{where}$$

$$F_{\mathcal{T}} = \mathrm{cov}(\mathrm{vec}(\mathcal{D}W), \mathrm{vec}(\mathcal{D}W)|\mathcal{T}) = \mathrm{cov}\left(\sum_{t=1}^{\mathcal{T}} w_t, \sum_{t=1}^{\mathcal{T}} w_t \,\middle|\, \mathcal{T}\right) = \sum_{t=1}^{\mathcal{T}}\sum_{s=1}^{\mathcal{T}} \mathrm{cov}(w_t, w_s|\mathcal{T})$$

Basic initial approximations

- Denote $V_{t,s} = \text{cov}(w_t, w_s)$
- If we make the following approximating assumptions:
  - $\mathcal{T}$ is independent of the w_t's
  - $V_{t,s}$ depends only on $d = t - s$ and is given by V_d ("Temporal homogeneity")
  - a_t's and g_t's are independent (the original "K-FAC approximation"), so that:

    $V_d = A_d \otimes G_d$ where $A_{t-s} = A_{t,s} = \mathbb{E}[a_t a_s^\top]$ and $G_{t-s} = G_{t,s} = \mathbb{E}[g_t g_s^\top]$

  then we have the initial approximation:

$$F_{\mathcal{T}} = \sum_{d=-\mathcal{T}}^{\mathcal{T}} (\mathcal{T} - |d|) V_d = \sum_{d=-\mathcal{T}}^{\mathcal{T}} (\mathcal{T} - |d|)(A_d \otimes G_d)$$

Assuming independence across time

- Because a large sum of Kronecker products cannot be efficiently inverted we need to make additional approximating assumptions
- The simplest one we can make is to assume that the w_t's are independent across time (or more weakly that the g_t's are uncorrelated across time), so that $V_d = 0$ for $d \neq 0$.
- This gives us $F_{\mathcal{T}} = \sum_{d=-\mathcal{T}}^{\mathcal{T}} (\mathcal{T} - |d|) V_d = (\mathcal{T} - 0) V_0 = \mathcal{T} V_0 = \mathcal{T} A_0 \otimes G_0$

  and thus:

$$F = \mathbb{E}[\mathcal{T}](A_0 \otimes G_0)$$

This is just a single Kronecker-product and therefore easy to estimate and invert!

Modeling temporal relationships using an LGGM

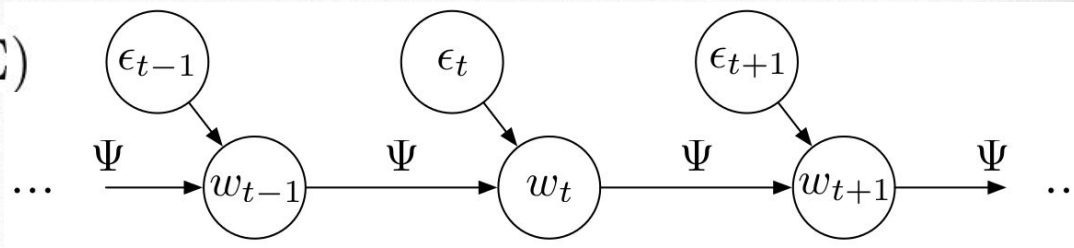
- Instead of assuming that temporal relationships between the w_t's is non-existent we can try to model them using a simple statistical model
- Perhaps the simplest such (non-trivial) model is a chained structured Linear Gaussian Graphical Model (LGGM) defined by

$$w_t = \Psi w_{t-1} + \epsilon_t$$

where,

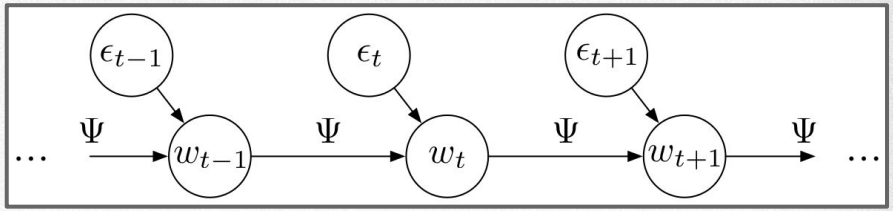
$$\epsilon_t \text{ are i.i.d. from } \mathcal{N}(0, \Sigma)$$

and Ψ is a square matrix with spectral radius < 1



- simplify the computations we will assume that this models extends infinitely in both directions

