Scalable natural gradient using probabilistic models of backprop

Roger Grosse

UNIVERSITY OF TORONTO

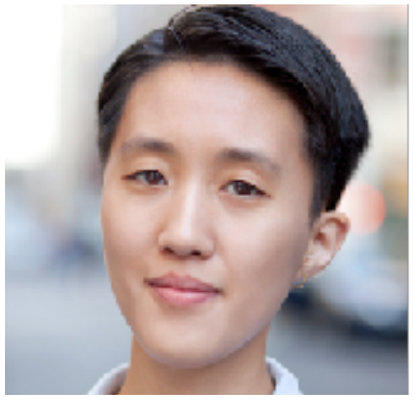
VECTOR INSTITUTE

Overview

- Overview of natural gradient and second-order optimization of neural nets
- Kronecker-Factored Approximate Curvature (K-FAC), an approximate natural gradient optimizer which scales to large neural networks
    - based on fitting a probabilistic graphical model to the gradient computation
- Current work: a variational Bayesian interpretation of K-FAC

Overview

Background material from a forthcoming Distill article.



Katherine Ye

Matt Johnson

Chris Olah

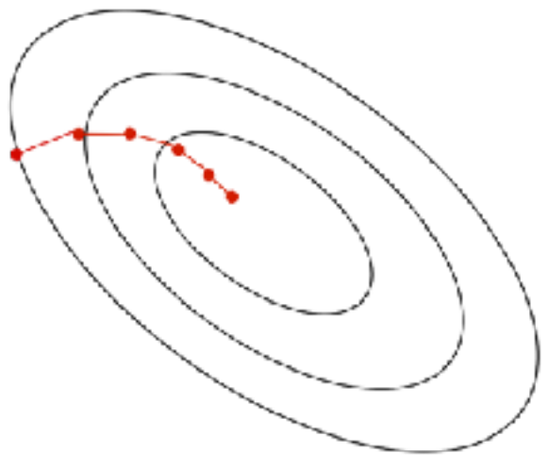
Overview

Most neural networks are still trained using variants of **stochastic gradient descent (SGD)**.

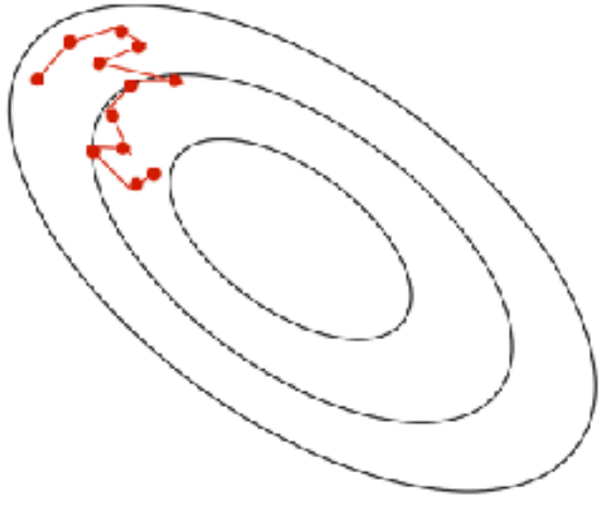
Variants: SGD with momentum, Adam, etc.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{t})$$

- α: learning rate
- $f(\mathbf{x}, \boldsymbol{\theta})$: network's predictions
- $\mathbf{t}$: label
- $\boldsymbol{\theta}$: parameters (weights/biases)
- $\mathcal{L}$: loss function
- $\mathbf{x}$: input



batch gradient descent



stochastic gradient descent

Backpropagation is a way of computing the gradient, which is fed into an optimization algorithm.

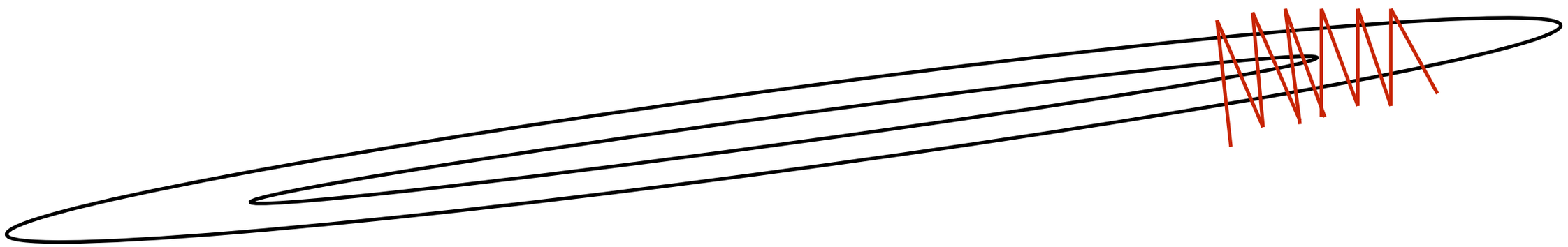
Overview

SGD is a first-order optimization algorithm (only uses first derivatives)

First-order optimizers can perform badly when the **curvature is badly conditioned**

bounce around a lot in high curvature directions

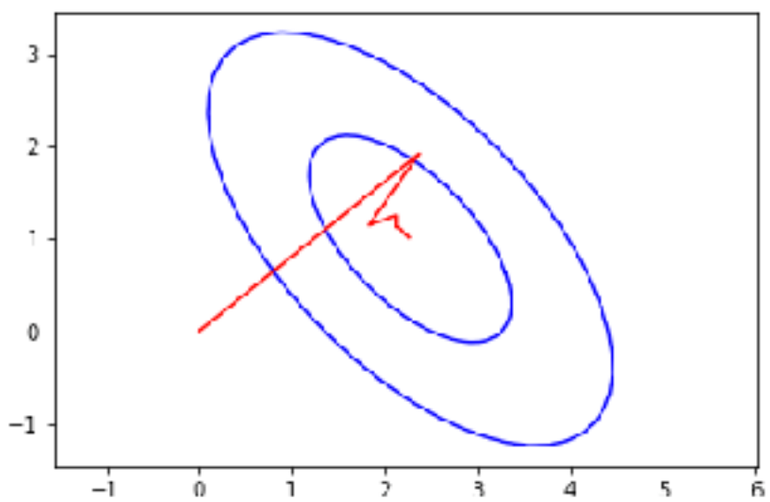
make slow progress in low curvature directions



Recap: normalization

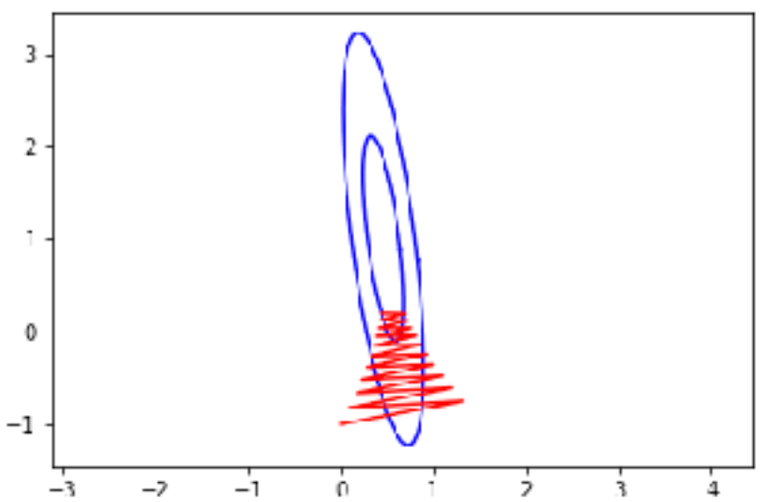
original data

x_1	x_2	y
0.49	0.18	0.79
-1.67	-0.46	-4.43
0.22	0.37	1.12
1.76	-0.22	3.36
⋮	⋮	⋮



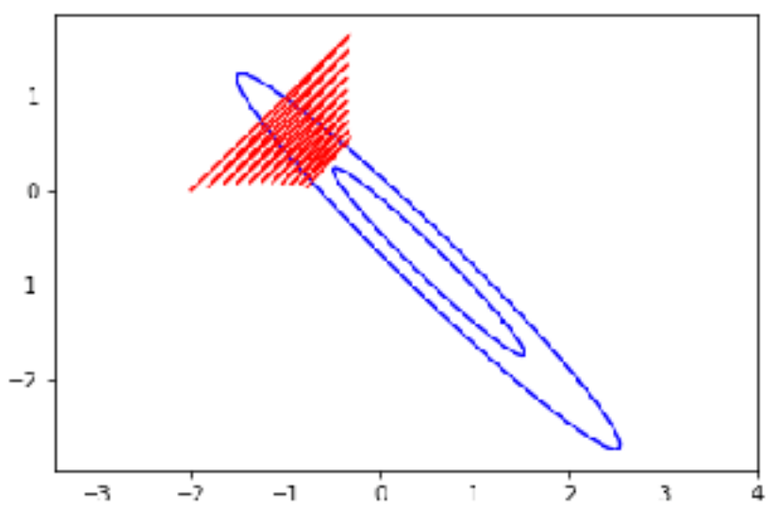
multiply x_1 by 5

x_1	x_2	y
2.43	0.18	0.79
-8.33	-0.46	-4.43
1.11	0.37	1.12
8.79	-0.22	3.36
⋮	⋮	⋮

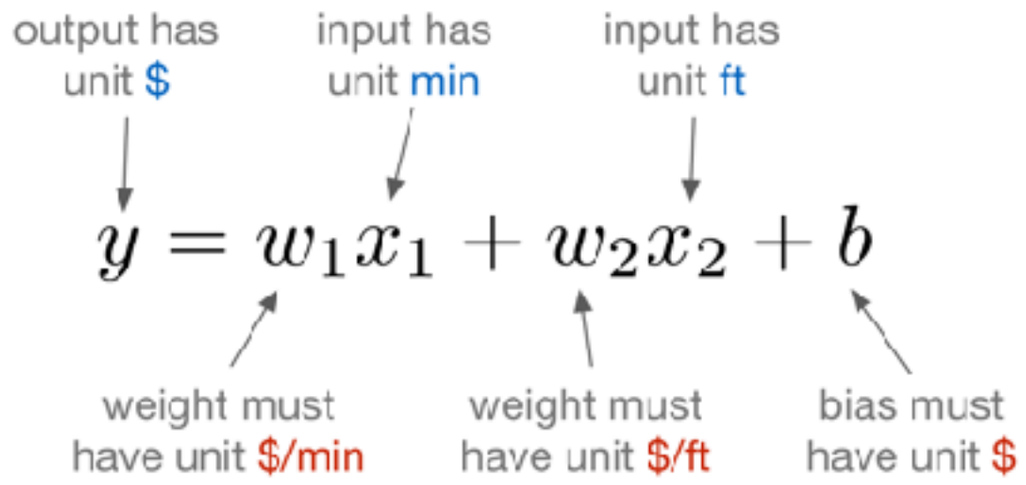


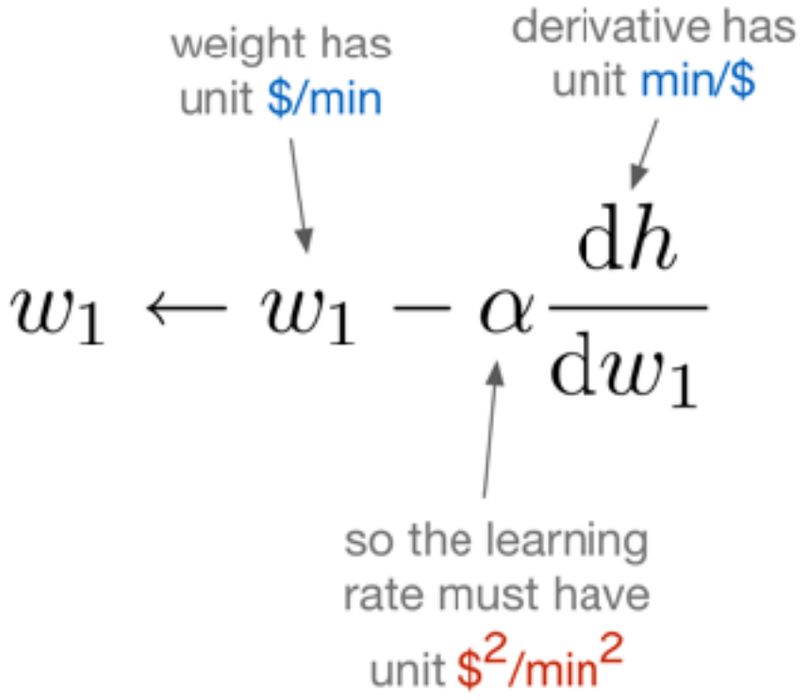
add 5 to both

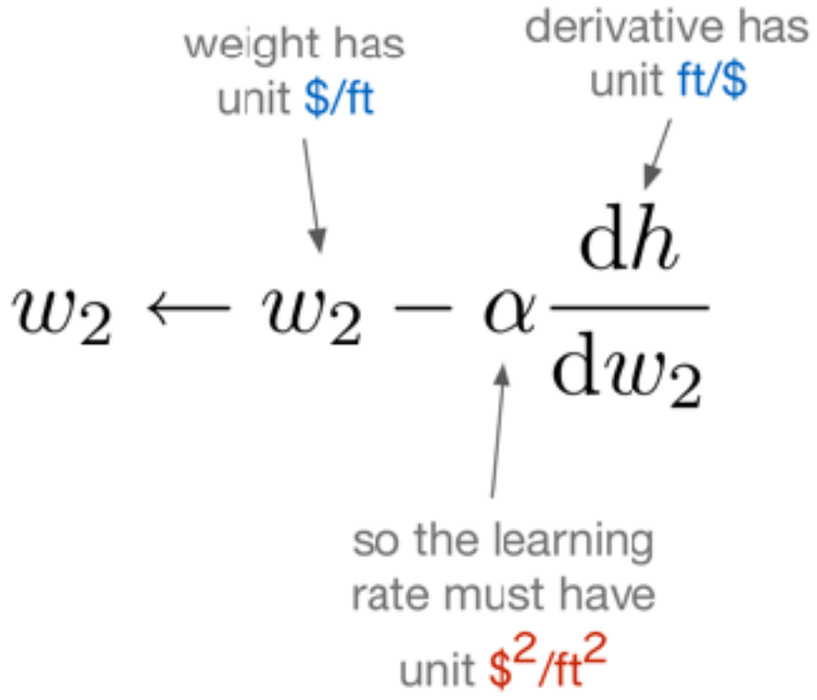
x_1	x_2	y
5.49	5.18	0.79
3.33	4.54	-4.43
5.22	5.37	1.12
6.76	4.78	3.36
⋮	⋮	⋮



Recap: normalization







Recap: normalization

parameterize
in minutes

$$w_1 \leftarrow w_1 - 0.02 \xleftarrow{\text{GD with LR = 0.01}} \frac{\mathrm{d}h}{\mathrm{d}w_1} = 2 \qquad w_1 \leftarrow w_1 - 72$$

$$\frac{\mathrm{d}h}{\mathrm{d}\tilde{w}_1} = 60\frac{\mathrm{d}h}{\mathrm{d}w_1} \qquad w_1 = 60\tilde{w}_1$$

parameterize
in seconds

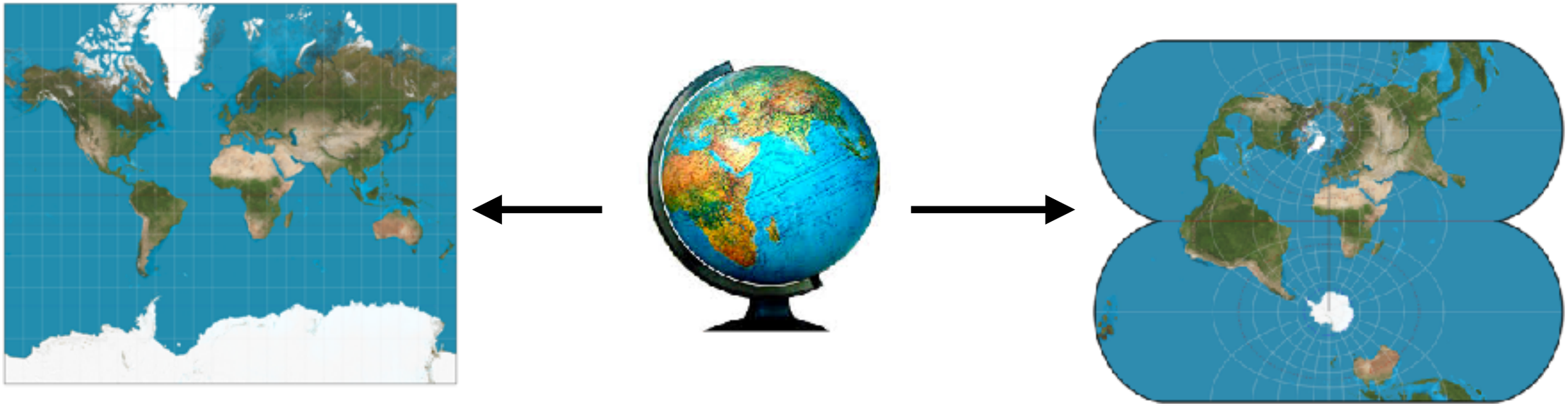
$$\frac{\mathrm{d}h}{\mathrm{d}\tilde{w}_1} = 120 \xrightarrow[\text{LR = 0.01}]{\text{GD with}} \tilde{w}_1 \leftarrow \tilde{w}_1 - 1.2$$

Background: neural net optimization

These 2-D cartoons are misleading.

Millions of optimization variables, contours stretched by a factor of millions

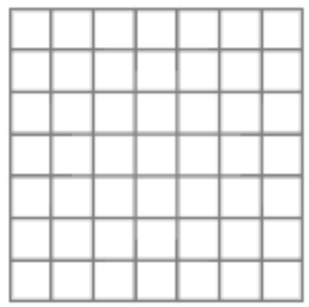
When we train a network, we're trying to learn a function, but we need to parameterize it in terms of weights and biases.



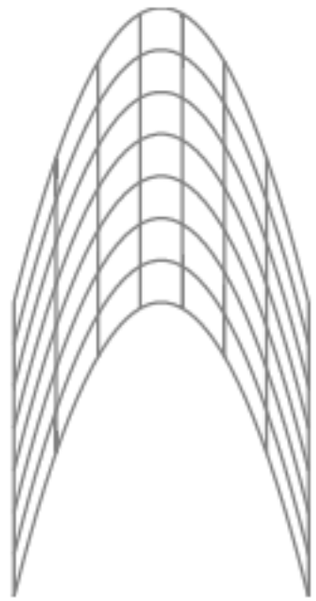
Mapping a manifold to a coordinate system **distorts distances**