Initial computations



- It is straightforward to show that

$$\Psi = V_1 V_0^{-1}$$

- Define “transformed” quantities

$$\hat{F}_{\mathcal{T}} = V_0^{1/2} F_{\mathcal{T}} V_0^{1/2} \quad \text{and} \quad \hat{\Psi} = \hat{V}_1 = V_0^{-1/2} \Psi V_0^{1/2}.$$

- And note that because we have

$$F^{-1} = V_0^{-1/2} \hat{F}^{-1} V_0^{-1/2}$$

it suffices to compute $\hat{F}^{-1}$

Option 1: V_1 is symmetric

- If we assume that V_1, the 1-step temporal cross-covariance, is symmetric, this implies that $\hat{\Psi}$ is symmetric

- Let $U \operatorname{diag}(\hat{\psi}) U^\top = \hat{\Psi}$ be the eigendecomposition of $\hat{\Psi}$

- It can be shown that

$$\hat{F}^{-1} = U \operatorname{diag}(\gamma(\hat{\psi})) U^\top$$

where

$$\gamma(x) = 1/\mathbb{E}_{\mathcal{T}}[\eta_{\mathcal{T}}(x)] \quad \text{with} \quad \eta_{\mathcal{T}}(x) = \frac{\mathcal{T}(1-x^2) - 2x(1-x^{\mathcal{T}})}{(1-x)^2}$$

Option 2: Using the limiting value as $\mathcal{T} \to \infty$

- A second option to obtain a tractable formula is to compute the limiting value:

$$\hat{F} = \mathbb{E}_{\mathcal{T}}[\hat{F}_{\mathcal{T}}^{(\infty)}]$$

where we define

$$\hat{F}_{\mathcal{T}}^{(\infty)} \equiv \lim_{\mathcal{T}' \to \infty} \frac{\mathcal{T}}{\mathcal{T}'} \hat{F}_{\mathcal{T}'}.$$

This gives (with some work) the remarkably simple expression:

$$\hat{F}^{-1} = \frac{1}{\mathbb{E}_{\mathcal{T}}[\mathcal{T}]}(I - \hat{\Psi})(I - \hat{\Psi}^{\top}\hat{\Psi})^{-1}(I - \hat{\Psi}^{\top})$$

Efficient computation with Kronecker products

- The formulae for $\hat{F}^{-1}$ in **Option 1** and **Option 2** can be used to efficiently multiply a vector by $\hat{F}^{-1}$, starting from the identities:

$$V_0 = A_0 \otimes G_0 \quad \text{and} \quad V_1 = A_1 \otimes G_1$$

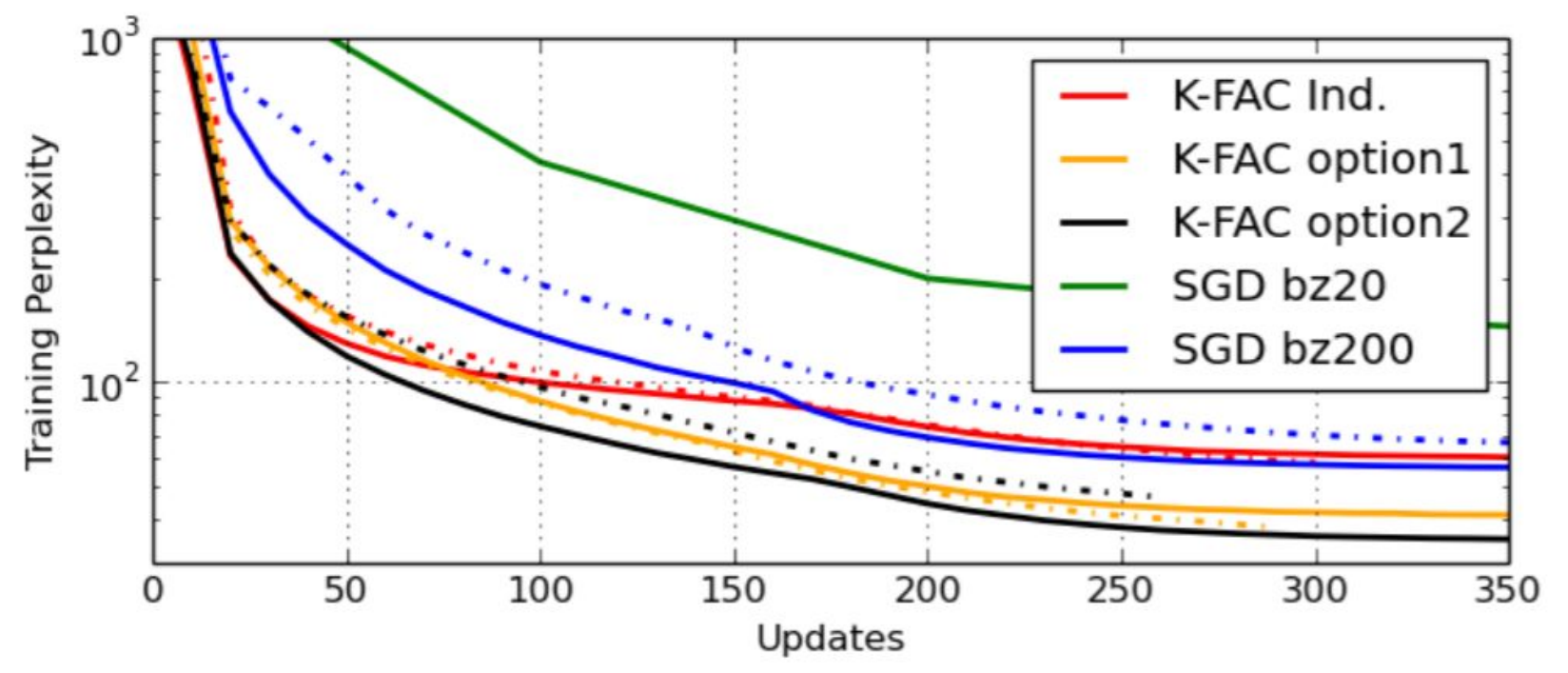
  (Boils down to several eigen-decompositions and a dozen or so matrix-matrix multiplications with ***d*** by ***d*** *matrices*, where ***d*** = layer width.)

- Cost of these operations is independent of $\mathcal{T}$, and can be amortized over iterations and parallelized.

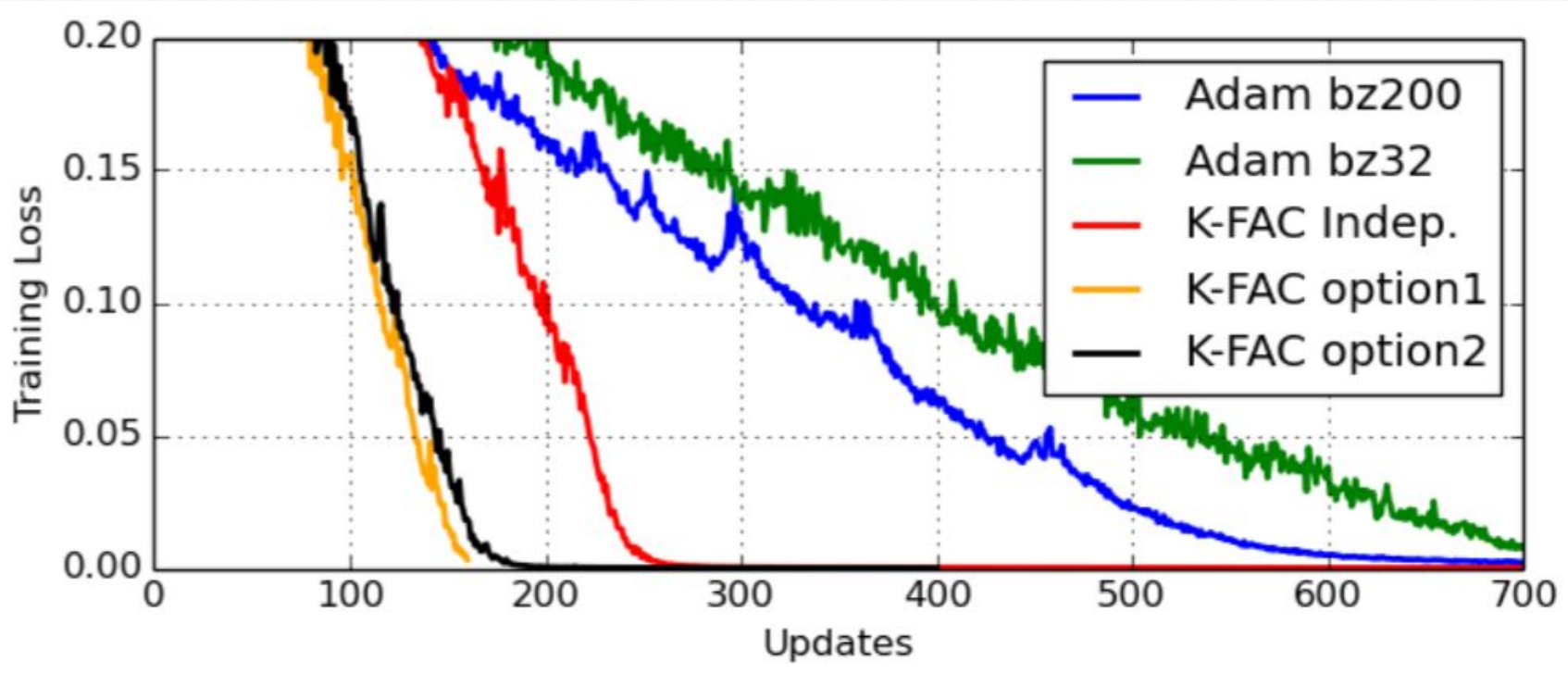
- Factors estimated using decayed averages that are also averaged over time-steps. e.g.

$$G_1 = \frac{1}{\mathcal{T}} \sum_{t=1}^{\mathcal{T}} \mathbb{E}[g_{t+1} g_t^\top]$$

Experiment 1: 2-layer LSTM on Penn TreeBank



Experiment 2: DNC “copy task”



Kronecker approximation for conv layers (KFC)

- A convolutional layer can be described as follows:
  - extract a "patch vector" a_t for each "location" $t \in \{1, 2, ..., \mathcal{T}\}$ from the image/feature map incoming to the layer
  - multiply each patch vector by a "filter bank" matrix W:

$$s_t = W a_t,$$

  - form the output feature map from the s_t's according location t
- Gradient is once again just $\mathcal{D}W = \sum_{t=1}^{\mathcal{T}} g_t a_t^\top$ where $g_t = \mathcal{D}s_t$
- This is structurally very similar to the recurrent case, with locations playing the role of time-steps

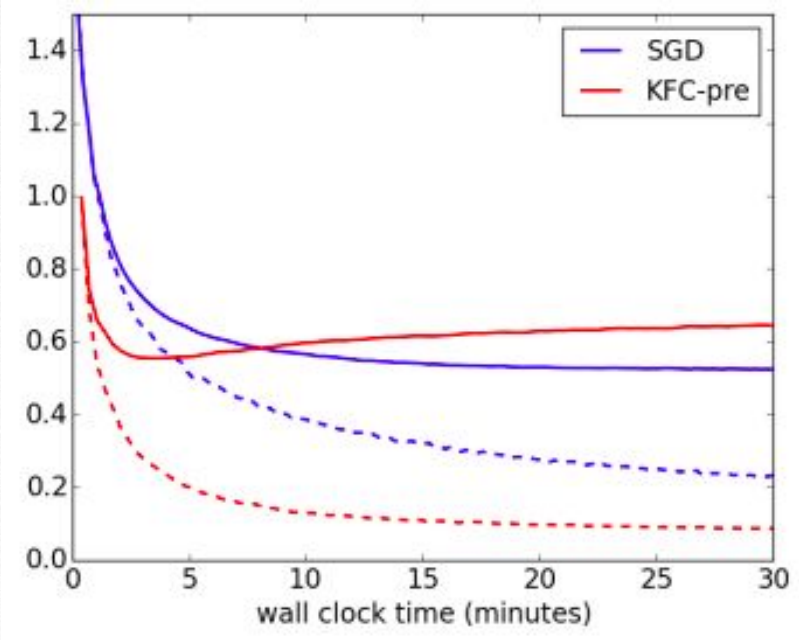
Kronecker approximation for conv layers (KFC)