Natural gradient: compute the gradient on the globe, not on the map

Recap: Rosenbrock Function

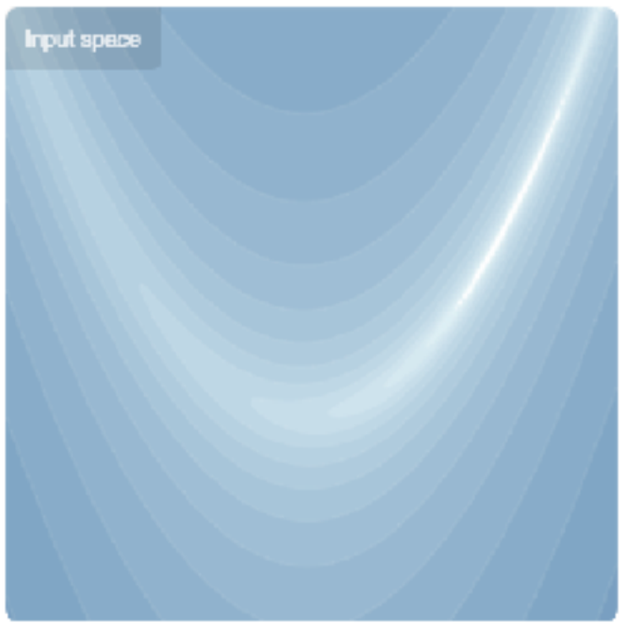


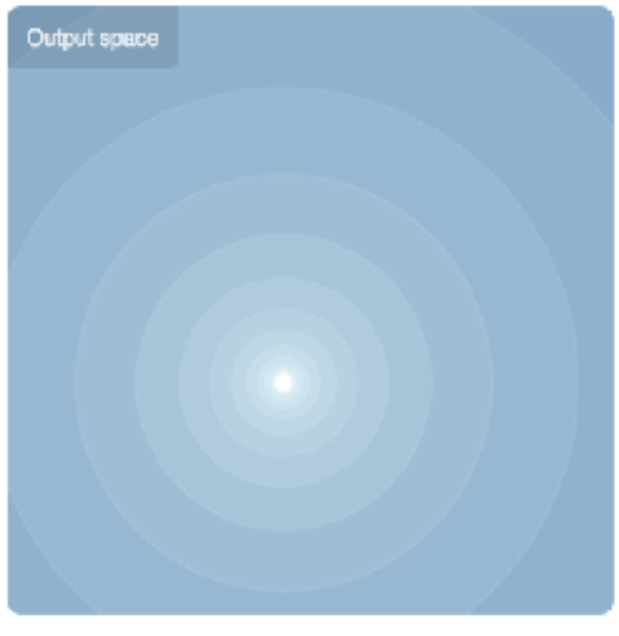
$\xrightarrow{f}$



$$f(x_1, x_2) = (a - x_1, \sqrt{b}(x_2 - x_1^2))$$

$$= (y_1, y_2)$$

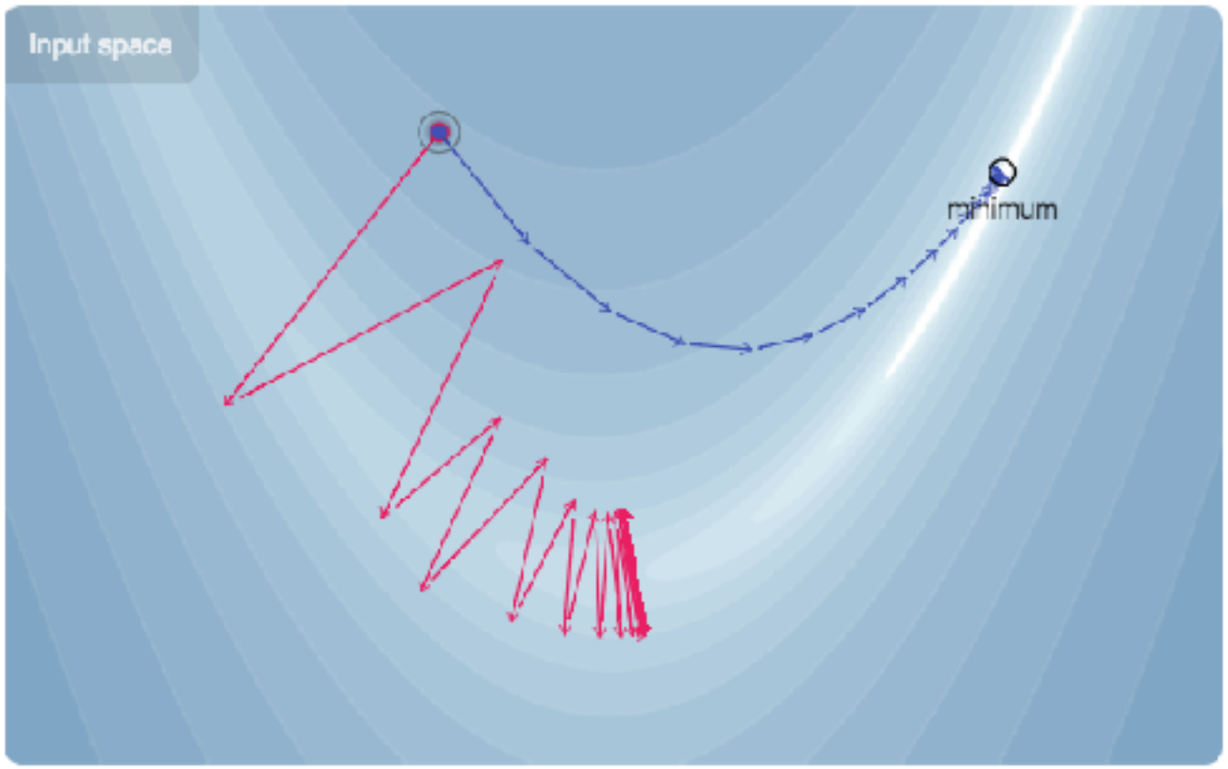


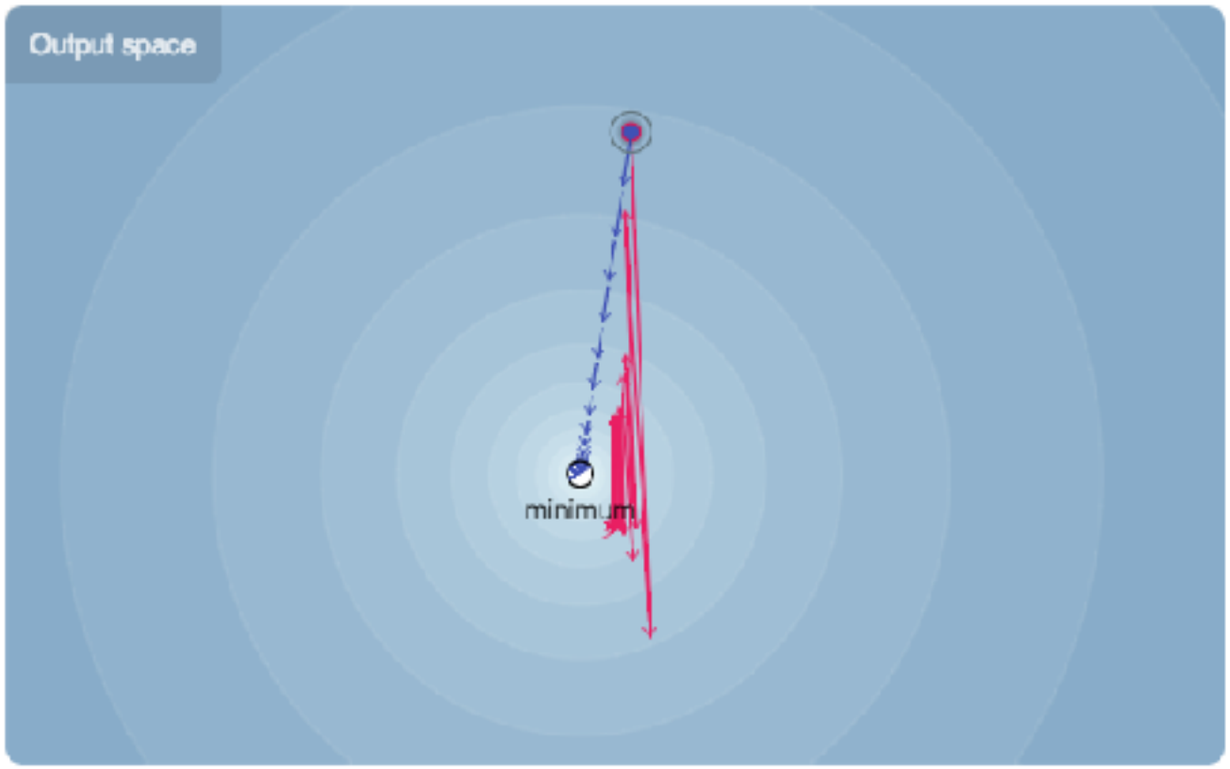


$$h(y_1, y_2) = L(y_1, y_1) = y_1^2 + y_2^2.$$

Recap: steepest descent

If only we could do gradient descent on output space…





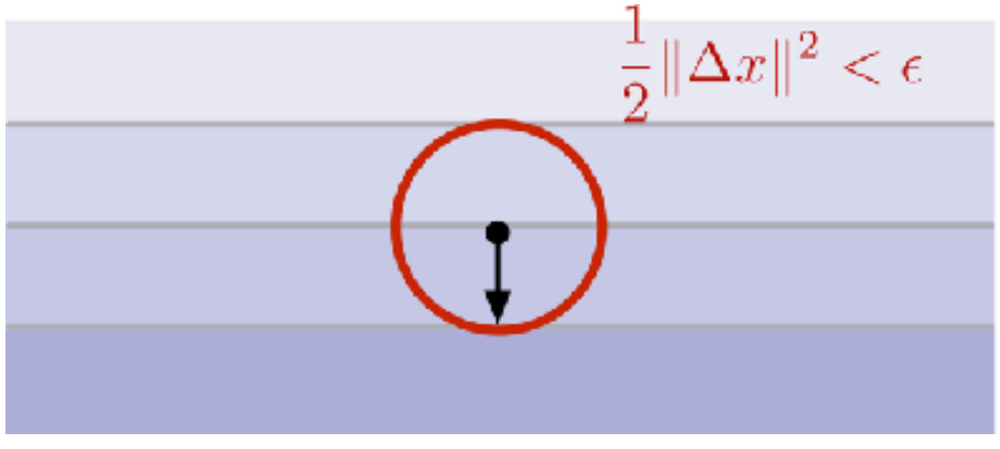
Recap: steepest descent

Steepest descent:

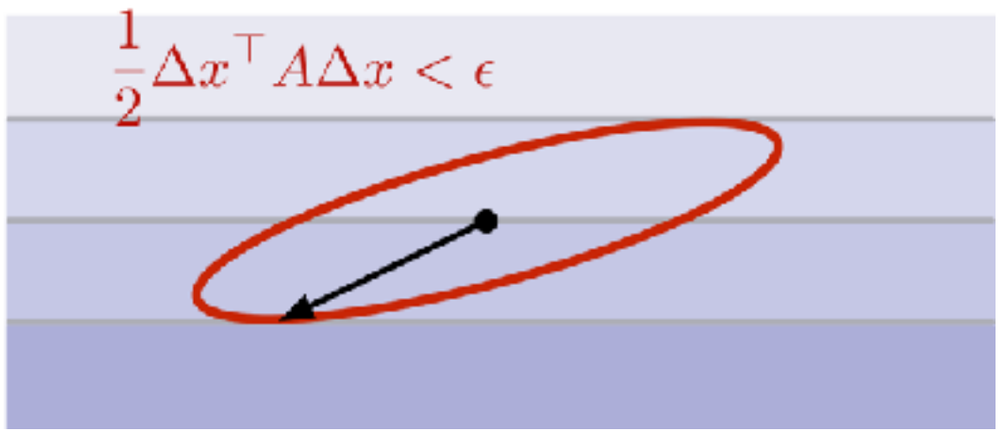
$$x^{k+1} \leftarrow \arg\min_x \left\{ \underline{\nabla h(x^k)^\top (x - x^k)} + \underline{\lambda \mathrm{D}(x, x^k)} \right\},$$

linear approximation

dissimilarity measure



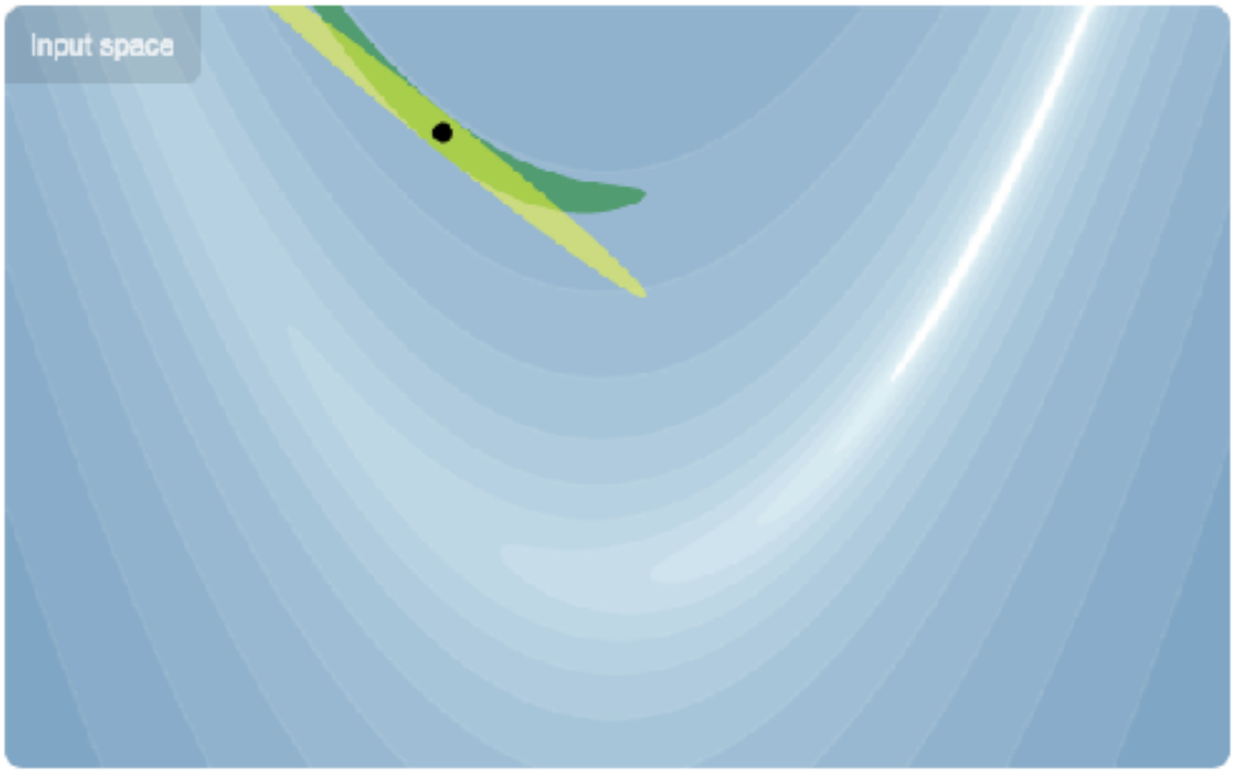
Euclidean D
=> gradient descent

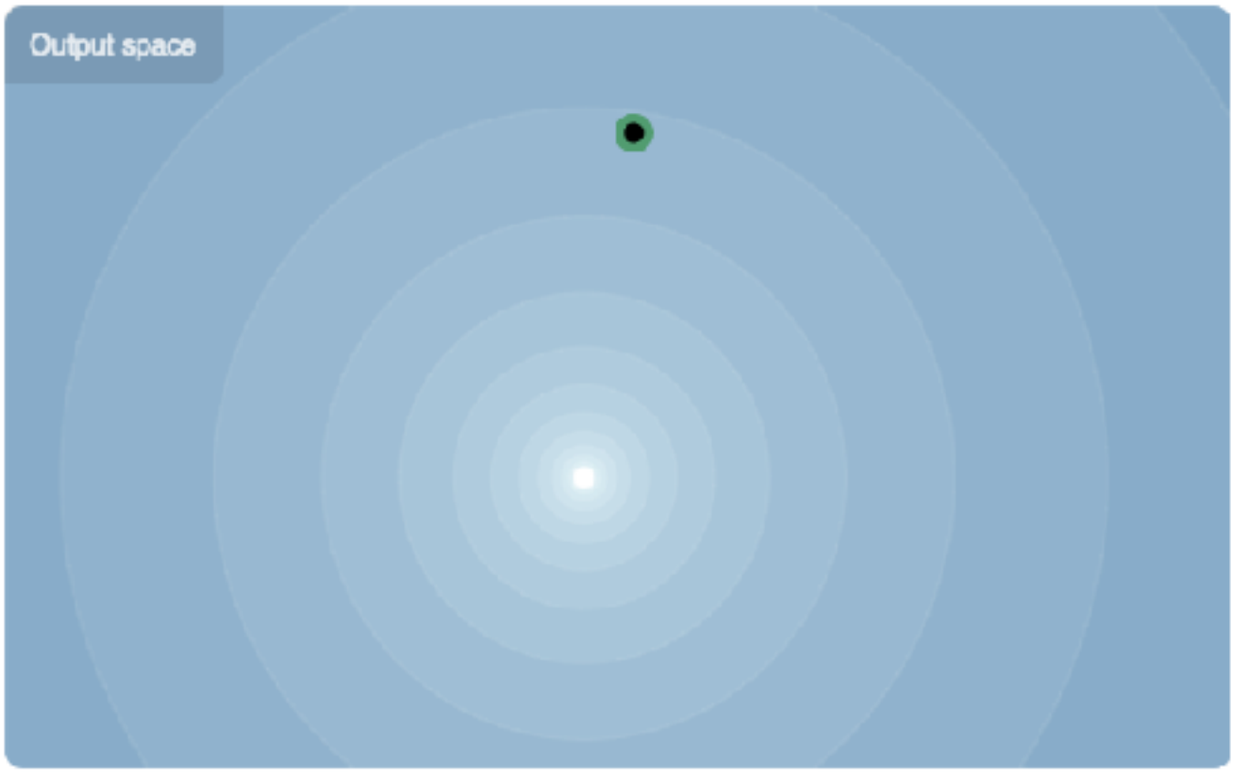


Another Mahalanobis (quadratic) metric

Recap: steepest descent

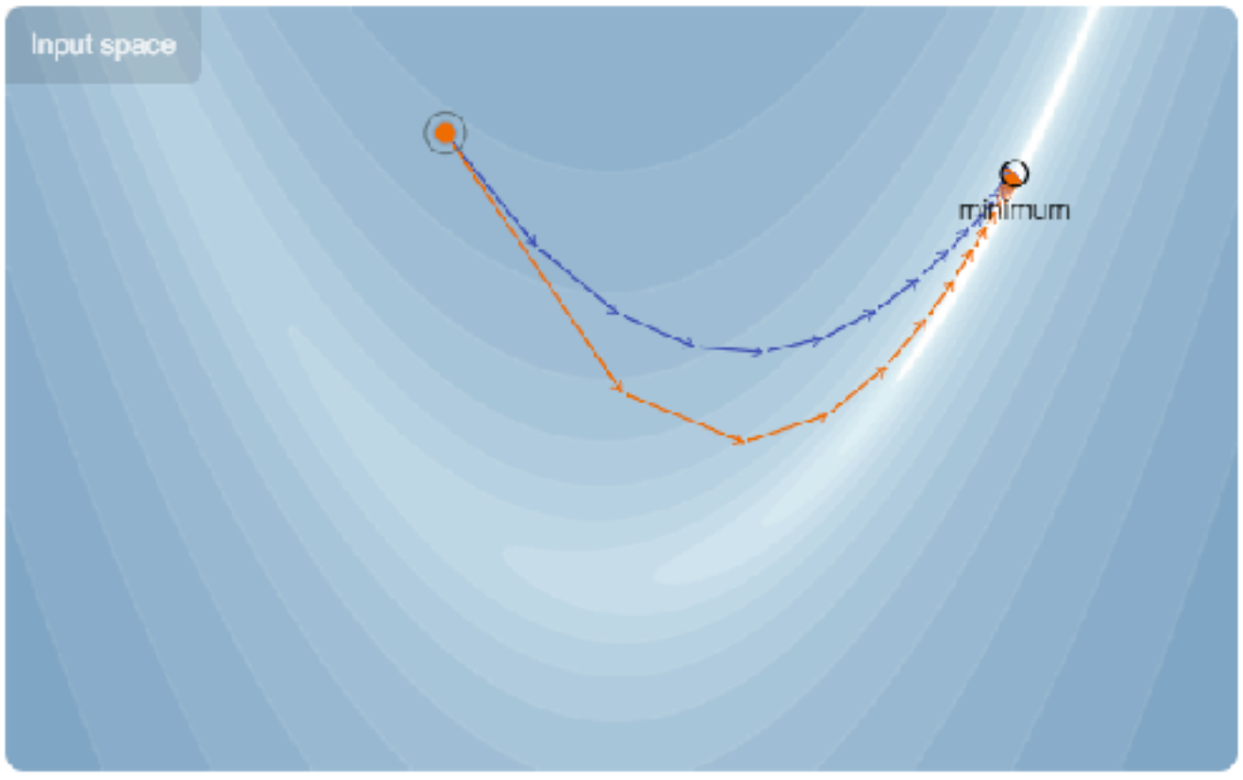
Take the quadratic approximation:

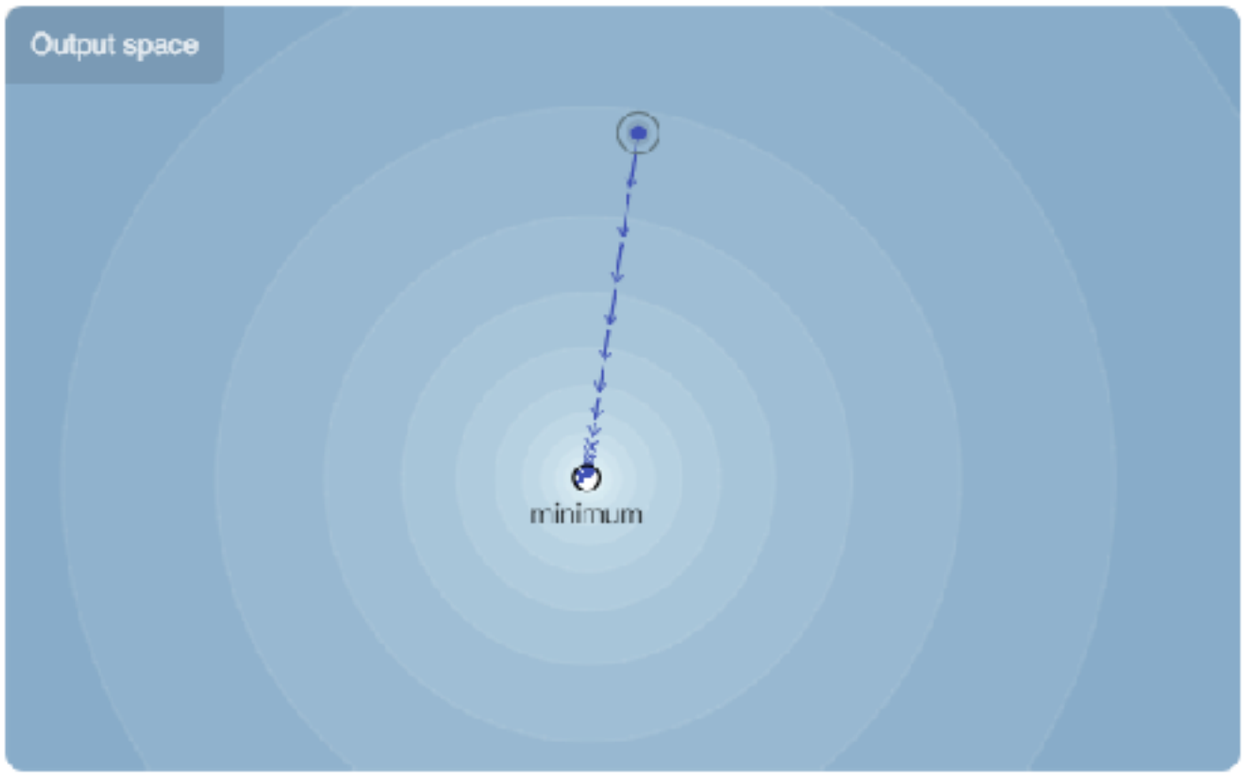




Recap: steepest descent

Steepest descent mirrors gradient descent in output space:





Even though “gradient descent on output space” has no analogue for neural nets, this steepest descent insight does generalize!

Recap: Fisher metric and natural gradient

For fitting probability distributions (e.g. maximum likelihood), a natural dissimilarity measure is KL divergence.

$$\mathrm{D}_{\mathrm{KL}}(q||p) = \mathbb{E}_{x\sim q}[\log q(x) - \log p(x)]$$

The second-order Taylor approximation to KL divergence is the Fisher information matrix:

$$\nabla_\theta^2 \mathrm{D}_{\mathrm{KL}} = F = \mathrm{Cov}_{x\sim p_\theta}(\nabla_\theta \log p_\theta(x))$$

Steepest ascent direction, called the **natural gradient:**

$$\tilde{\nabla}_\theta h = F^{-1}\nabla_\theta h$$

Recap: Fisher metric and natural gradient

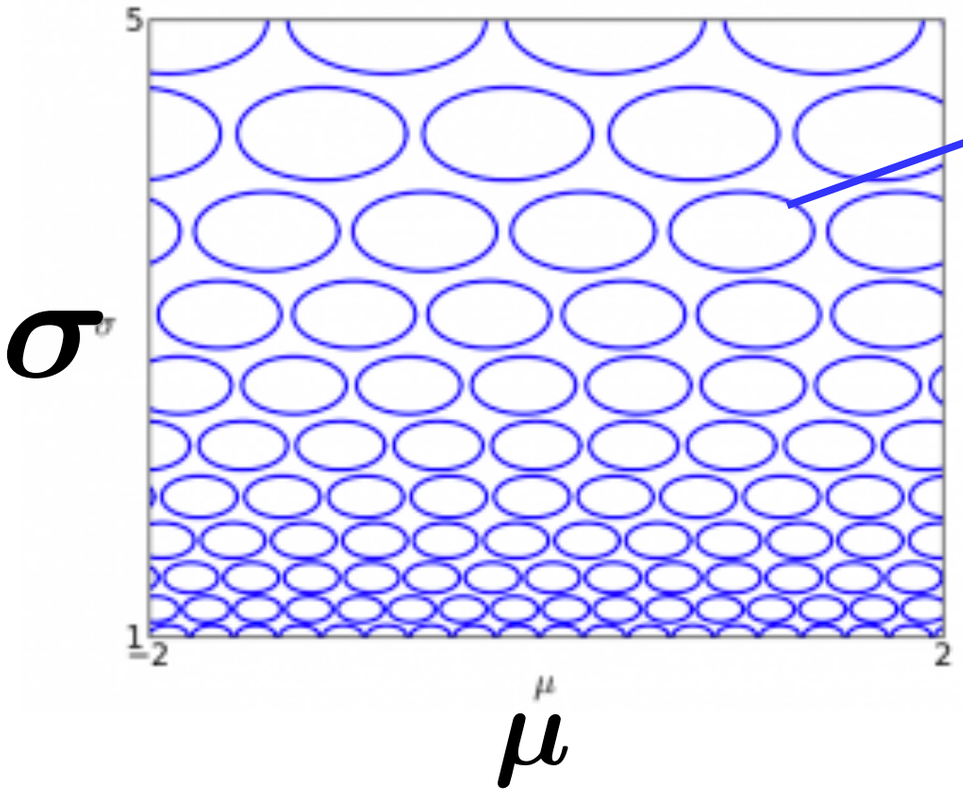
If you phrase your algorithm in terms of Fisher information,
it's invariant to reparameterization.

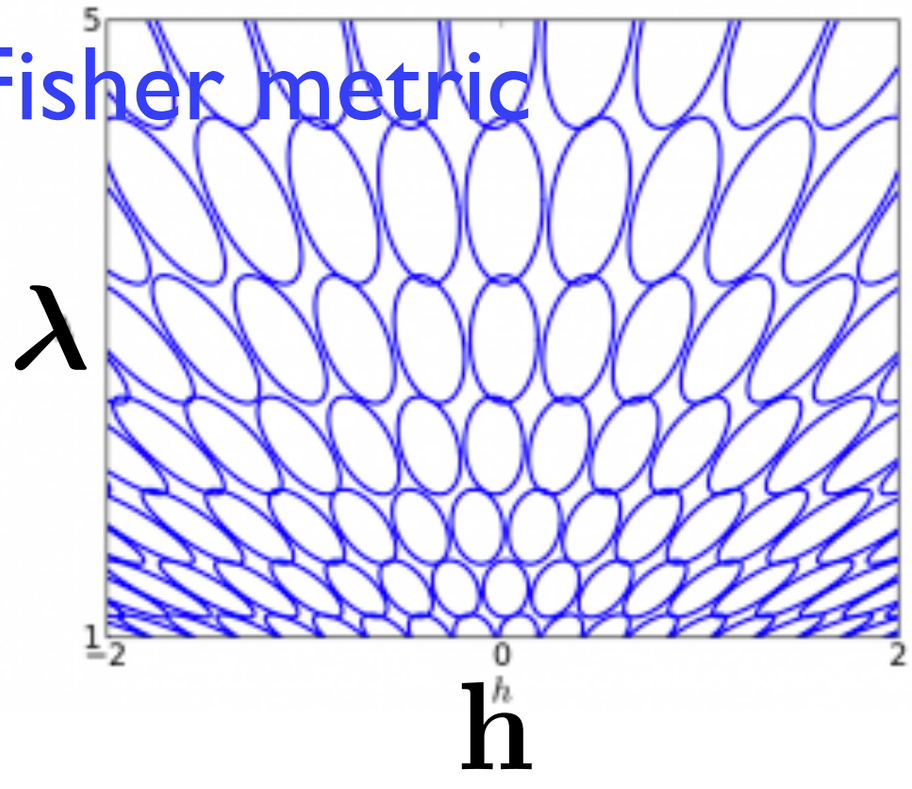
mean and variance

information form

unit of

Fisher metric





$$p(x) \propto \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

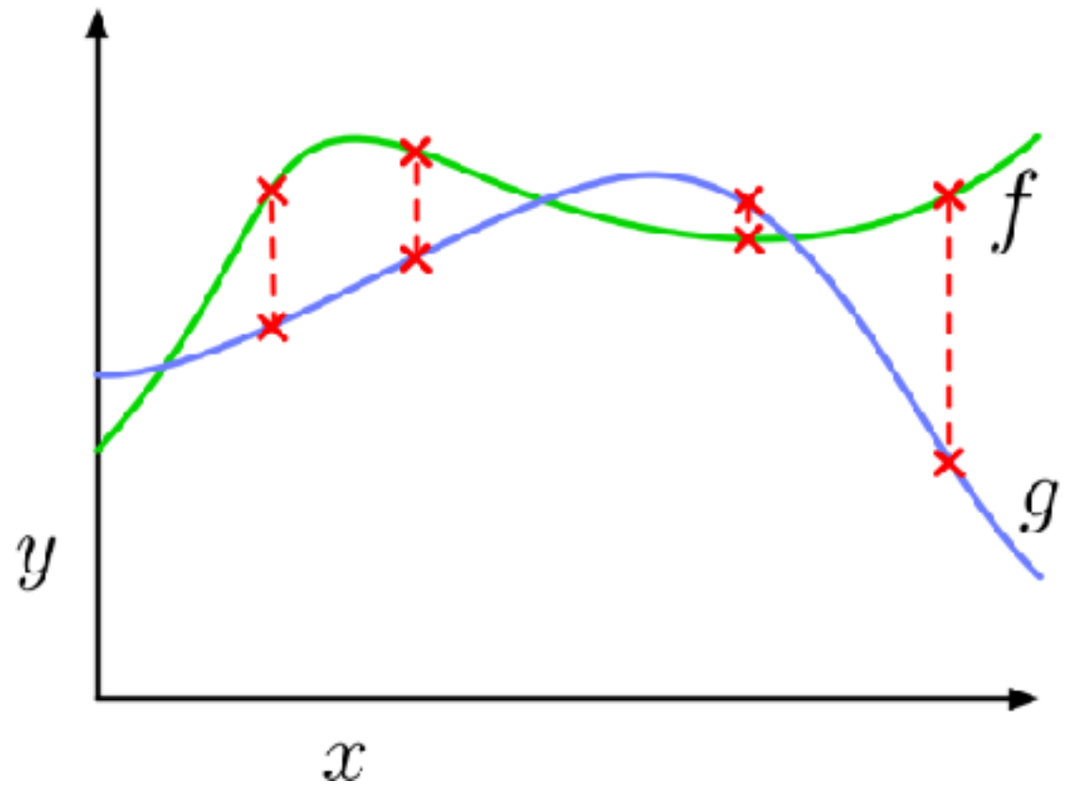
$$p(x) \propto \exp\left(hx - \frac{\lambda}{2}x^2\right)$$

Background: natural gradient

When we train a neural net, we're learning a function. How do we define a distance between functions?

Assume we have a dissimilarity metric d on the output space,
e.g. $\rho(y_1, y_2) = \|y_1 - y_2\|^2$

$$D(f, g) = \mathbb{E}_{x \sim \mathcal{D}}[\rho(f(x), g(x))]$$



Second-order Taylor approximation:

$$D(f_\theta, f_{\theta'}) \approx \frac{1}{2}(\theta' - \theta)^\top \mathbf{G}_\theta (\theta' - \theta)$$

$$\mathbf{G}_\theta = \frac{\partial y}{\partial \theta}^\top \frac{\partial^2 \rho}{\partial y^2} \frac{\partial y}{\partial \theta}$$

This is the **generalized Gauss-Newton matrix**.

Background: natural gradient (Amari, 1998)

Many neural networks output a predictive distribution (e.g. over categories).

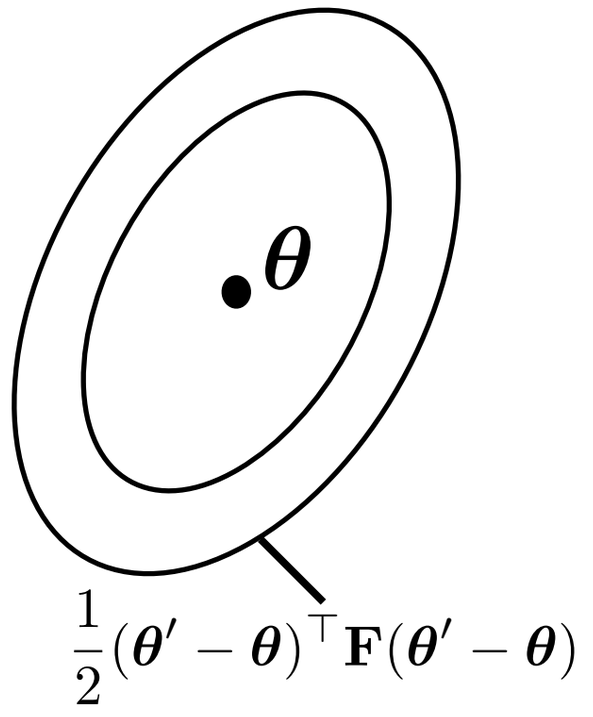
We can measure the "distance" between two networks in terms of the average KL divergence between their predictive distributions $r_{\boldsymbol{\theta}}(\mathbf{y} \,|\, \mathbf{x})$

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

$$\mathbf{F}_{\boldsymbol{\theta}} \triangleq \mathbb{E}\left[\nabla^2_{\boldsymbol{\theta}'} \mathrm{D}_{\mathrm{KL}}(r_{\boldsymbol{\theta}'}(\mathbf{y} \,|\, \mathbf{x}) \,\|\, r_{\boldsymbol{\theta}}(\mathbf{y} \,|\, \mathbf{x}))\Big|_{\boldsymbol{\theta}'=\boldsymbol{\theta}}\right]$$

This equals the covariance of the log-likelihood derivatives:

$$\mathbf{F}_{\boldsymbol{\theta}} = \operatorname*{Cov}_{\substack{\mathbf{x}\sim p_{\mathrm{data}} \\ \mathbf{y}\sim r_{\boldsymbol{\theta}}(\mathbf{y} \,|\, \mathbf{x})}} \left(\nabla_{\boldsymbol{\theta}} \log r_{\boldsymbol{\theta}}(\mathbf{y} \,|\, \mathbf{x})\right)$$



Three optimization algorithms

Newton-Raphson

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{H}^{-1} \nabla h(\boldsymbol{\theta})$$

Hessian matrix

$$\mathbf{H} = \frac{\partial^2 h}{\partial \boldsymbol{\theta}^2}$$

Generalized Gauss-Newton

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{G}^{-1} \nabla h(\boldsymbol{\theta})$$

GGN matrix

$$\mathbf{G} = \mathbb{E}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2} \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

Natural gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{F}^{-1} \nabla h(\boldsymbol{\theta})$$

Fisher information matrix

$$\mathbf{F} = \mathrm{Cov}\left(\frac{\partial}{\partial \boldsymbol{\theta}} \log p(y|\mathbf{x})\right)$$

Are these related?

Three optimization algorithms

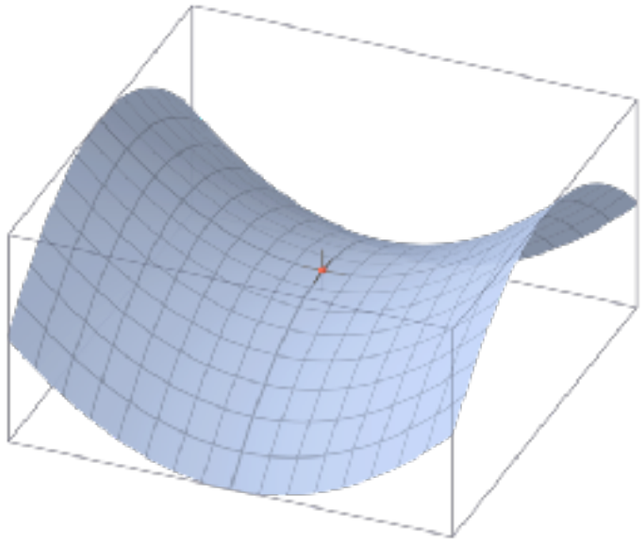
Newton-Raphson is the canonical second-order optimization algorithm.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{H}^{-1} \nabla h(\boldsymbol{\theta}) \qquad \mathbf{H} = \frac{\partial^2 h}{\partial \boldsymbol{\theta}^2}$$

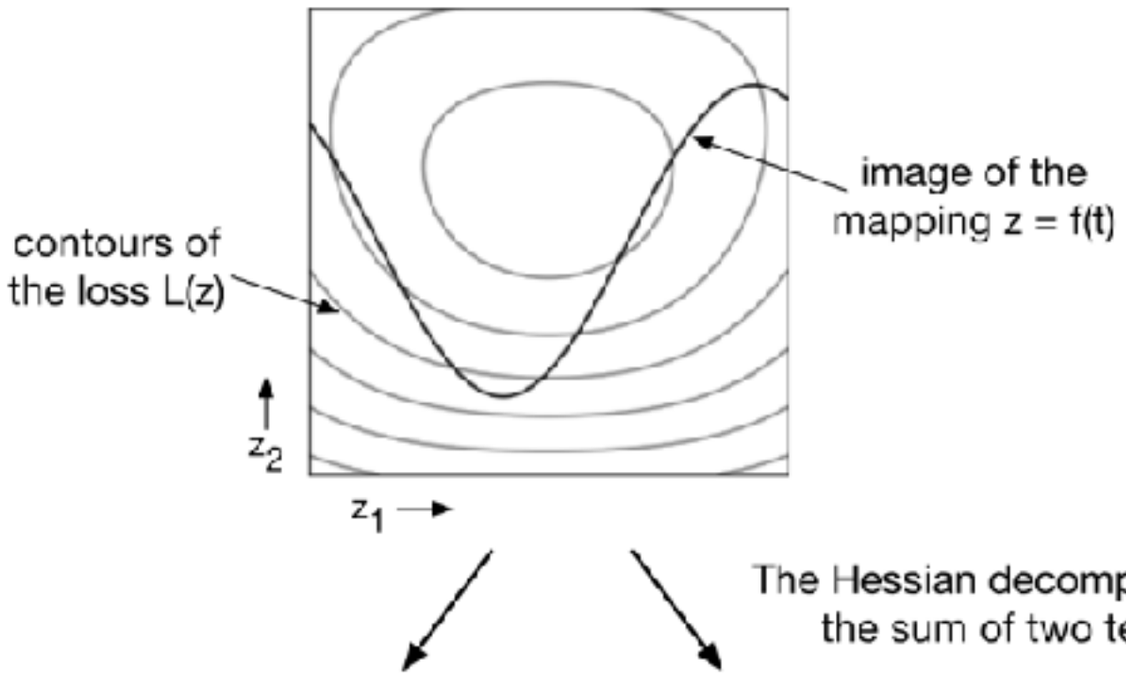
It works very well for convex cost functions (as long as the number of optimization variables isn't too large.)

In a non-convex setting, it looks for critical points, which could be local maxima or saddle points.

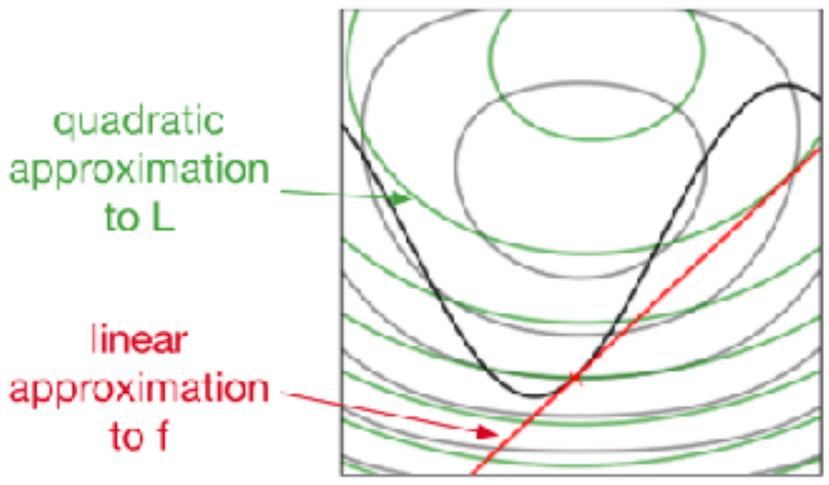
For neural nets, saddle points are common because of symmetries in the weights.



Newton-Rhapson and GGN

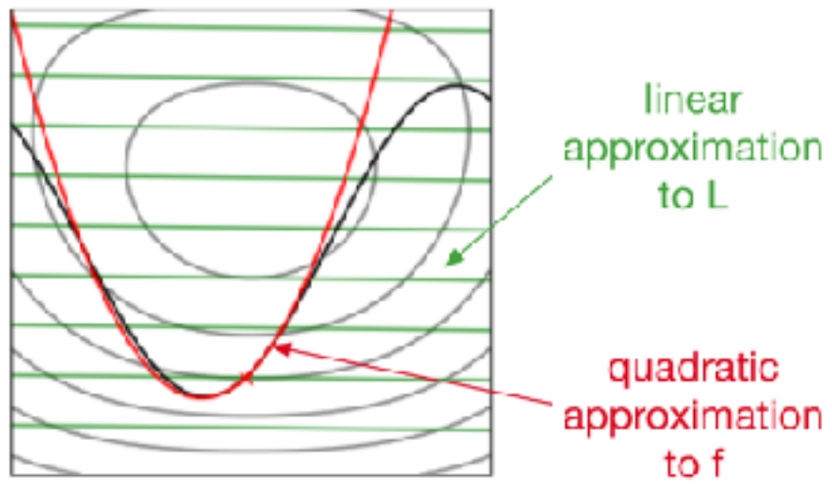


The Hessian decomposes as
the sum of two terms



Term 1: $\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}^\top \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2} \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}$

This is the GGN matrix, which
is always positive semidefinite.

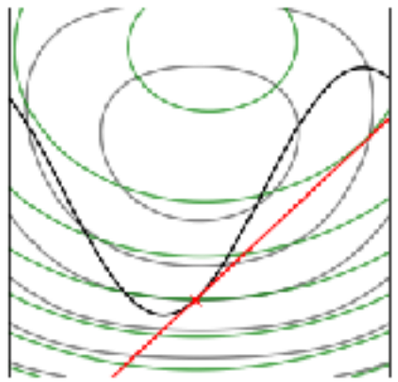


Term 2: $\sum_a \frac{\partial \mathcal{L}}{\partial z_a} \frac{\mathrm{d}^2 z_a}{dt^2}$

This term may have negative
eigenvalues. It vanishes if
z is at the optimum.

Newton-Rhapson and GGN

G is positive semidefinite as long as the loss function L(z) is convex, because it is a linear slice of a convex function.



This means GGN is guaranteed to give a descent direction — a very useful property in non-convex optimization.

$$\begin{aligned}\nabla h(\boldsymbol{\theta})^\top \Delta\boldsymbol{\theta} &= -\alpha \nabla h(\boldsymbol{\theta})^\top \mathbf{G}^{-1} \nabla h(\boldsymbol{\theta}) \\ &\leq 0\end{aligned}$$

The second term of the Hessian vanishes if the prediction errors are very small, in which case G is a good approximation to H. But this might not happen, i.e. if your model can't fit all the training data.

$$\sum_a \frac{\partial \mathcal{L}}{\partial z_a} \frac{\mathrm{d}^2 z_a}{\mathrm{d}\boldsymbol{\theta}^2}$$

vanishes if prediction errors are small

Three optimization algorithms

Newton-Raphson

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{H}^{-1} \nabla h(\boldsymbol{\theta})$$

Hessian matrix

$$\mathbf{H} = \frac{\partial^2 h}{\partial \boldsymbol{\theta}^2}$$

Generalized Gauss-Newton

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{G}^{-1} \nabla h(\boldsymbol{\theta})$$

GGN matrix

$$\mathbf{G} = \mathbb{E}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2} \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}} \right]$$

Natural gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{F}^{-1} \nabla h(\boldsymbol{\theta})$$

Fisher information matrix

$$\mathbf{F} = \mathrm{Cov}\left(\frac{\partial}{\partial \boldsymbol{\theta}} \log p(y|\mathbf{x}) \right)$$

GGN and natural gradient

Rewrite the Fisher matrix:

$$\mathbf{F} = \mathrm{Cov}\left(\frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)$$

$$= \mathbb{E}\left[\frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}^\top\right] - \mathbb{E}\left[\frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right] \mathbb{E}\left[\frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right]^\top$$

= 0 since y is sampled from the model's predictions

Chain rule (backprop):

$$\frac{\partial \log p}{\partial \boldsymbol{\theta}} = \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \log p}{\partial \mathbf{z}}$$

Plugging this in:

$$\mathbb{E}_{\mathbf{x},y}\left[\frac{\partial \log p}{\partial \boldsymbol{\theta}} \frac{\partial \log p}{\partial \boldsymbol{\theta}}^\top\right] = \mathbb{E}_{\mathbf{x},y}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \log p}{\partial \mathbf{z}} \frac{\partial \log p}{\partial \mathbf{z}}^\top \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

$$= \mathbb{E}_{\mathbf{x}}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \mathbb{E}_y\left[\frac{\partial \log p}{\partial \mathbf{z}} \frac{\partial \log p}{\partial \mathbf{z}}^\top\right] \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

GGN and natural gradient

$$\mathbb{E}_{\mathbf{x},y}\left[\frac{\partial \log p}{\partial \boldsymbol{\theta}}\frac{\partial \log p}{\partial \boldsymbol{\theta}}^\top\right] = \mathbb{E}_{\mathbf{x},y}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \log p}{\partial \mathbf{z}}\frac{\partial \log p}{\partial \mathbf{z}}^\top \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

$$= \mathbb{E}_{\mathbf{x}}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \underbrace{\mathbb{E}_{y}\left[\frac{\partial \log p}{\partial \mathbf{z}}\frac{\partial \log p}{\partial \mathbf{z}}^\top\right]}_{\text{Fisher matrix w.r.t. the output layer}} \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

If the loss function L is negative log-likelihood for an exponential family and the network's outputs are the natural parameters, then the Fisher matrix in the top layer is the same as the Hessian.