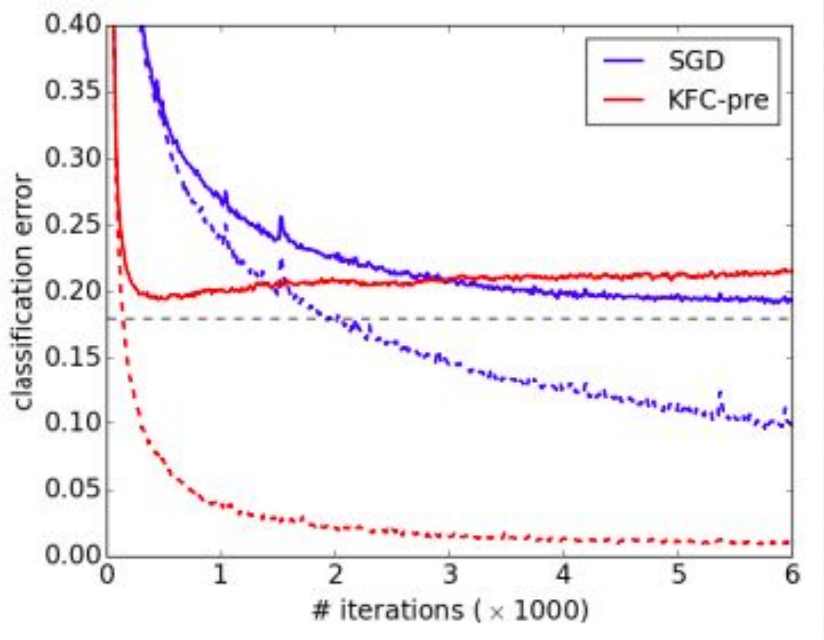
- If we make the following approximating assumptions:
  - the a_t's are independent of the g_t's,
  - different g_t's uncorrelated,
  - the distributions of a_t and g_t don't depend on index t (i.e. "spatially homogeneous")

  Then following a similar (but simpler) argument to the recurrent case, the Fisher block for W is given by $$F = \mathcal{T} \cdot (A \otimes G)$$

  Factors estimated using decayed averages that are also averaged over locations. e.g. $$A = \frac{1}{\mathcal{T}} \sum_{t=1}^{\mathcal{T}} \mathbb{E}[a_t a_t^\top]$$