Examples: softmax-cross-entropy, squared error (i.e. Gaussian)

In this case, this expression reduces to the GGN matrix:

$$\mathbf{G} = \mathbb{E}_{\mathbf{x}}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 L}{\partial \mathbf{z}^2}\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}\right]$$

Three optimization algorithms

So all three algorithms are related! This is why we call natural gradient a "second-order optimizer."

Newton-Raphson

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{H}^{-1} \nabla h(\boldsymbol{\theta})$$

Hessian matrix

$$\mathbf{H} = \frac{\partial^2 h}{\partial \boldsymbol{\theta}^2}$$

Generalized Gauss-Newton

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{G}^{-1} \nabla h(\boldsymbol{\theta})$$

GGN matrix

$$\mathbf{G} = \mathbb{E}\left[\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2} \frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}} \right]$$

Natural gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{F}^{-1} \nabla h(\boldsymbol{\theta})$$

Fisher information matrix

$$\mathbf{F} = \mathrm{Cov}\left(\frac{\partial}{\partial \boldsymbol{\theta}} \log p(y|\mathbf{x}) \right)$$

Background: natural gradient (Amari, 1998)

Problem: dimension of **F** is the number of trainable parameters

Modern networks can have **tens of millions of parameters**!

e.g. weight matrix between two 1000-unit layers has
1000 x 1000 = 1 million parameters

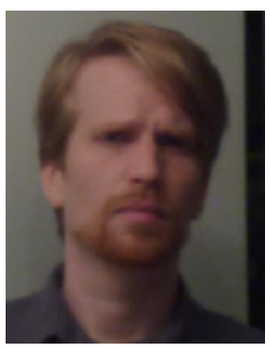
Cannot store a dense 1 million x 1 million matrix, let alone compute $\mathbf{F}^{-1}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$

Background: approximate second-order training

- diagonal methods
  - e.g. Adagrad, RMSProp, Adam
  - very little overhead, but sometimes not much better than SGD
- iterative methods
  - e.g. Hessian-Free optimization (Martens, 2010); Byrd et al. (2011); TRPO (Schulman et al., 2015)
  - may require many iterations for each weight update
  - only uses metric/curvature information from a single batch
- subspace-based methods
  - e.g. Krylov subspace descent (Vinyals and Povey 2011); sum-of-functions (Sohl-Dickstein et al., 2014)
  - can be memory intensive

Optimizing neural networks using Kronecker-factored approximate curvature

A Kronecker-factored Fisher matrix for convolution layers



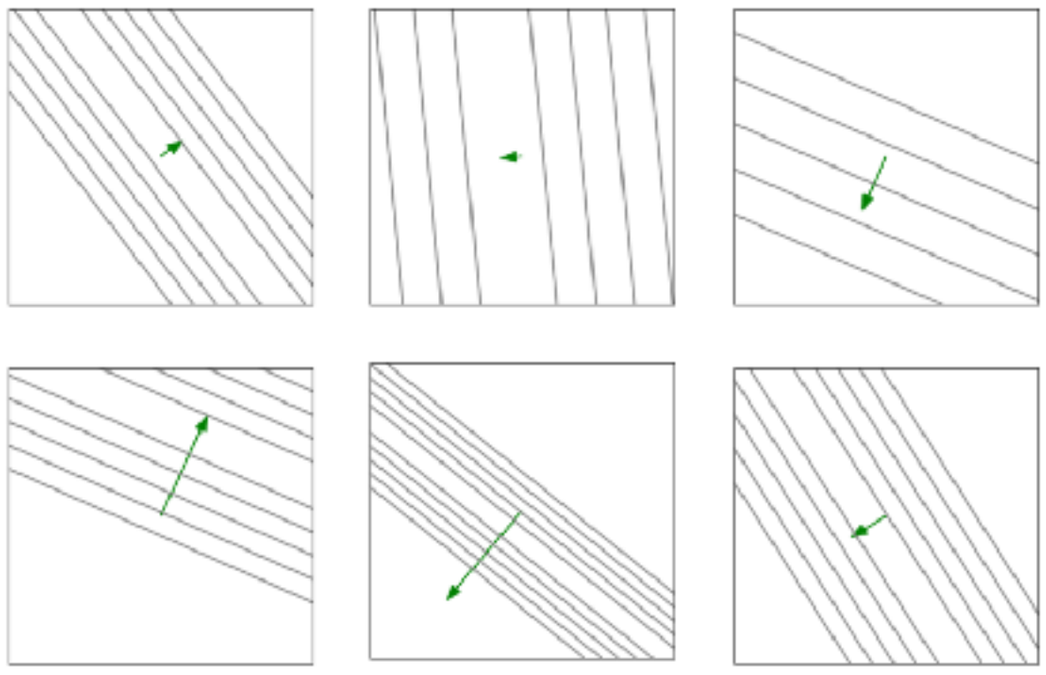
James Martens

Probabilistic models of the gradient computation

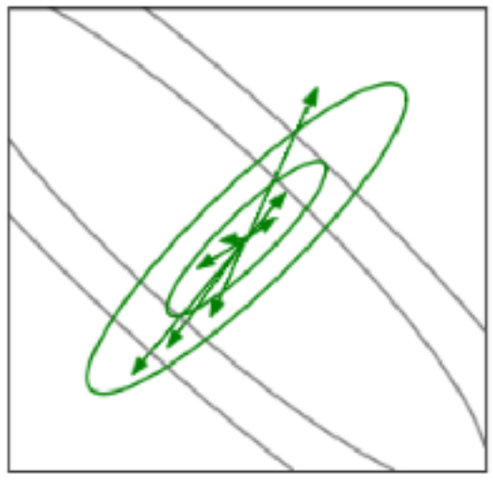
Recall: $\mathbf{F}$ is the covariance matrix of the log-likelihood gradient

$$\mathbf{F}_{\boldsymbol{\theta}} = \operatorname*{Cov}_{\substack{\mathbf{x}\sim p_{\text{data}} \\ \mathbf{y}\sim r_{\boldsymbol{\theta}}(\mathbf{y}\,|\,\mathbf{x})}} \left(\nabla_{\boldsymbol{\theta}} \log r_{\boldsymbol{\theta}}(\mathbf{y}\,|\,\mathbf{x})\right)$$

Samples from this distribution for a regression problem:



Log-likelihood contours and
gradients for data points
sampled from the model's predictions



Average log-likelihood contour
and distribution of gradients

Probabilistic models of the gradient computation

Recall that $\mathbf{F}$ may be 1 million x 1 million or larger

Want a probabilistic model such that:

the distribution can be **compactly represented**

$\mathbf{F}^{-1}$ can be **efficiently computed**

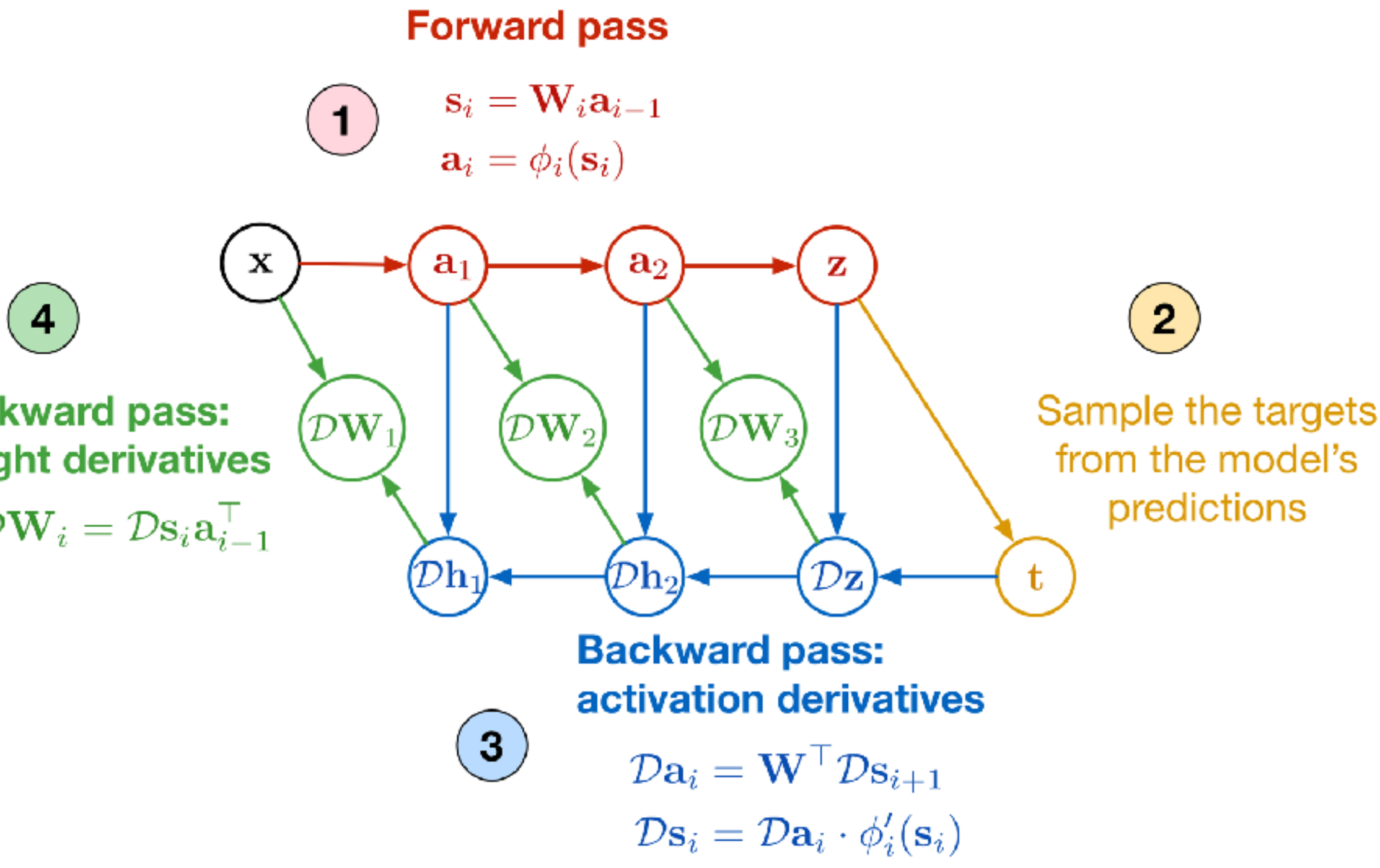
Strategy: impose conditional independence structure based on:

structure of the computation graph

empirical observations

Can make use of what we know about probabilistic graphical models!

Natural gradient for classification networks



Natural gradient for classification networks

Forward pass:

$$\mathbf{s}_\ell = \mathbf{W}_\ell \mathbf{h}_{\ell-1} + \mathbf{b}_\ell$$
$$\mathbf{h}_\ell = \phi(\mathbf{s}_\ell)$$

Backward pass:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} = \mathbf{W}_\ell^\top \frac{\partial \mathcal{L}}{\partial \mathbf{s}_{\ell+1}}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{s}_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} \circ \phi'(\mathbf{s}_\ell)$$

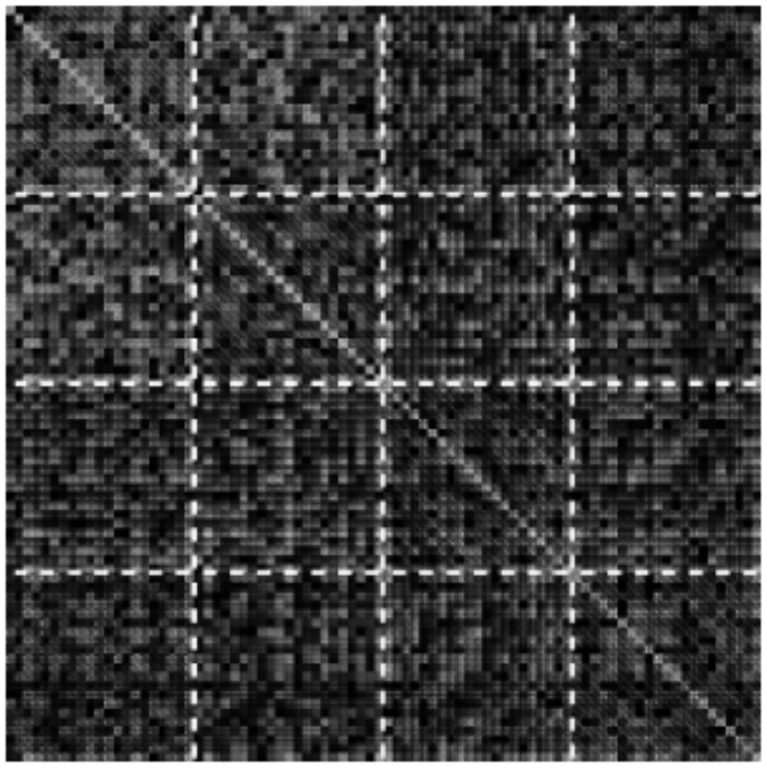
Approximate with a linear-Gaussian model:

$$\mathbf{h}_\ell = \mathbf{A}_\ell \mathbf{h}_{\ell-1} + \mathbf{B}_\ell \boldsymbol{\varepsilon}$$
$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

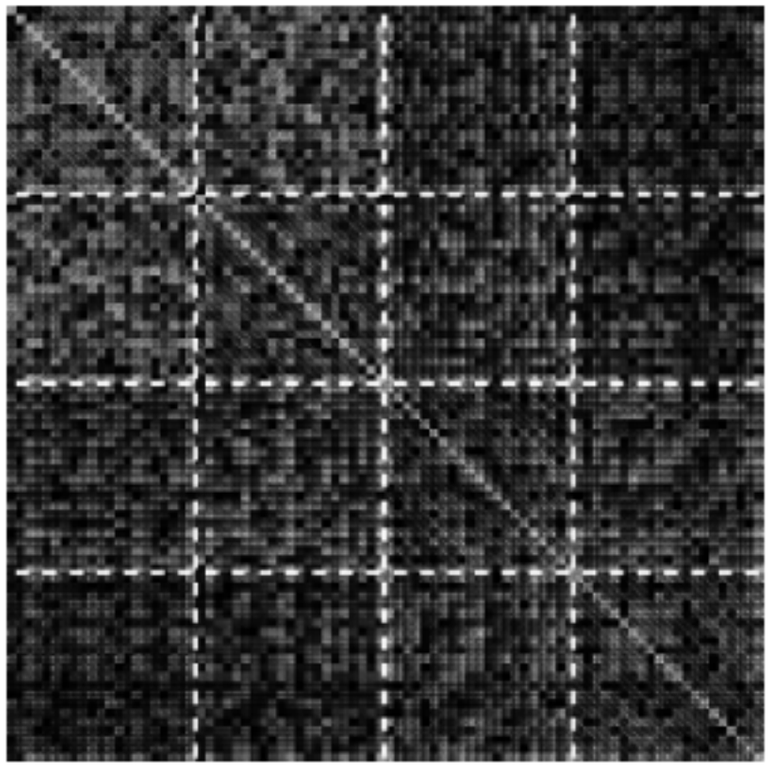
$$\frac{\partial \mathcal{L}}{\partial \mathbf{s}_\ell} = \mathbf{C}_\ell \frac{\partial \mathcal{L}}{\partial \mathbf{s}_{\ell+1}} + \mathbf{D}_\ell \boldsymbol{\varepsilon}$$
$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

Kronecker-Factored Approximate Curvature (K-FAC)

Quality of approximate Fisher matrix on a very small network:



exact



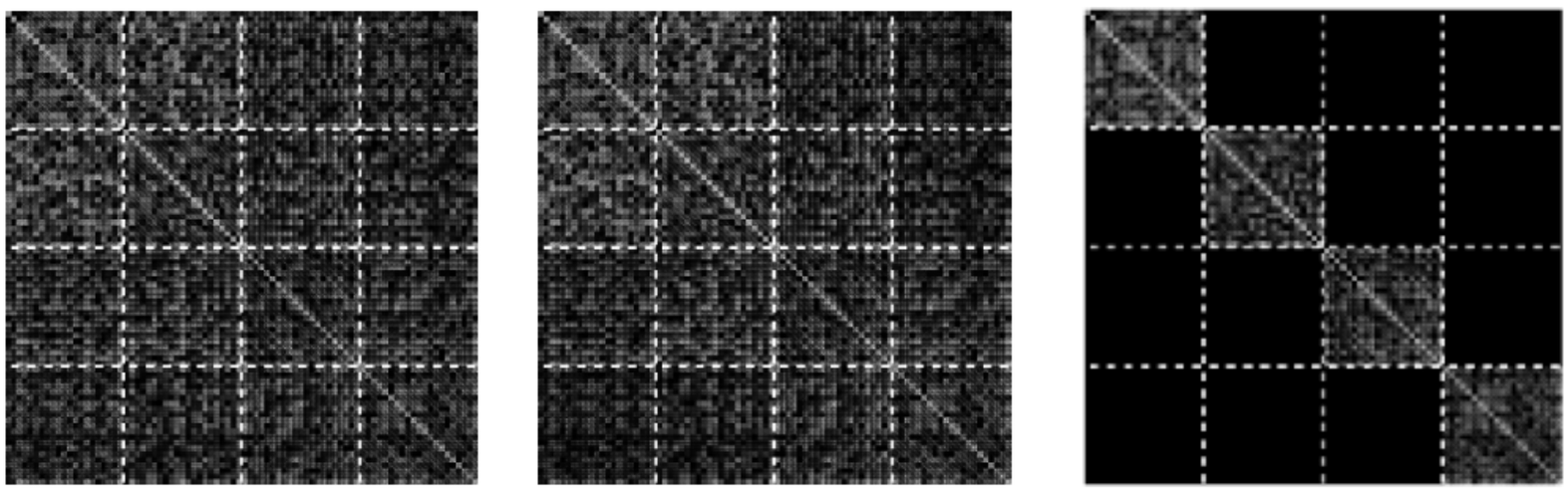
approximation

Kronecker-Factored Approximate Curvature (K-FAC)

Assume a **fully connected network**

Impose probabilistic modeling assumptions:

- **dependencies between different layers** of the network
  - Option 1: chain graphical model. Principled, but complicated.
  - Option 2: full independence between layers. Simple to implement, and works almost as well in practice.
- **activations and activation gradients are independent**
  - we can show they are uncorrelated. Note: this depends on the activations being sampled from the model's predictions.



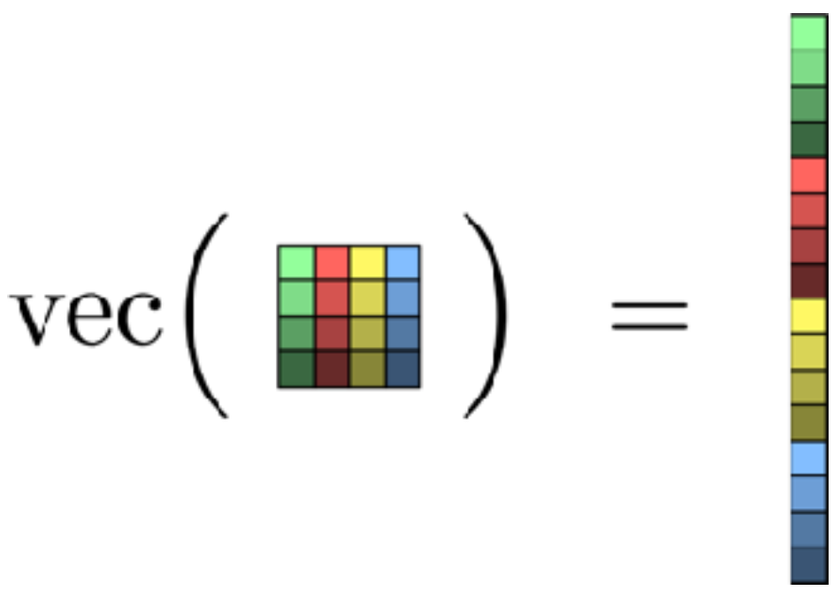
exact block tridiagonal block diagonal

Kronecker products

Kronecker product:

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

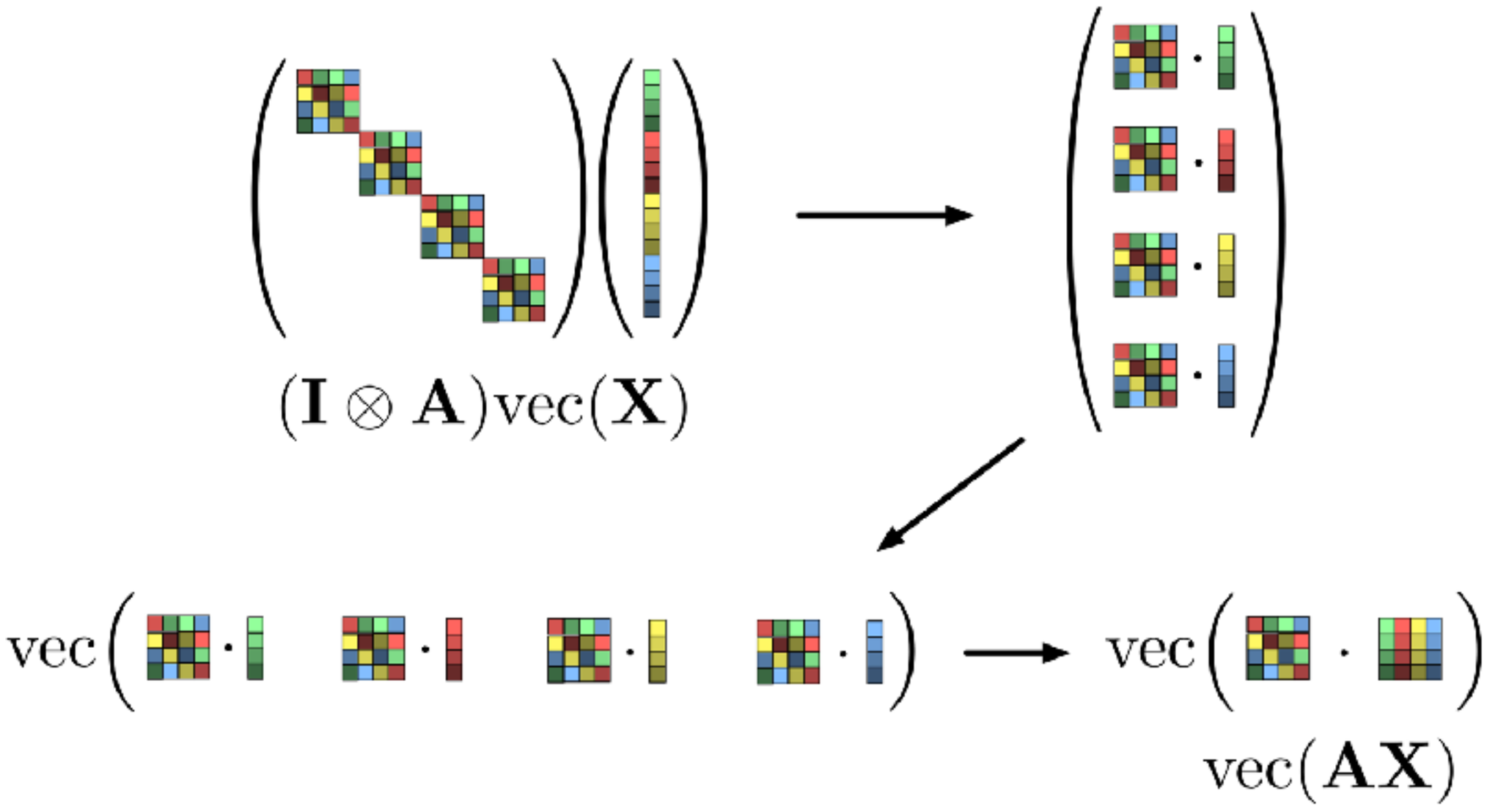
vec operator:



Kronecker products

Matrix multiplication is a linear operation, so we should be able to write it as a matrix-vector product.

Kronecker products let us do this.



Kronecker products

The more general identity:

$$(A \otimes B)\mathrm{vec}(X) = \mathrm{vec}(BXA^\top)$$

Other convenient identities:

$$(A \otimes B)^\top = A^\top \otimes B^\top$$

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

Justification:

$$\begin{aligned}(A^{-1} \otimes B^{-1})(A \otimes B)\mathrm{vec}(X) &= (A^{-1} \otimes B^{-1})\mathrm{vec}(BXA^\top) \\ &= \mathrm{vec}(B^{-1}BXA^\top A^{-\top}) \\ &= \mathrm{vec}(X)\end{aligned}$$

Kronecker-Factored Approximate Curvature (K-FAC)

Entries of the Fisher matrix for one layer of a multilayer perceptron:

$$
\begin{aligned}
F_{(i,j),(i',j')} &= \mathbb{E}\left[\frac{\partial \mathcal{L}}{\partial w_{ij}} \frac{\partial \mathcal{L}}{\partial w_{i'j'}}\right] \\
&= \mathbb{E}\left[a_j \frac{\partial \mathcal{L}}{\partial s_i} a_{j'} \frac{\partial \mathcal{L}}{\partial s_{i'}}\right] \\
&= \mathbb{E}\left[a_j a_{j'}\right] \mathbb{E}\left[\frac{\partial \mathcal{L}}{\partial s_i} \frac{\partial \mathcal{L}}{\partial s_{i'}}\right]
\end{aligned}
$$

under the approximation that activations and derivatives are independent

In vectorized form:

$$
\begin{aligned}
\mathbf{F} &= \mathbf{\Omega} \otimes \mathbf{\Gamma} \\
\mathbf{\Omega} &= \mathrm{Cov}(\mathbf{a}) \\
\mathbf{\Gamma} &= \mathrm{Cov}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{s}}\right)
\end{aligned}
$$

Kronecker-Factored Approximate Curvature (K-FAC)

Under the approximation that layers are independent,

$$\hat{\mathbf{F}} = \begin{pmatrix} \boldsymbol{\Psi}_0 \otimes \boldsymbol{\Gamma}_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \boldsymbol{\Psi}_{L-1} \otimes \boldsymbol{\Gamma}_L \end{pmatrix}$$

$\boldsymbol{\Psi}$ and $\boldsymbol{\Gamma}$ represent covariance statistics that are estimated during training.

Efficient computation of the approximate natural gradient:

$$\hat{\mathbf{F}}^{-1}\nabla h = \begin{pmatrix} \operatorname{vec}\left(\boldsymbol{\Gamma}_1^{-1}(\nabla_{\bar{\mathbf{W}}_1} h)\boldsymbol{\Psi}_0^{-1}\right) \\ \vdots \\ \operatorname{vec}\left(\boldsymbol{\Gamma}_L^{-1}(\nabla_{\bar{\mathbf{W}}_L} h)\boldsymbol{\Psi}_{L-1}^{-1}\right) \end{pmatrix}$$

Representation is comparable in size to the number of weights!

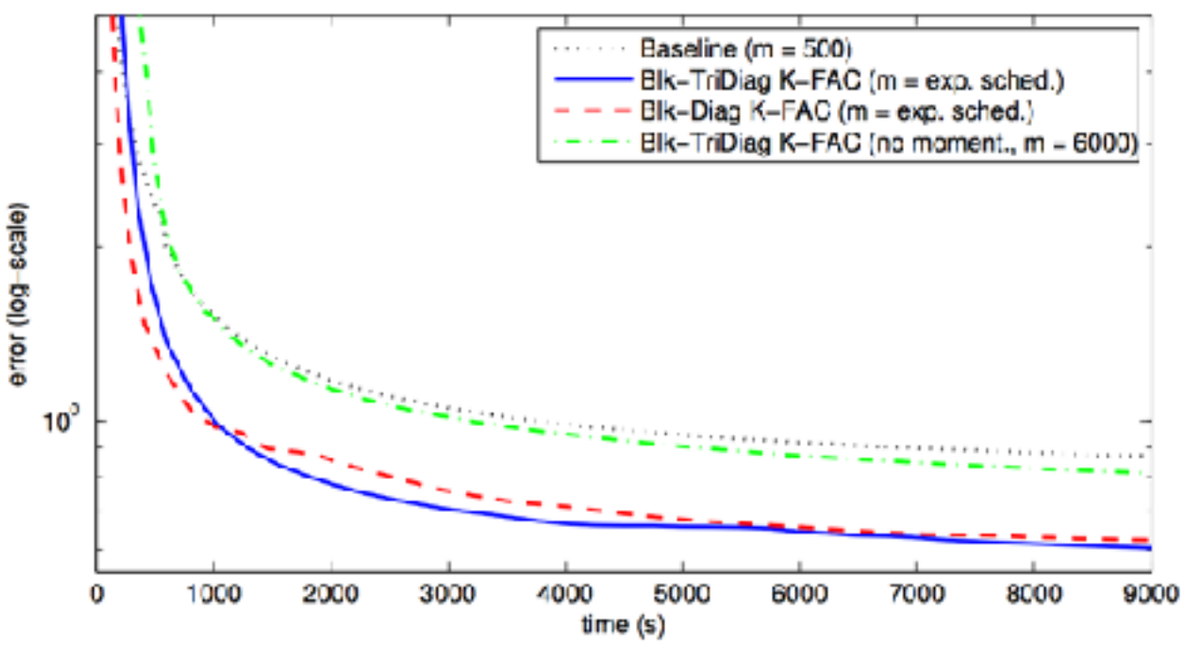
Only involves operations on matrices approximately the size of **W**

Small constant factor overhead (1.5x) compared with SGD

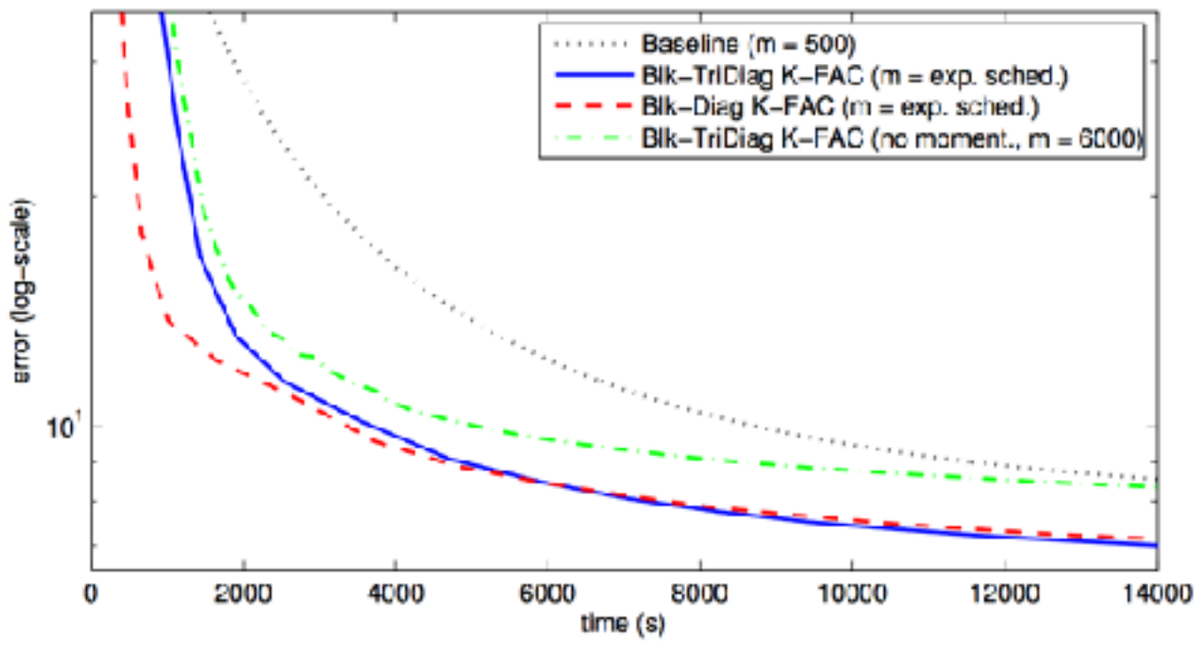
Experiments

Deep autoencoders (wall clock)

MNIST



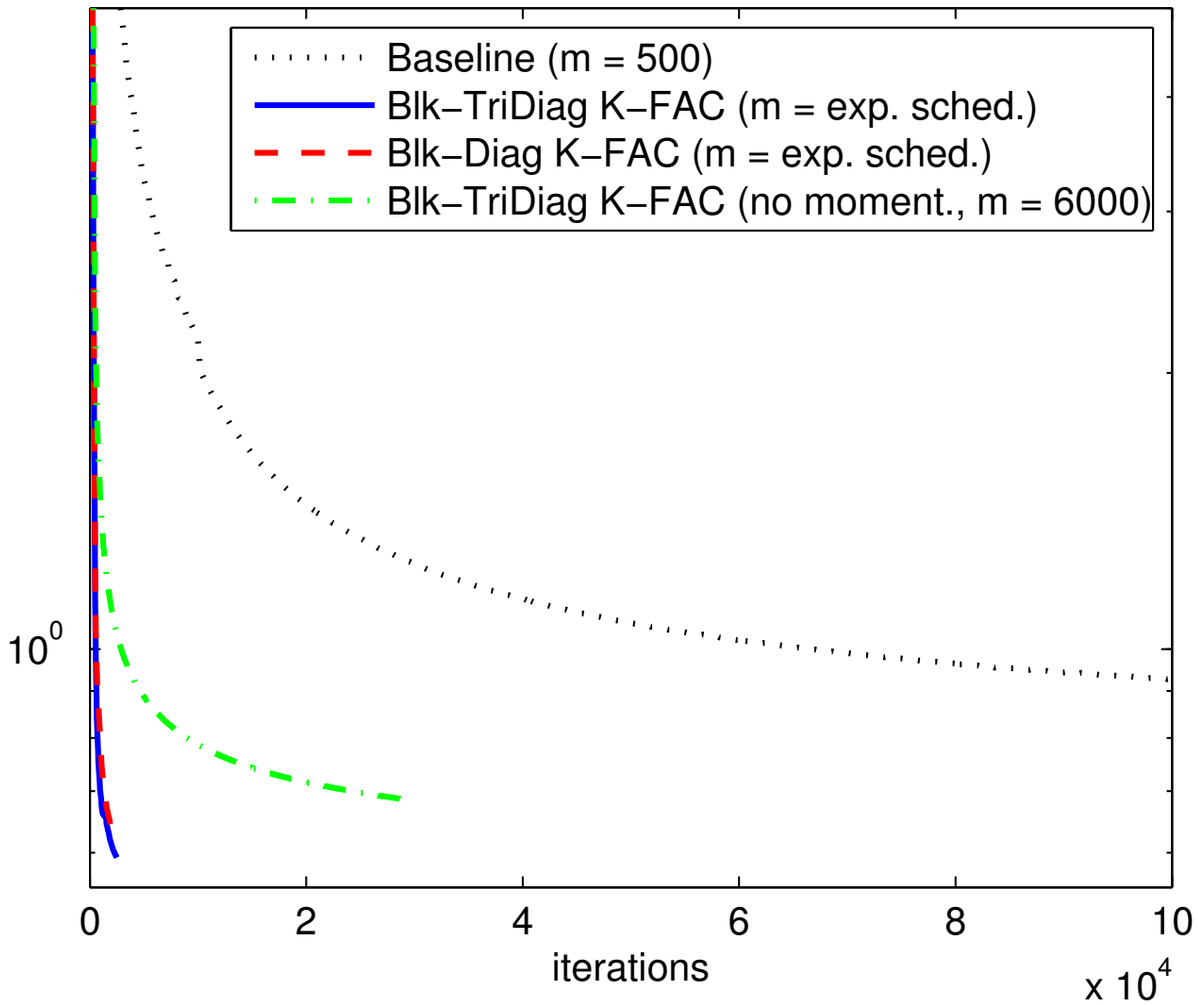
faces



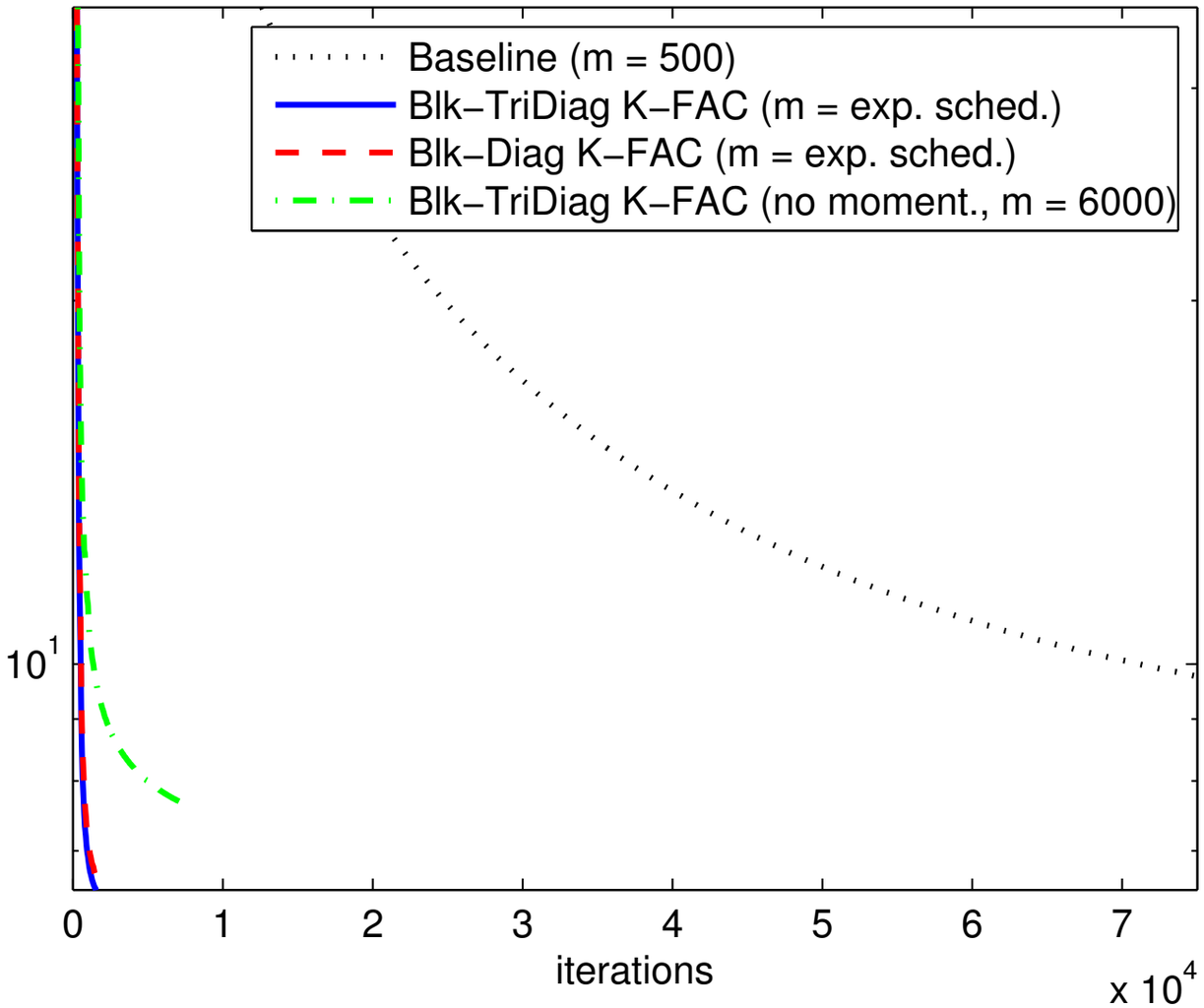
Experiments

Deep autoencoders (iterations)

MNIST



faces



Kronecker Factors for Convolution (KFC)

Can we extend this to convolutional networks?

Types of layers in conv nets:

Fully connected: already covered by K-FAC

Pooling: no parameters, so we don't need to worry about them

Normalization: few parameters; can fit a full covariance matrix

Convolution: this is what I'll focus on!

$$s_{i,t} = \sum_{\delta} w_{i,j,\delta} a_{j,t+\delta} + b_i,$$

$$a'_{i,t} = \phi(s_{i,t})$$

Kronecker Factors for Convolution (KFC)

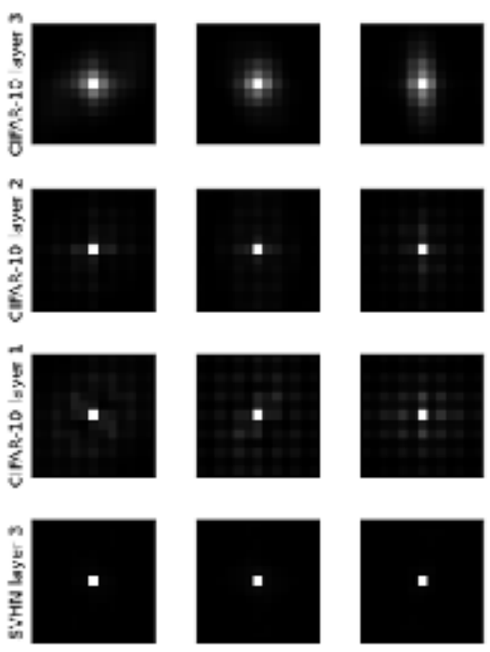
For tractability, we must make some **modeling assumptions**:

- activations and derivatives are independent (or jointly Gaussian)
- no between-layer correlations
- spatial homogeneity
  - implicitly assumed by conv nets
- spatially uncorrelated derivatives

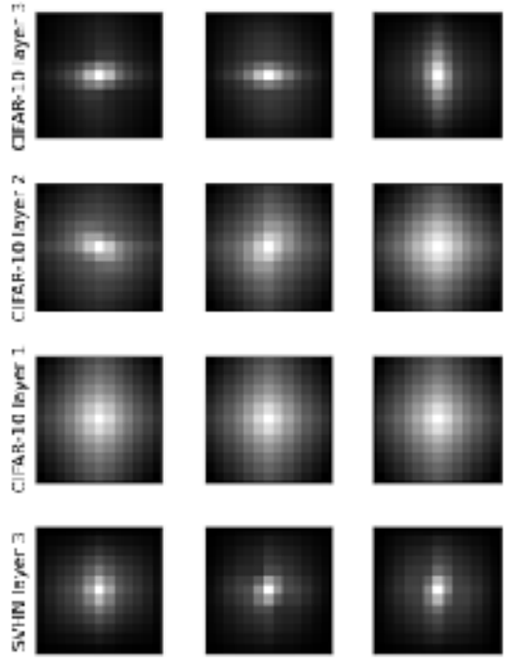
Under these assumptions, we derive the same Kronecker-factorized approximation and update rules as in the fully connected case.

Kronecker Factors for Convolution (KFC)

Are the error derivatives actually spatially uncorrelated?