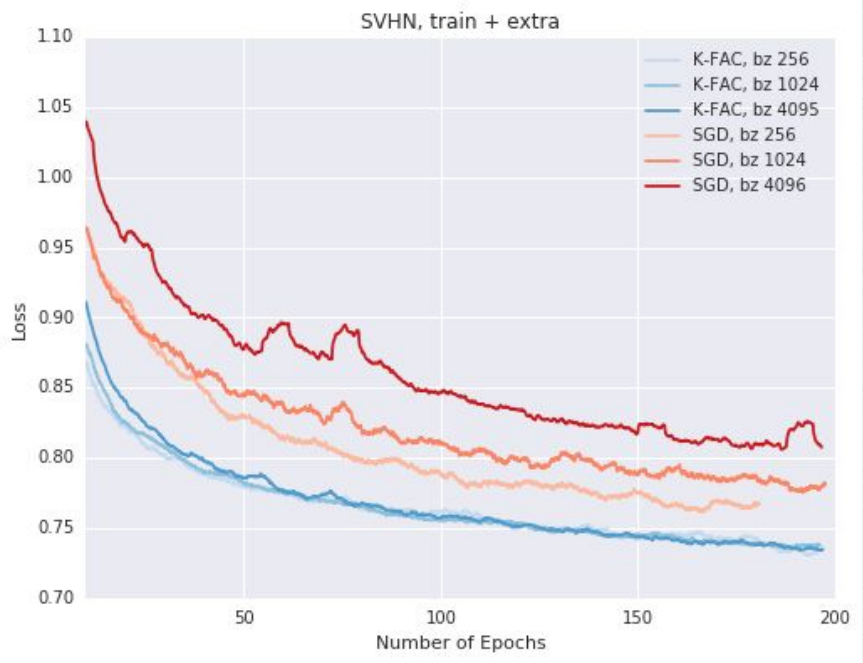
CIFAR-10 convnet





Recent large mini-batch experiments

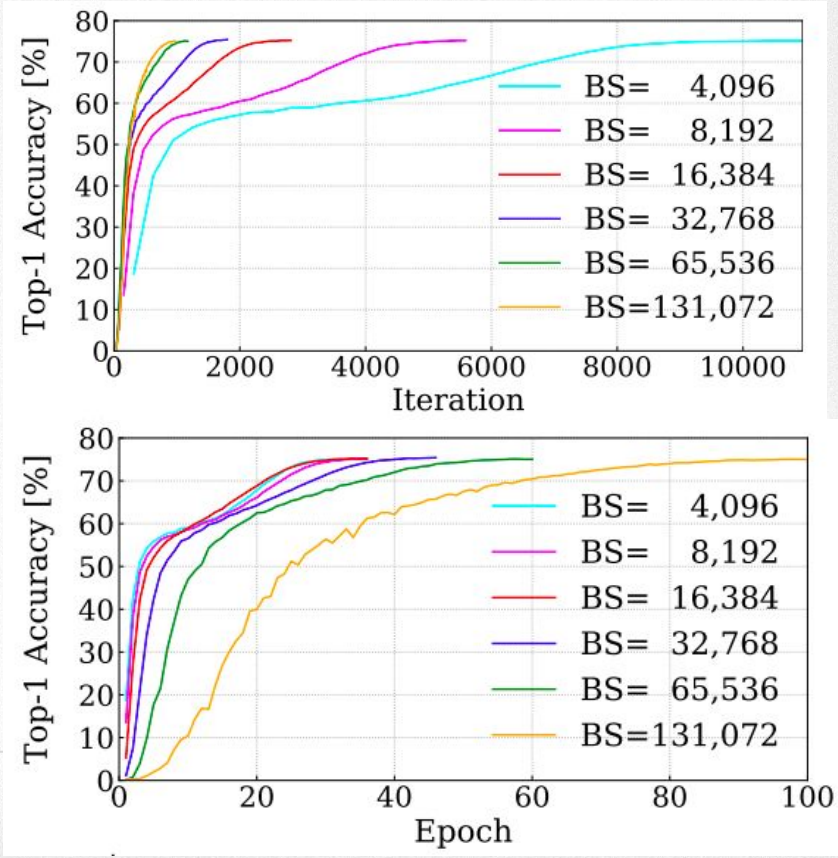
- Resnet-50 trained on augmented SVHN dataset
- K-FAC maintains data efficiency as batch size increases while SGD w/ momentum baseline tops out quickly



Credit: Daniel Duckworth

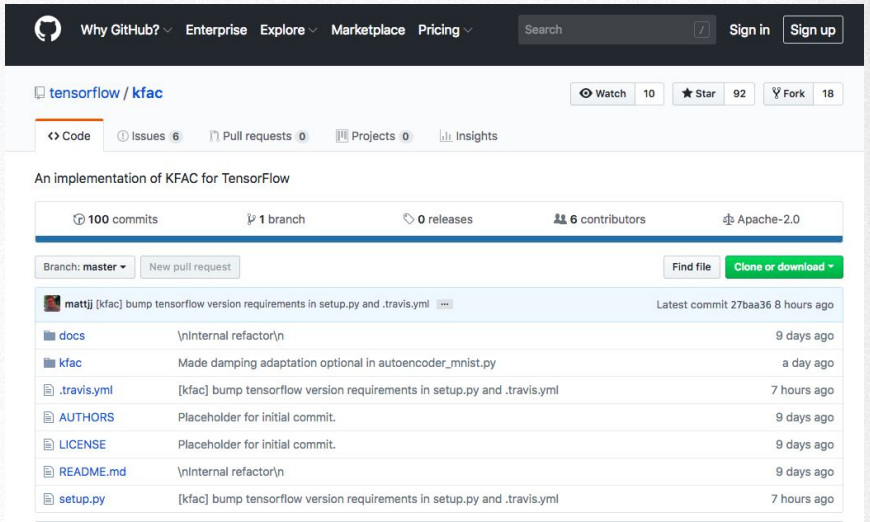
Recent large mini-batch experiments

- Recent paper from the RIKEN lab has applied K-FAC to Resnet-50 on *Imagenet*
- They use extremely large mini-batches up to 130k with massively parallel computation
- Show significant improvement in number of iterations all the way up to mini-batch sizes of 65k



Public TensorFlow implementation

- There is a highly sophisticated implementation of K-FAC in TensorFlow [available on Github]
- Supports the following and more:
  - Fully-connected, convolutional, and recurrently layers
  - Various distribution strategies
  - Automatic structure determination of the graph
  - Automatic adjustment of damping, learning rate and momentum



Thanks for listening!
Questions?