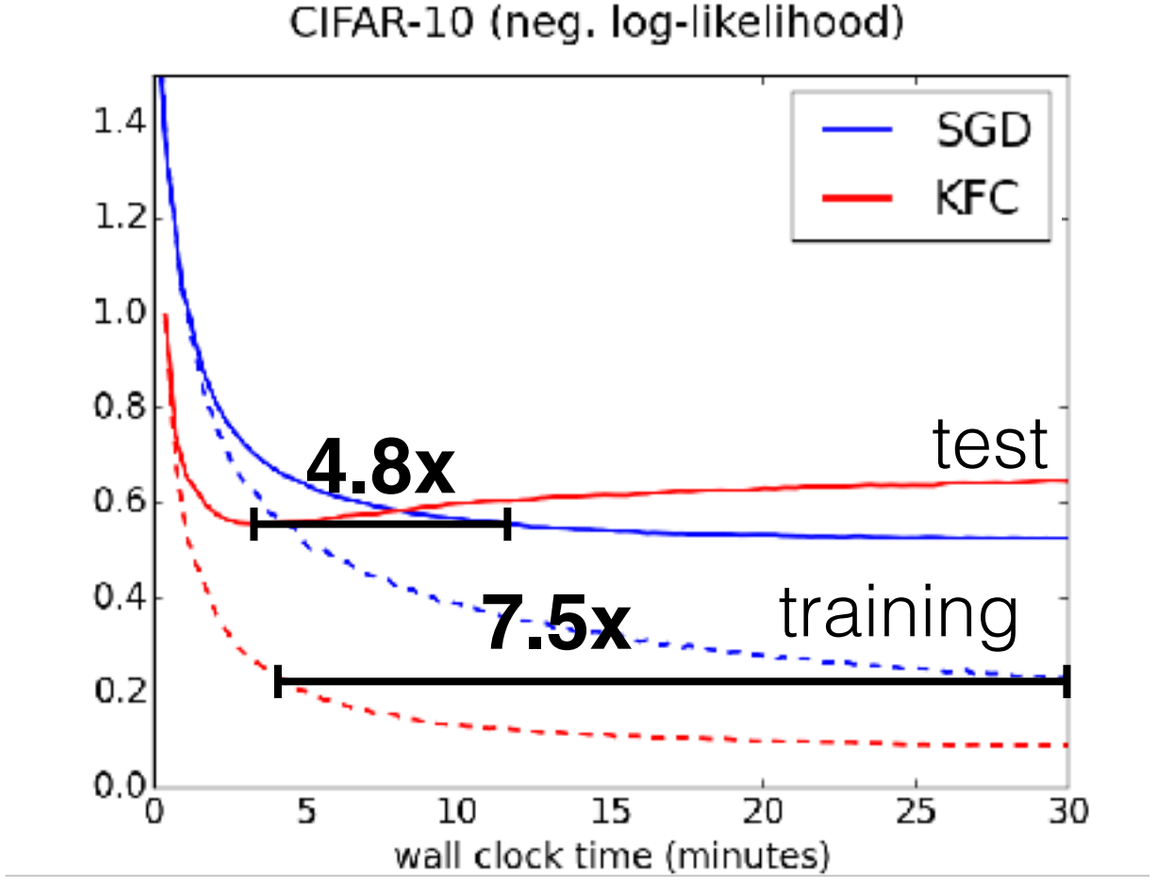
Spatial autocorrelations
of error derivatives

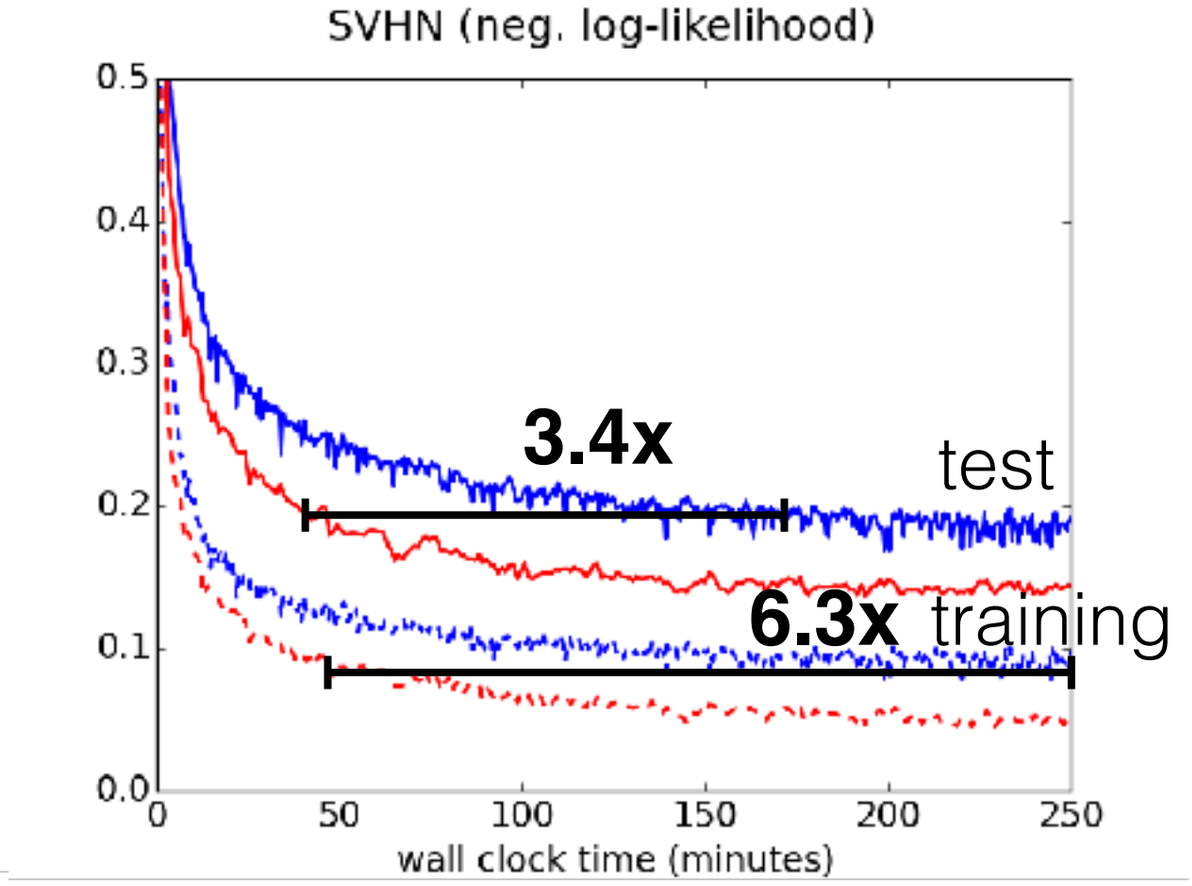


Spatial autocorrelations
of activations

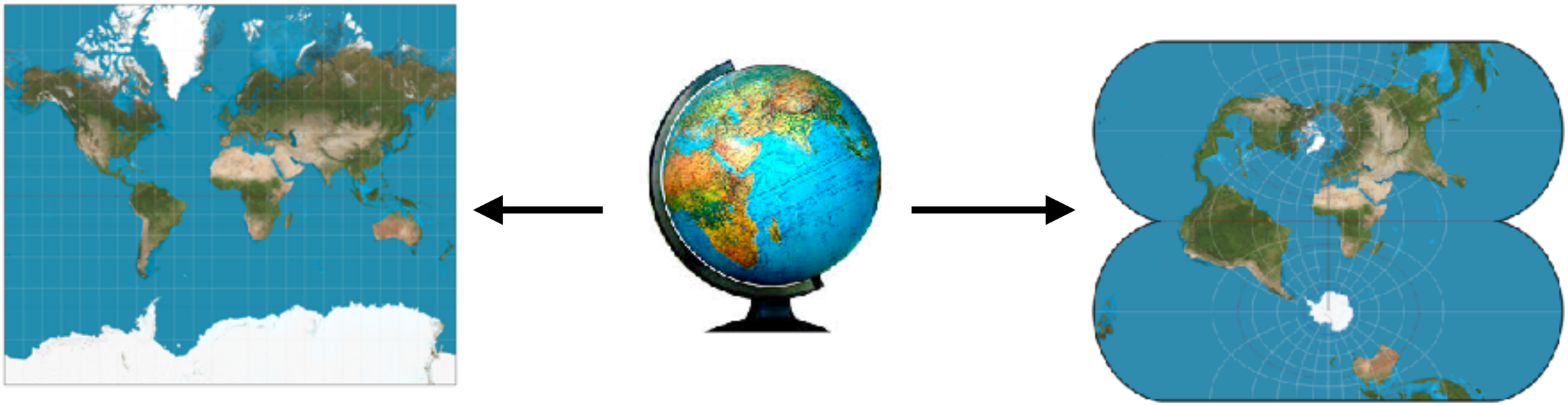
Experiments

conv nets (wall clock)





Invariance to reparameterization



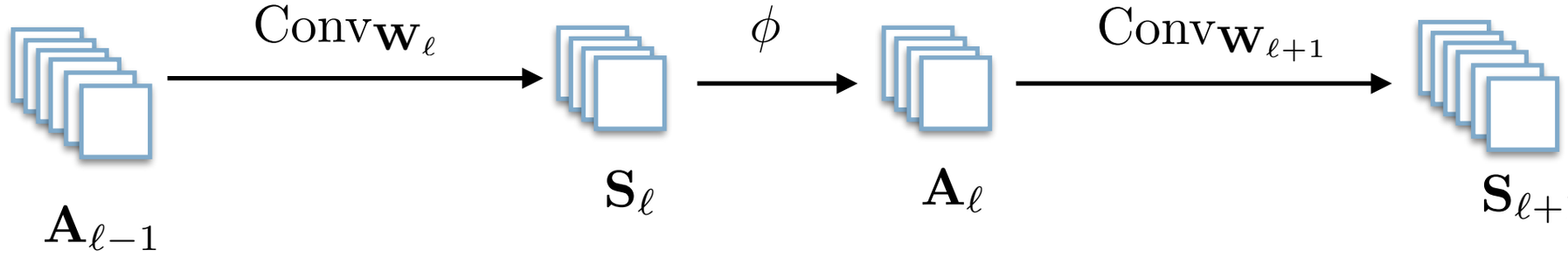
One justification of (exact) natural gradient descent
is that it's invariant to reparameterization

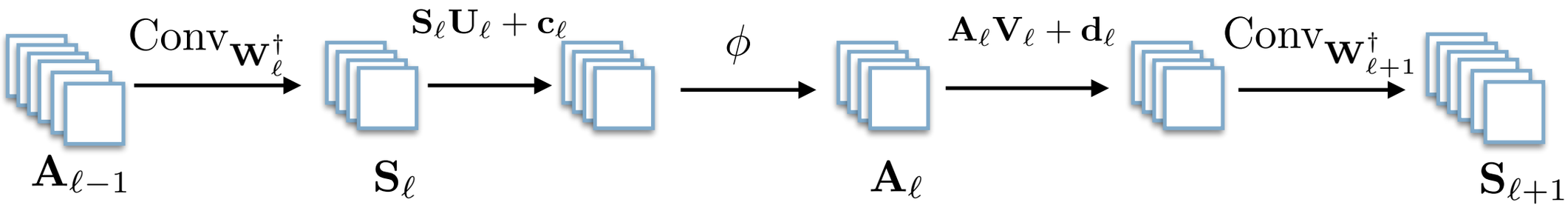
Can analyze approximate natural gradient in terms of invariance to
restricted classes of reparameterizations

Invariance to reparameterization

KFC is invariant to **homogeneous pointwise affine transformations** of the activations.

I.e., consider the following equivalent networks with different parameterizations:



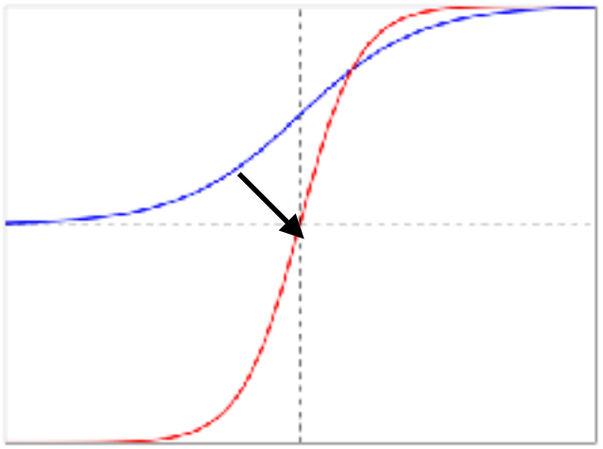


After an SGD update, the networks compute different functions

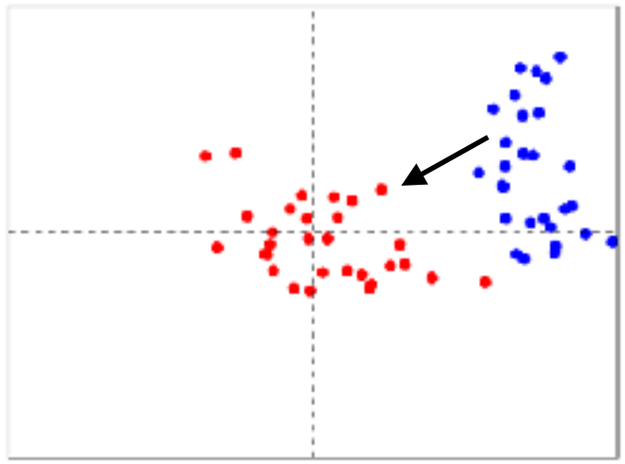
After a KFC update, they still compute the same function

Invariance to reparameterization

KFC preconditioning is invariant to **homogeneous pointwise affine transformations** of the activations. This includes:



Replacing logistic nonlinearity with tanh

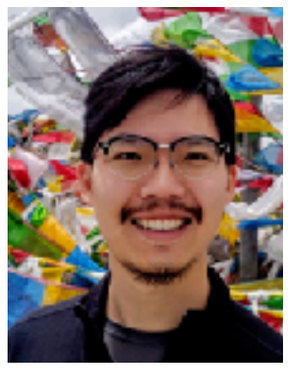


Centering activations to zero mean, unit variance

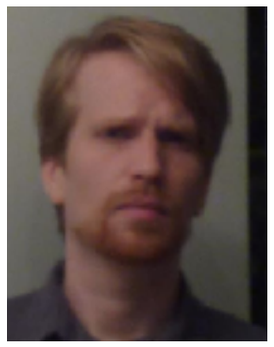
Whitening the images in color space

New interpretation: K-FAC is doing **exact natural gradient on a different metric**. The invariance properties follow almost immediately from this fact. (coming soon on arXiv)

Distributed second-order optimization using Kronecker-factored approximations



Jimmy Ba



James Martens

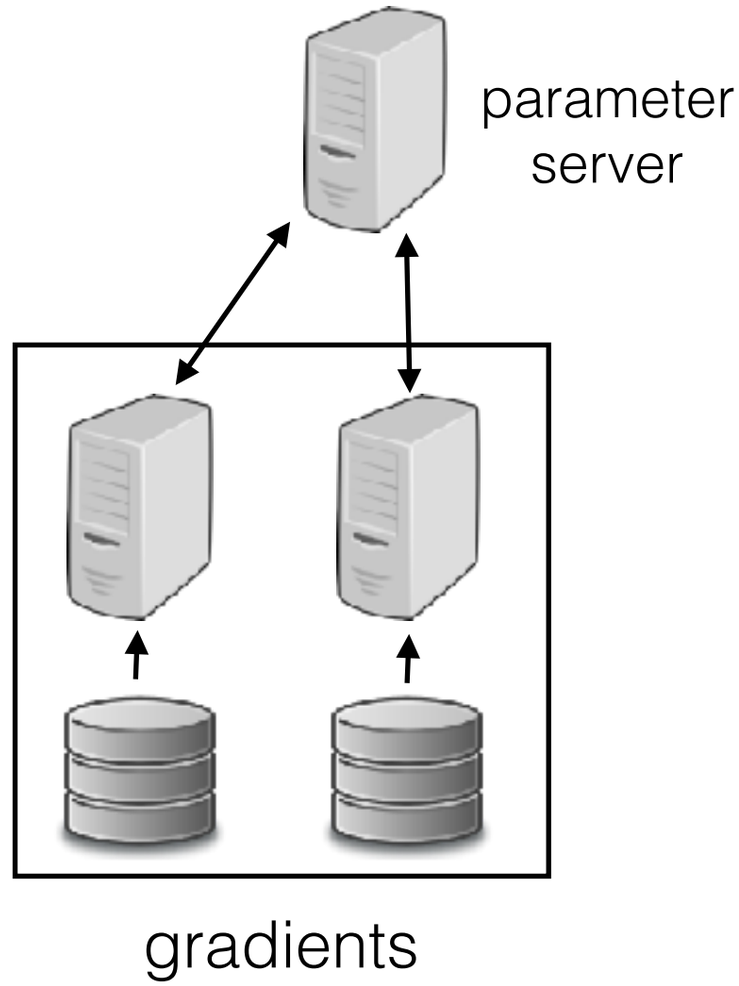
Background: distributed SGD

Suppose you have a cluster of GPUs. How can you use this to speed up training?

One common solution is synchronous stochastic gradient descent: have a bunch of worker nodes computing gradients on different subsets of the data.

This lets you efficiently compute SGD updates on large mini-batches, which reduces the variance of the updates.

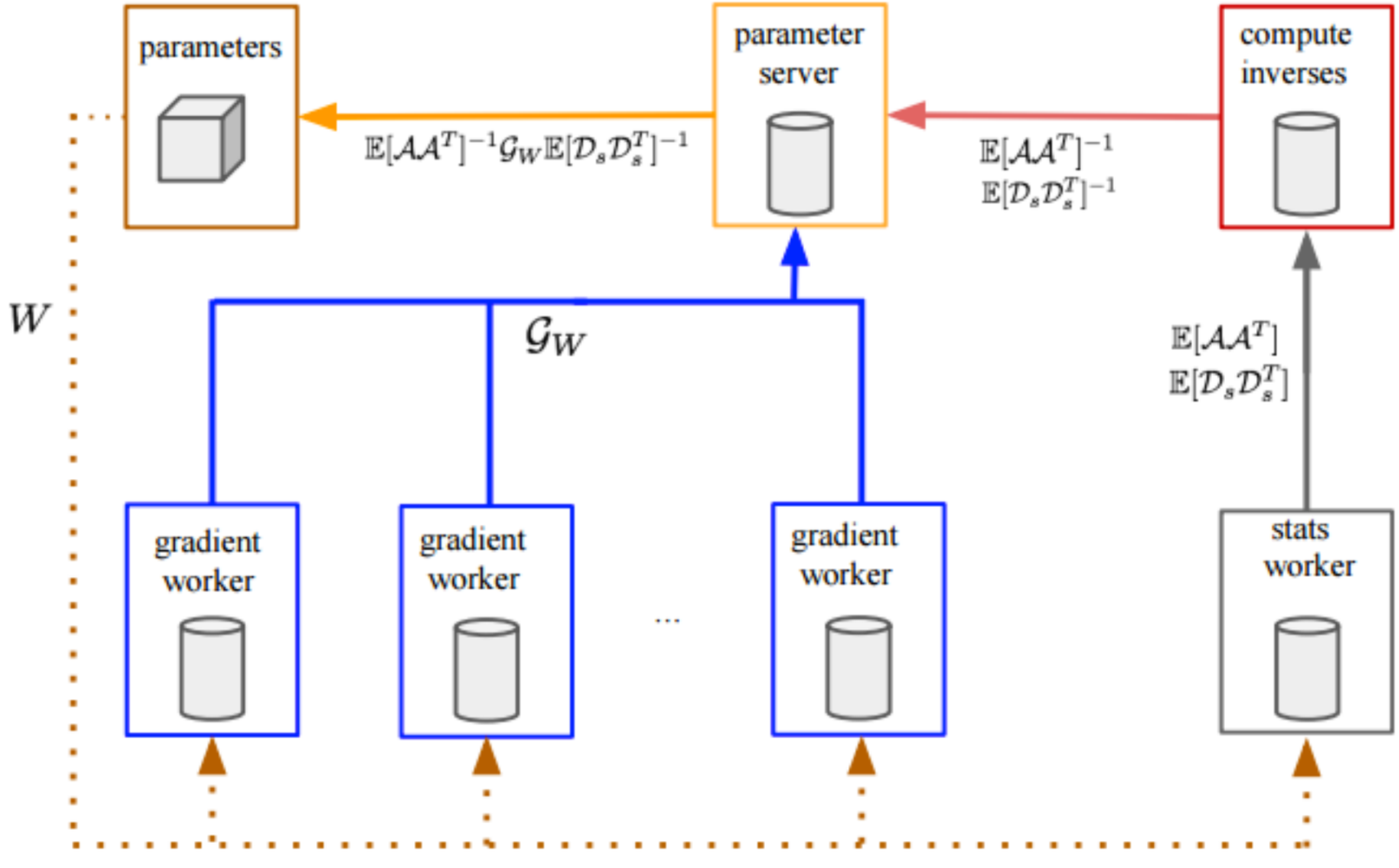
But you quickly get diminishing returns as you add more workers, because curvature, rather than stochasticity, becomes the bottleneck.



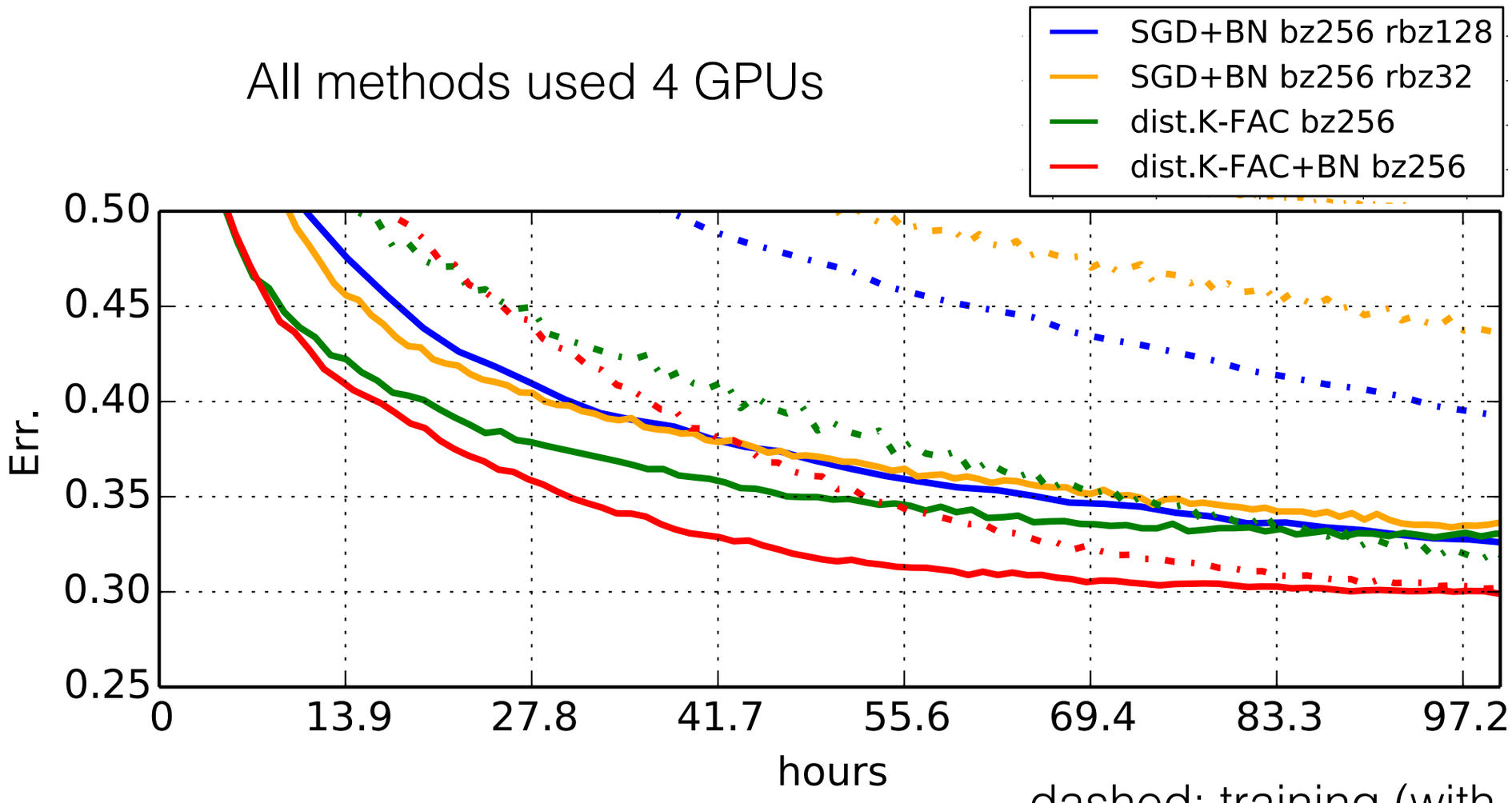
Distributed K-FAC

Because K-FAC accounts for curvature information, it ought to scale to a higher degree of parallelism, and continue to benefit from reduced variance updates.

We base our method off of synchronous SGD, and perform K-FAC's additional computations on separate nodes.



Training GoogLeNet on ImageNet

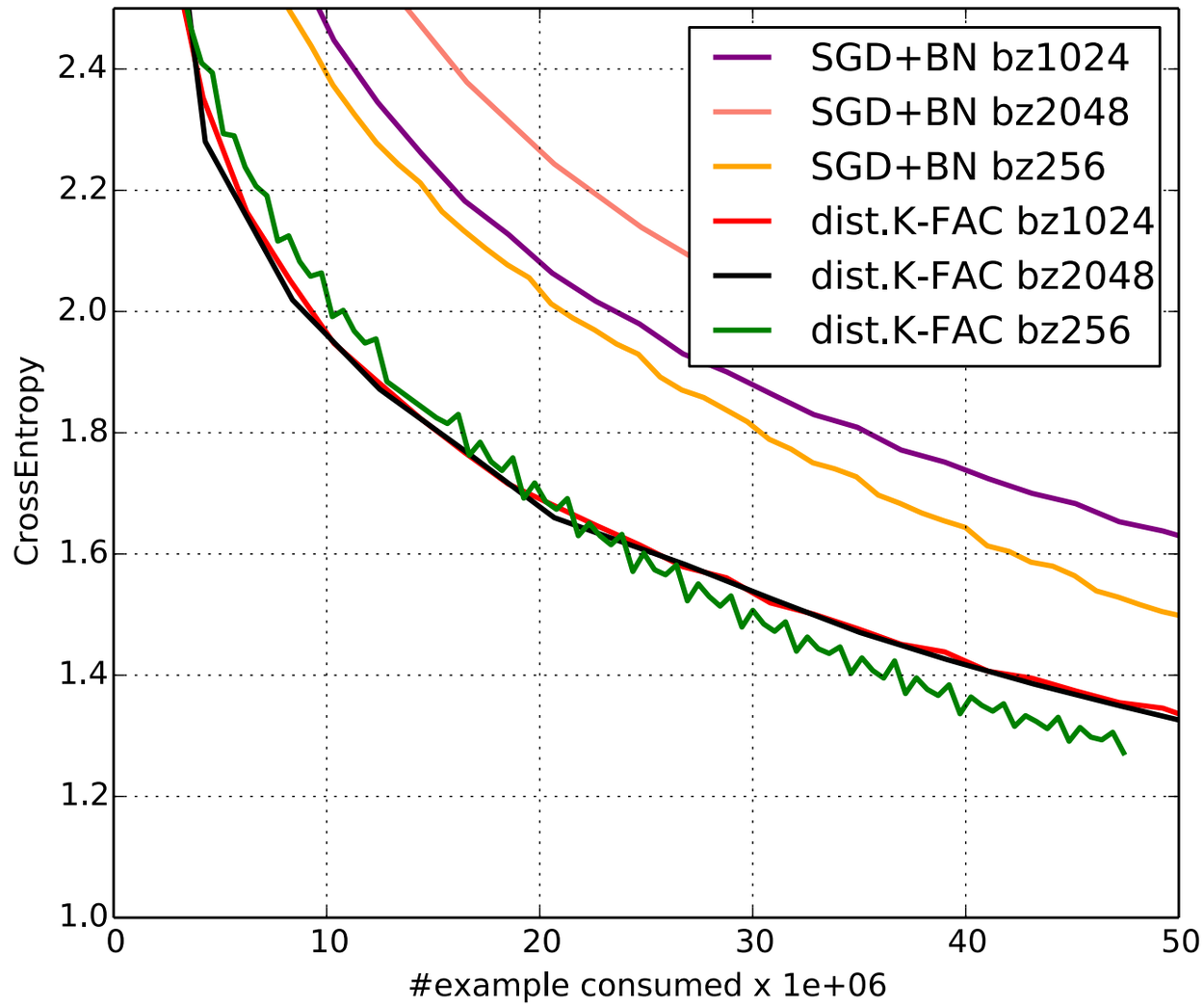


dashed: training (with distortions)
solid: test

Similar results on AlexNet, VGGNet, ResNet

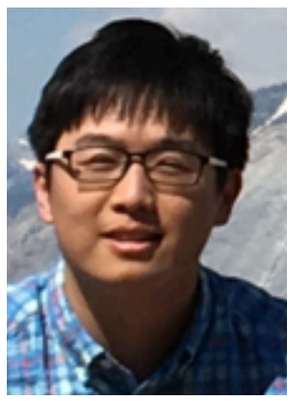
Scaling with mini-batch size

GoogLeNet Performance as a function of # examples:



This suggests distributed K-FAC can be scaled to a higher degree of parallelism.

Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation



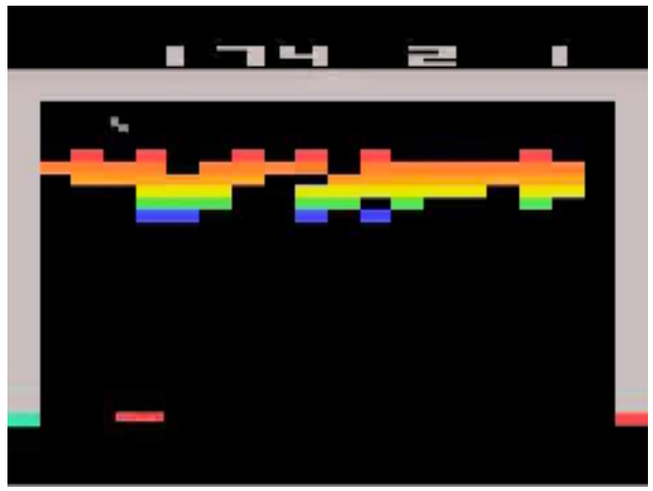
Yuhuai Wu

Elman Mansimov

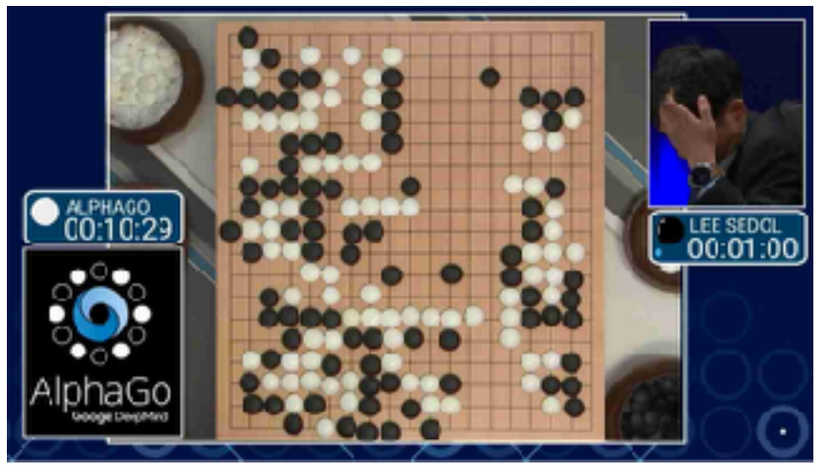
Jimmy Ba

Reinforcement Learning

Neural networks have recently seen key successes in reinforcement learning (i.e. deep RL)



human-level Atari
(Mnih et al., 2015)



AlphaGo
(Silver et al., 2016)

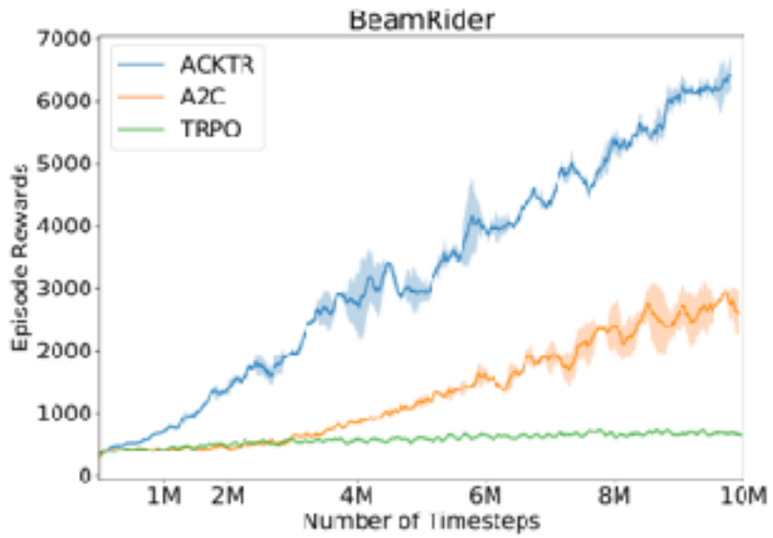
Most of these networks are still being trained using SGD-like procedures. Can we apply second-order optimization?

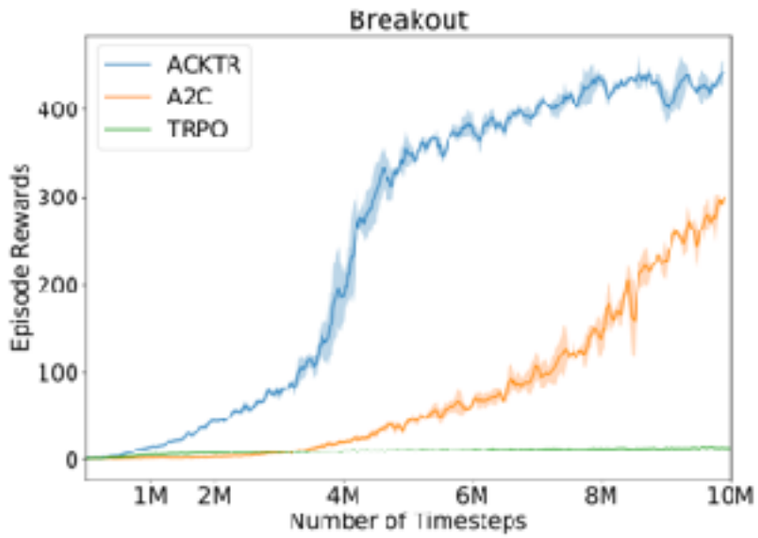
Reinforcement Learning

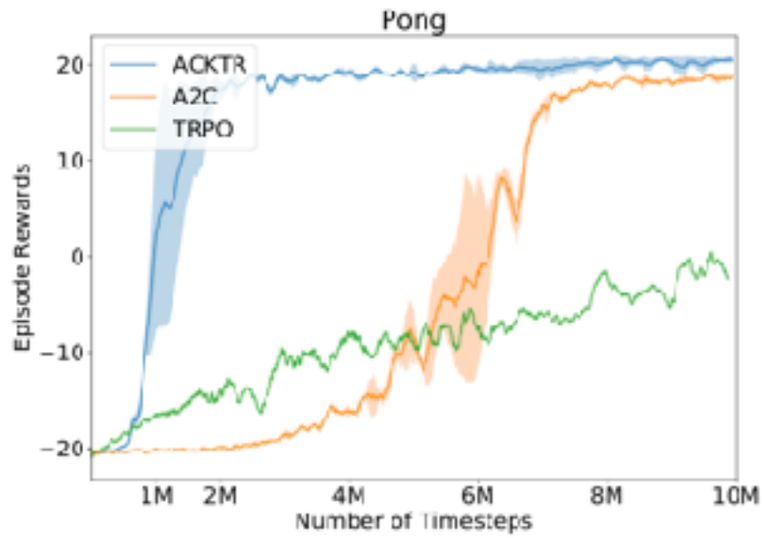
- We'd like to achieve sample efficient RL without sacrificing computational efficiency.
- TRPO approximates the natural gradient using conjugate gradient, similarly to Hessian-free optimization
  - very efficient in terms of the number of parameter updates
  - but requires an expensive iterative procedure for each update
  - only uses curvature information from the current batch
- applying K-FAC to advantage actor critic (A2C)
  - Fisher metric for actor network (same as prior work)
  - Gauss-Newton metric for critic network (i.e. Euclidean metric on values)
  - re-scale updates using trust region method, analogously to TRPO
    - approximate the KL using the Fisher metric

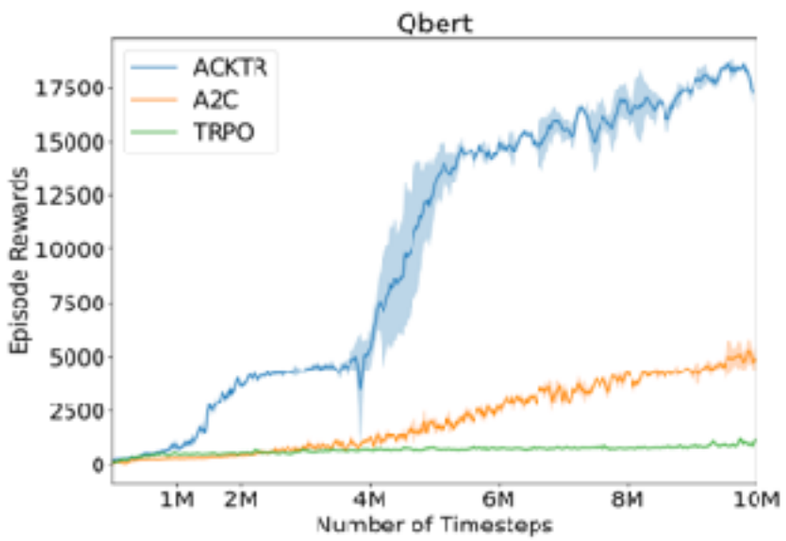
Reinforcement Learning

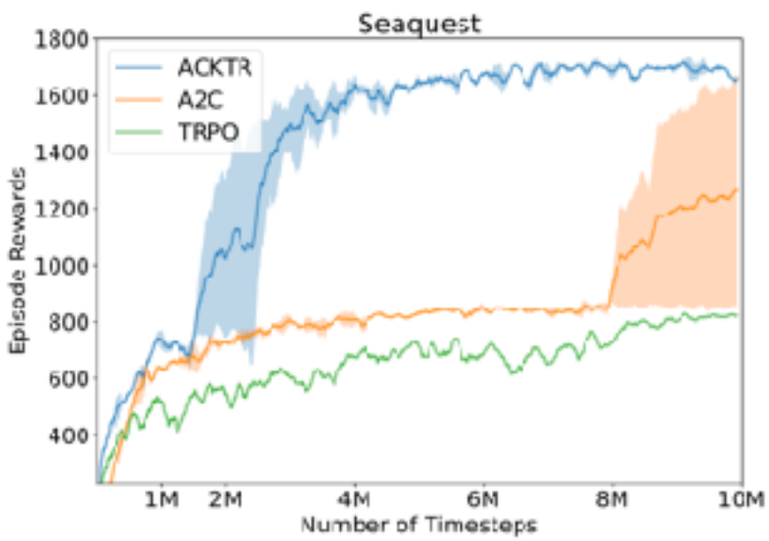
Atari games:

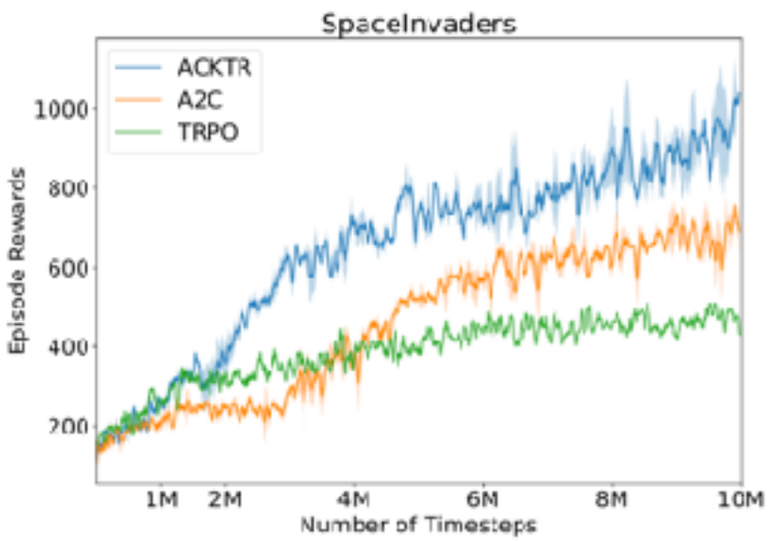






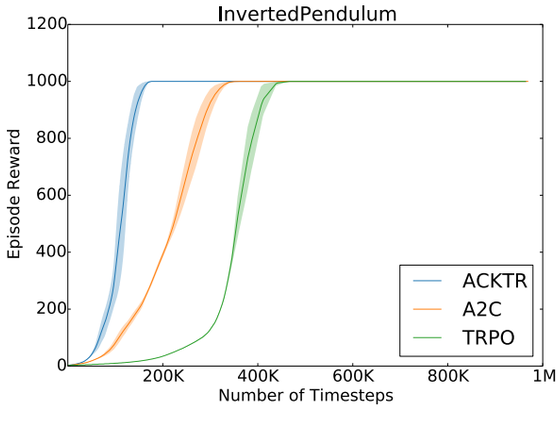


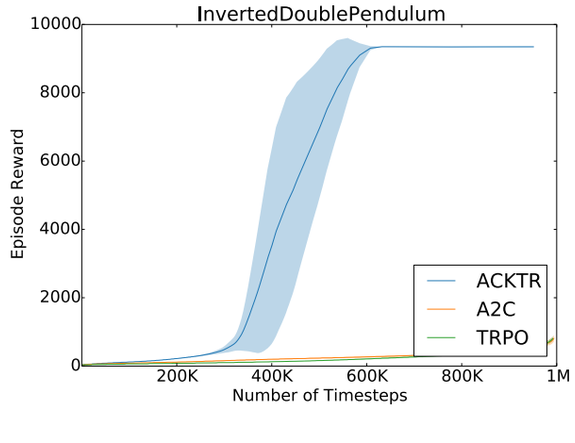


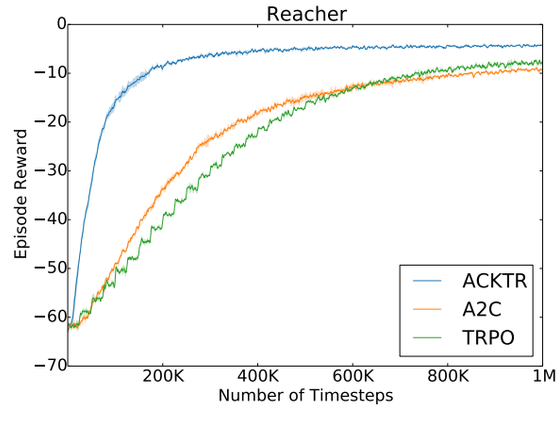


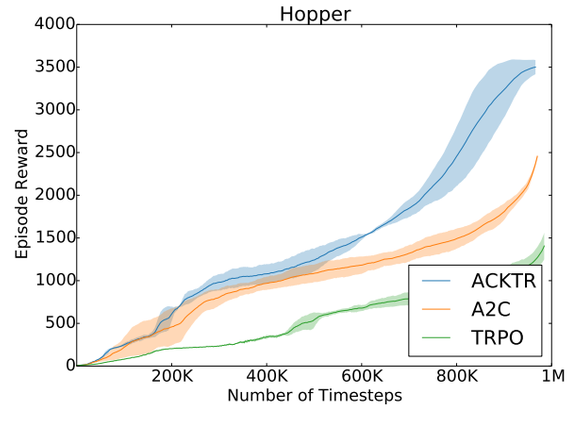
Reinforcement Learning

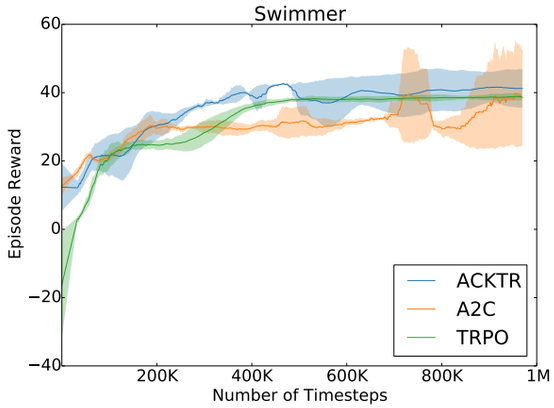
MuJoCo (state space)

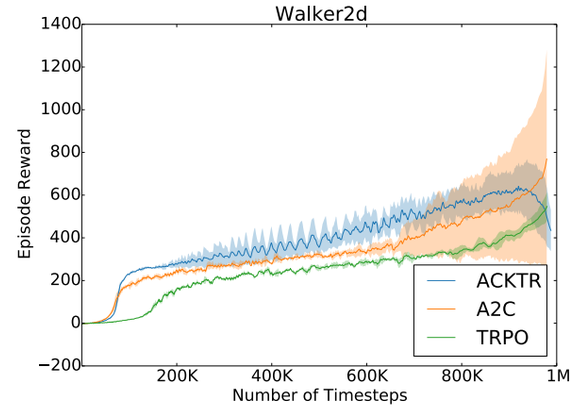


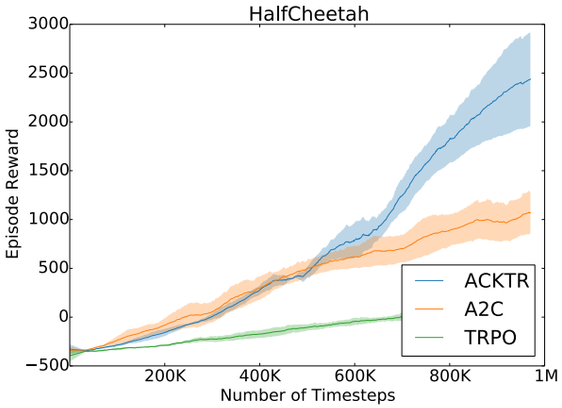


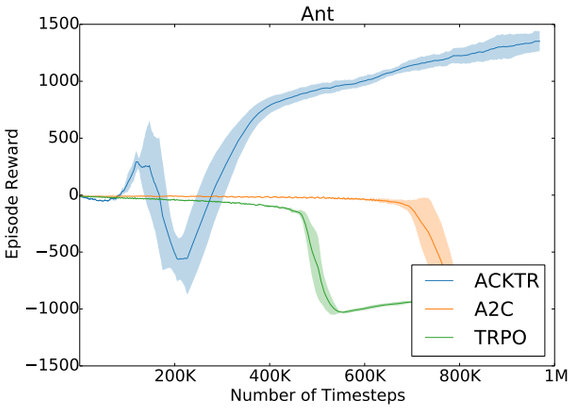






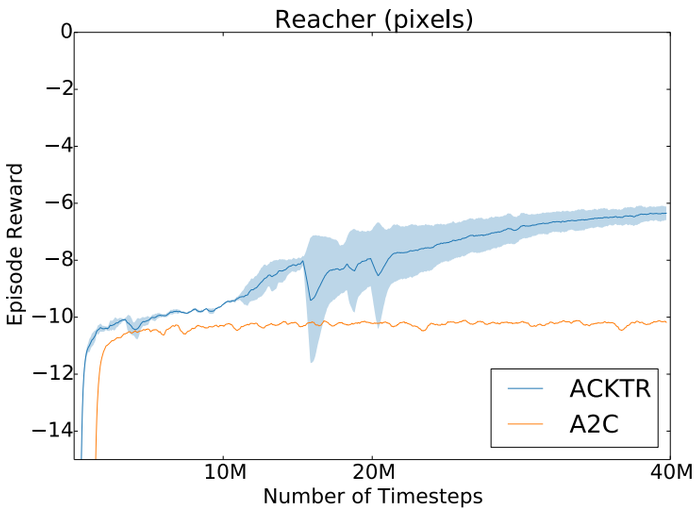


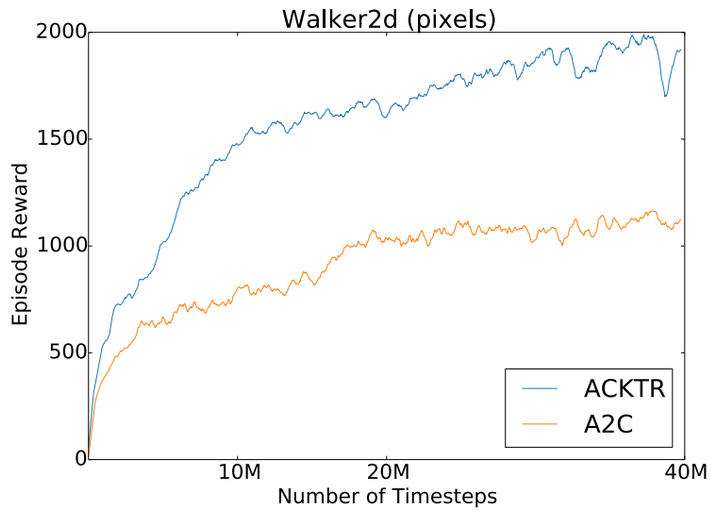


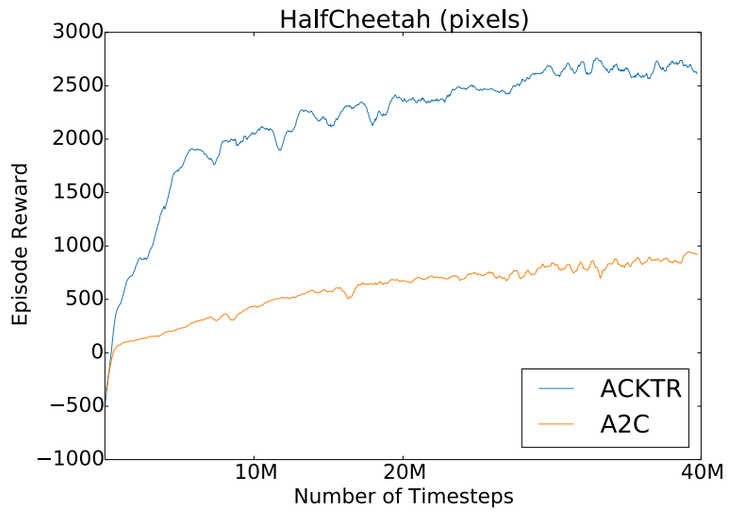


Reinforcement Learning

MuJoCo (pixels)







Noisy natural gradient as variational inference

w/ Guodong Zhang and Shengyang Sun

Two kinds of natural gradient

- We've covered two kinds of natural gradient in this course:
  - Natural gradient for point estimation (as in K-FAC)
    - Optimization variables: weights and biases
    - Objective: expected log-likelihood
    - Uses (approximate) Fisher matrix for the model's predictive distribution

$$\mathbf{F} = \underset{\mathbf{x} \sim p_{\text{data}}, y \sim p(y|\mathbf{x};\boldsymbol{\theta})}{\text{Cov}} \left(\frac{\partial \log p(y|\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)$$

  - Natural gradient for variational Bayes (Hoffman et al., 2013)
    - Optimization variables: parameters of variational posterior
    - Objective: ELBO
    - Uses (exact) Fisher matrix for variational posterior

$$\mathbf{F} = \underset{\boldsymbol{\theta} \sim q(\boldsymbol{\theta};\boldsymbol{\phi})}{\text{Cov}} \left(\frac{\partial \log q(\boldsymbol{\theta};\boldsymbol{\phi})}{\partial \boldsymbol{\phi}} \right)$$

Natural gradient for the ELBO

- Surprisingly, these two viewpoints are closely related.
- Assume a multivariate Gaussian posterior $q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$
- Gradients of the ELBO

$$\nabla_{\boldsymbol{\mu}} \mathcal{F} = \mathbb{E}\left[\nabla_{\boldsymbol{\theta}} \log p(\mathcal{D} \mid \boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta})\right]$$

$$\nabla_{\boldsymbol{\Sigma}} \mathcal{F} = \frac{1}{2}\mathbb{E}\left[\nabla^2_{\boldsymbol{\theta}} \log p(\mathcal{D} \mid \boldsymbol{\theta}) + \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta})\right] + \frac{1}{2}\boldsymbol{\Sigma}^{-1}$$

- Natural gradient updates (after a bunch of math):

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \alpha \boldsymbol{\Lambda}^{-1}\left[\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x}; \boldsymbol{\theta}) + \frac{1}{N}\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta})\right]$$

stochastic Newton-Raphson update for weights

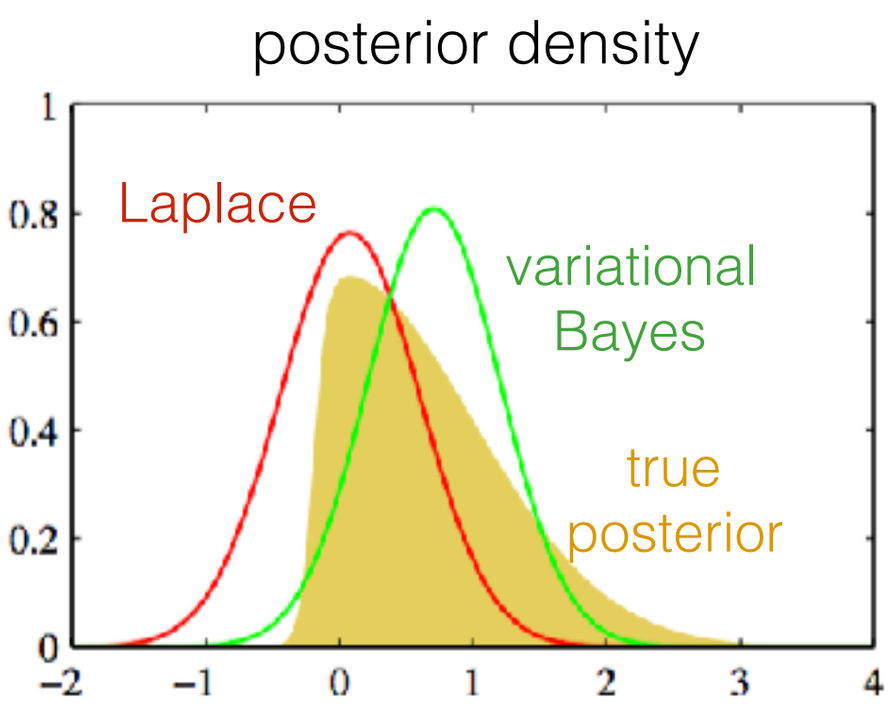
$$\boldsymbol{\Lambda} \leftarrow \left(1 - \frac{\beta}{N}\right)\boldsymbol{\Lambda} - \beta\left[\nabla^2_{\boldsymbol{\theta}} \log p(y|\mathbf{x}; \boldsymbol{\theta}) + \frac{1}{N}\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta})\right]$$

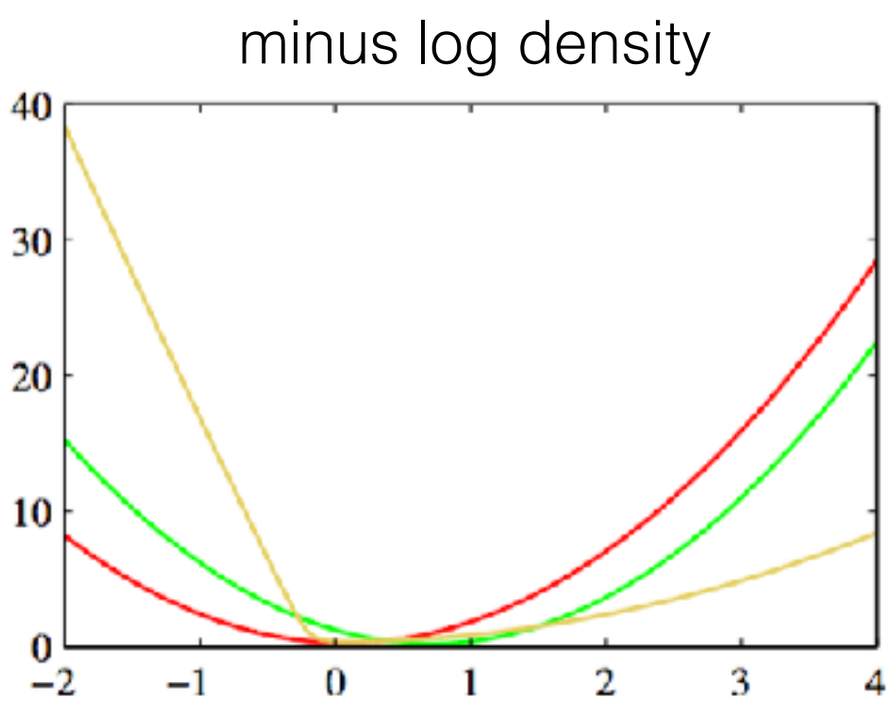
exponential moving average of the Hessian

- Note: these are evaluated at $\boldsymbol{\theta}$ sampled from q

Natural gradient for the ELBO

- Related: Laplace approximation vs. variational Bayes
- So it's not too surprising that $\mathbf{\Lambda}$ should look something like $\mathbf{H}^{-1}$





posterior density

minus log density

(Bishop, PRML)

Natural gradient for the ELBO

- Recall: under certain assumptions, the Fisher matrix (for point estimates) is approximately the Hessian of the negative log-likelihood:
  - The Hessian is approximately the GGN matrix if the prediction errors are small
  - The GNN matrix equals the Fisher if the output layer is the natural parameters of an exponential family
- Recall: Graves (2011) approximated the stochastic gradients of the ELBO by replacing the log-likelihood Hessian with the Fisher.
- Applying the Graves approximation, natural gradient SVI becomes natural gradient for the point estimate, with a moving average of F, and weight noise.

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \alpha \boldsymbol{\Lambda}^{-1} \left[\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta}) + \frac{1}{N} \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}) \right]$$

$$\boldsymbol{\Lambda} \leftarrow \left(1 - \frac{\beta}{N}\right) \boldsymbol{\Lambda} - \beta \left[\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta})^\top + \frac{1}{N} \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}) \right]$$

for a spherical Gaussian prior, this term is a multiple of **I**, so it acts as a damping term.

Natural gradient for the ELBO

- A slight simplification of this algorithm:

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \tilde{\alpha}\left(\overline{\mathbf{F}} + \frac{1}{N\eta}\mathbf{I}\right)^{-1}\left[\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta}) - \frac{1}{N\eta}\boldsymbol{\theta}\right]$$

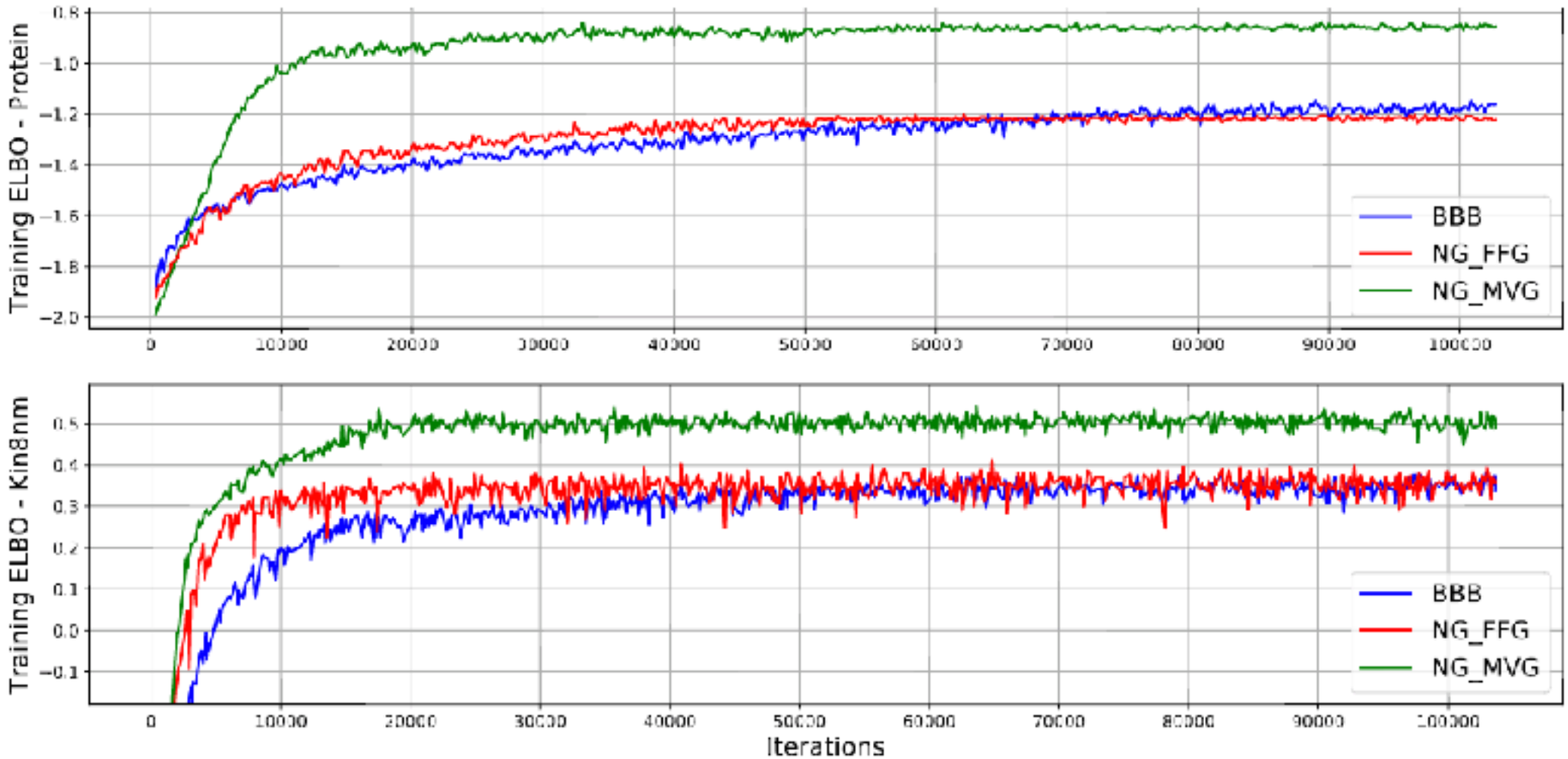
$$\overline{\mathbf{F}} \leftarrow (1-\tilde{\beta})\overline{\mathbf{F}} + \tilde{\beta}\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} \log p(y|\mathbf{x};\boldsymbol{\theta})^\top$$

- Hence, both the weight updates and the Fisher matrix estimation are viewed as natural gradient on the same ELBO objective.

- What if we plug in approximations to G?
  - Diagonal F
    - corresponds to a fully factorized Gaussian posterior, like Graves (2011) or Bayes By Backprop (Blundell et al., 2015)
    - update is like Adam with adaptive weight noise
  - K-FAC approximation
    - corresponds to a matrix-variate Gaussian posterior for each layer
    - captures posterior correlations between different weights
    - update is like K-FAC with correlated weight noise

Preliminary Results: ELBO

BBB: Bayes by Backprop (Blundell et al., 2015)
NG_FFG: natural gradient for fully factorized Gaussian posterior (same as BBB)
NG_MVG: natural gradient for matrix-variate Gaussian model (i.e. noisy K-FAC)



NG_FFG performs about the same as **BBB** despite the Graves approximation.

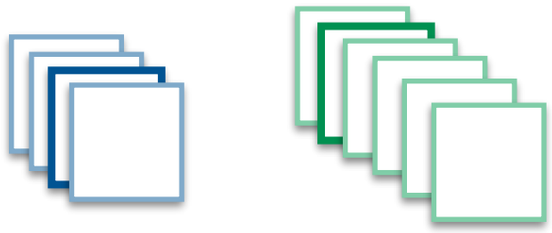
NG_MVG achieves a higher ELBO because of its more flexible posterior, and also trains pretty quickly.

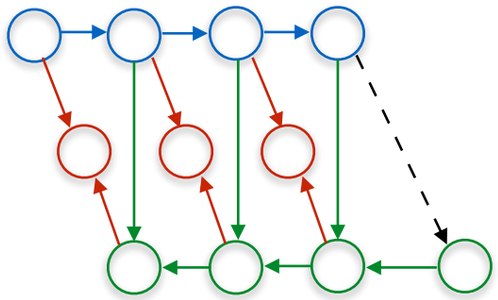
Preliminary Results: regression tasks

	Test RMSE			Test log-likelihood		
Dataset	BBB	NG_FFG	NG_MVG	BBB	NG_FFG	NG_MVG
Boston	2.517±0.022	2.396±0.016	**2.296±0.029**	-2.500±0.004	-2.430±0.004	**-2.336±0.005**
Concrete	5.770±0.066	5.916±0.053	**5.173±0.070**	-3.169±0.011	-3.166±0.009	**-3.073±0.014**
Energy	0.499±0.019	0.749±0.130	**0.438±0.003**	-1.552±0.006	-1.601±0.062	**-1.411±0.002**
Kin8nm	0.079±0.001	0.078±0.001	**0.076±0.000**	1.118±0.004	1.130±0.008	**1.151±0.006**
Naval	**0.000±0.000**	**0.000±0.000**	**0.000±0.000**	6.431±0.082	6.435±0.065	**7.182±0.057**
Pow. Plant	4.224±0.007	4.220±0.005	**4.085±0.006**	-2.851±0.001	-2.851±0.002	**-2.818±0.002**
Protein	4.390±0.009	4.397±0.009	**4.058±0.006**	-2.900±0.002	-2.900±0.002	**-2.820±0.002**
Wine	0.639±0.002	0.637±0.001	**0.634±0.001**	-0.971±0.003	-0.968±0.001	**-0.961±0.001**
Yacht	0.983±0.055	1.221±0.069	**0.827±0.017**	-2.380±0.004	-2.393±0.007	**-2.274±0.003**
Year	9.076±NA	9.078±NA	**8.885±NA**	-3.614±NA	-3.620±NA	**-3.595±NA**

Conclusions

- Approximate natural gradient by fitting probabilistic models to the gradient computation
    - check modeling assumptions empirically
- Invariant to most of the reparameterizations you actually care about
- Low (e.g. 50%) overhead compared to SGD
- Estimate curvature online using the entire dataset
- Consistent 3x improvement on lots of kinds of networks





Thank you!